

1 Computer hardware

Most computers are organized as shown in Figure 1.1. A computer contains several major subsystems --- such as the Central Processing Unit (CPU), memory, and peripheral device controllers. These components all plug into a "Bus". The bus is essentially a communications highway; all the other components work together by transferring data over the bus.

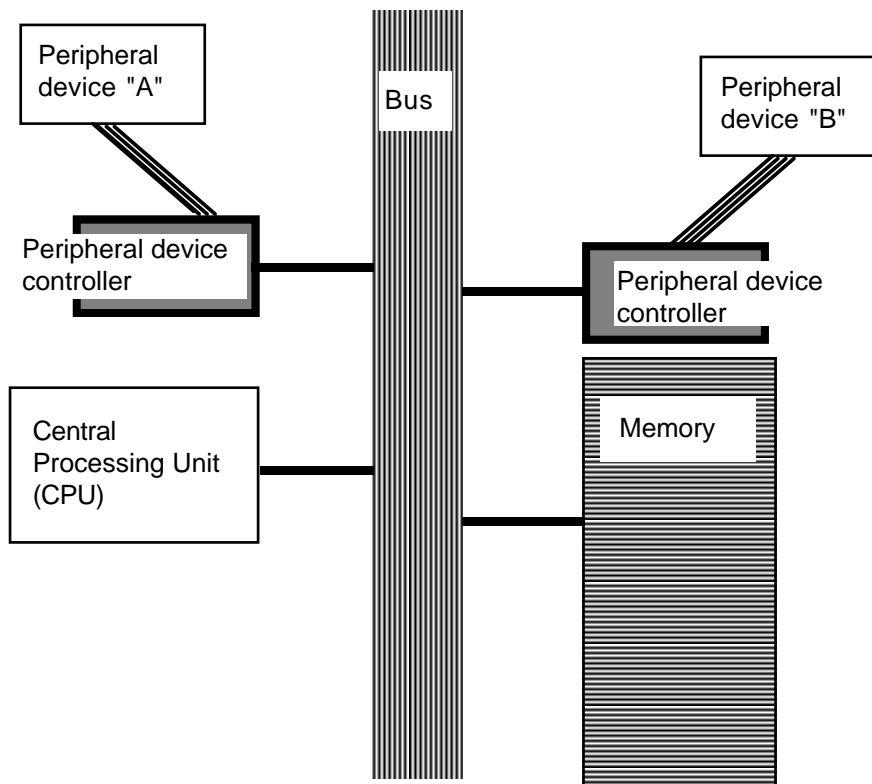


Figure 1.1 Schematic diagram of major parts of a simple computer.

The active part of the computer, the part that does calculations and controls all the other parts is the "Central Processing Unit" (CPU). The Central Processing Unit (CPU) contains electronic clocks that control the timing of all operations; electronic circuits that carry out arithmetic operations like addition and multiplication; circuits that identify and execute the instructions that make up a program; and circuits that fetch the data from memory.

Instructions and data are stored in main memory. The CPU fetches them as needed.

Peripheral device controllers look after input devices, like keyboards and mice, output devices, like printers and graphics displays, and storage devices like disks. The CPU and peripheral controllers work together to transfer information between the computer and its users. Sometimes, the CPU will arrange for data be taken from an input device, transfer through the controller, move over the bus and get loaded directly into the CPU. Data being output follows the same route in reverse – moving from the CPU, over the bus, through a controller and out to a device. In other cases, the CPU may get a device controller to move data directly into, or out of, main memory.

1.1 CPU AND INSTRUCTIONS

The CPU of a modern small computer is physically implemented as single silicon "chip". This chip will have engraved on it the million or more transistors and the interconnecting wiring that define the CPU's circuits. The chip will have one hundred or more pins around its rim --- some of these pins are connection points for the signal lines from the bus, others will be the points where electrical power is supplied to the chip.

Although physically a single component, the CPU is logically made up from a number of subparts. The three most important, which will be present in every CPU, are shown schematically in Figure 1.2.

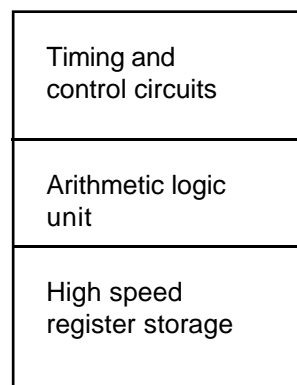


Figure 1.2 Principal components of a CPU.

The timing and control circuits are the heart of the system. A controlling circuit defines the computer's basic processing cycle:

Timing and control circuits

```
repeat
  fetch next instruction from memory
  decode instruction (i.e. determine which data
    manipulation circuit is to be activated)
  fetch from memory any additional data that are needed
  execute the instruction (feed the data to the
    appropriate manipulation circuit)
until "halt" instruction has been executed;
```

fetch, decode, execute cycle

Along with the controlling "fetch-decode-execute" circuit, the timing and control component of the CPU contains the circuits for decoding instructions and decoding addresses (i.e. working out the location in memory of required data elements).

The arithmetic logic unit (ALU) contains the circuits that manipulate data. There will be circuits for arithmetic operations like addition and multiplication. Often there will be different versions of such circuits – one version for integer numbers and a second for real numbers. Other circuits will implement comparison operations that permit a program check whether one data value is greater than or less than some other value. There will also be "logic" circuits that directly manipulate bit pattern data.

Arithmetic logic unit

While most data are kept in memory, CPUs are designed to hold a small amount of data in "registers" (data stores) in the CPU itself. It is normal for main memory to be large enough to hold millions of data values; the CPU may only have space for something like 16 values. A CPU register will hold as many bits as a "word" in the computer's memory. Bits, bytes, words etc are described more in section 1.2. Most current CPUs have registers that each store 32 bits of data.

High speed register storage

The circuits in the ALU often are organized so that some or all of their inputs and outputs must come from, or go to, CPU registers. Data values have to be fetched from memory and stored temporarily in CPU registers. Only then can they be combined using an ALU circuit, with the result again going to a register. If the result is from the final step in a calculation, it gets stored back into main memory.

While some of the CPU registers are used for data values that are being manipulated, others may be reserved for calculations that the CPU has to do when it is working out where in memory particular data values are to be stored.

CPU designs vary with respect to their use of registers. But, commonly, a CPU will have 8 or more "data" registers and another 8 "address" registers. Programmers who write in low-level "assembly languages" (Chapter 2) will be aware of these data and address registers in the CPU. Assembly language code defines details such as how data should be moved to specific data registers and how addresses are to be calculated and saved temporarily in address registers. Generally, programmers working with high level languages (Chapter 4) don't have to be concerned about such details; but, when necessary, a programmer can find out how the CPU registers are used in their code.

"Data registers" and "address registers"

In addition to the main data and address registers, the CPU contains many other registers, see Figure 1.3. The ALU will contain numerous registers for holding temporary values that are generated as arithmetic operations are performed. The

timing and control component contains a number of registers that hold control information.

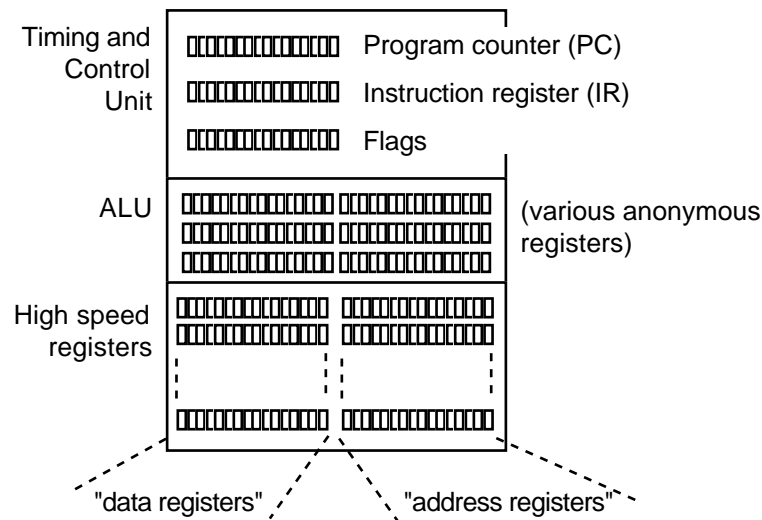


Figure 1.3 CPU registers.

Program Counter and Instruction Register

The Program Counter (PC) holds the address of the memory location containing the next instruction to be executed. The Instruction Register (IR) holds the bit pattern that represents the current instruction; different parts of the bit pattern are used to encode the "operation code" and the address of any data required.

Flags register

Most CPUs have a "flags" register. The individual bits in this register record various status data. Typically, one bit is used to indicate whether the CPU is executing code from an ordinary program or code that forms part of the controlling Operating Systems (OS) program. (The OS code has privileges; it can do things, which ordinary programs can not do, like change settings of peripheral device controllers. When the OS-mode bit is not set, these privileged instructions can not be executed.) Commonly, another group of bits in the flags register will be used to record the result of comparison instructions performed by the ALU. One bit in the flags register would be set if the comparator circuits found two values to be equal; a different bit would be set if the first of the two values was greater than the second.

Programs and instructions

Ultimately, a program has to be represented as a sequence of instructions in memory. Each instruction specifies either one data manipulation step or a control action. Normally, instructions are executed in sequence. The machine is initialized with the program counter holding the memory location of the first instruction from the program and then the fetch-decode-execute cycle is started. The CPU sends a fetch request to memory specifying the location specified by the PC (program counter); it receives back the instruction and stores this in the IR. The PC is then updated so that it holds the address of the next instruction in sequence. The instruction in the IR is then decoded and executed. Sometimes, execution of the instruction will change the contents of the PC. This can happen when one gets a

"branch" instruction (these instructions allow a program to do things like skip over processing steps that aren't required for particular data, or go back to the start of some code that must be repeated many times).

A CPU is characterized by its instruction repertoire – the set of instructions that can be interpreted by the circuits in the timing and control unit and executed using the arithmetic logic unit.. The Motorola 68000 CPU chip can serve as an example. The 68000 (and its variants like 68030 and 68040) were popular CPU chips in the 1980s being used in the Macintosh computers and the Sun3 workstations. The chip had, as part of its instruction repertoire, the following instructions:

Instruction repertoire

ADD	Add two integer values
AND	Perform an AND operation on two bit patterns
Bcc	Test a condition flag, and possibly branch to another instruction (variants like BEQ testing equality, BLT testing less than)
CLR	Clear, i.e. set to 0
CMP	Compare two values
JMP	Jump or goto
JSR	Call a subroutine
SUB	Subtract second value from first
RTS	Return from subroutine

(Instructions are usually given short "mnemonic" names – names that have been chosen to remind one of the effect achieved by the instruction, like ADD and CLear.)

mnemonic instruction names

Different CPU architectures, e.g. the Motorola 68000 and Intel-086 architectures, have different instruction sets. There will be lots of instructions that are common – ADD, SUB, etc. But each architecture will have its own special instructions that are not present on the other. Even when both architectures have similar instructions, e.g. the compare and conditional branch instructions, there may be differences in how these work.

Figure 1.4 is a simplified illustration of how instructions are represented inside a computer.

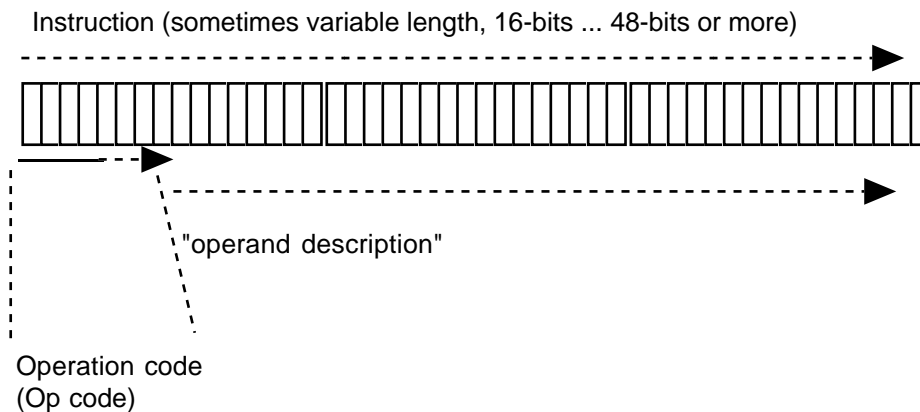


Figure 1.4 Simplified representation of an instruction inside a computer.

An instruction is represented by a set of bits. A few CPUs have fixed size instructions; on such machines, every instruction is 16-bits, or 32-bits or whatever. Most CPUs allow for different sizes of instructions. An instruction will be at least 16-bits in size, but may have an additional 16, 32, or more bits.

Op-code

The first few bits of an instruction form the "Op-code" (operation code). These bits identify the data manipulation or control operation required. Again CPUs vary; some use a fixed size op-code, most have several different layouts for instructions with differing numbers of bits allocated for the op-code. If a CPU uses a fixed size op-code, decoding is simple. The timing and control component will implement a form of multiway switch. Thus, if one had a 4-bit op-code, one could have a decoding scheme something like the following:

0000	Do an addition
0001	Do a subtraction
0010	Copy (move) some data
0011	Do an AND operation
...	
...	

The meaning of the remaining bits of an instruction depends on the actual instruction.

Operand(s)

Many instructions require that data be specified. Thus, an ADD instruction needs to identify which two values are to be summed, and must also specify a place where the result should be stored. Often some of this information can be implicit. An ADD instruction can be arranged so that the sum of the two specified values always replaces the first value wherever this was stored.

Although some data locations can be implicit, it is necessary to define either the source or destination locations for the other data. Sometimes a program will need to add numbers that are already held in data registers in the CPU; at other times, the program may need to fetch additional data from memory. So sometimes the "operand description" part of an add instruction will need to identify the two CPU data registers that are to be used; other times, the "operand description" will have to identify one CPU register and one memory location. Occasionally, the "operand description" part might be used to identify a CPU register and the *value* that is to be added to that register's existing contents.

Addressing modes

It is here that things get a bit complex. CPUs have many different ways of encoding information about the registers to be used, the addresses of memory locations, and the use of explicit data values. A particular machine architecture will have a set of "addressing modes" – each mode specifies a different way of using the bits of the operand description to encode details concerning the location of data values. Different architectures have quite different sets of addressing modes.

Some instructions don't need any data. For example, the "Bcc" (conditional branch) group instructions use only information recorded in the CPU's Flags register. These instructions have different ways of using the operand bits of instruction word. Often, as with the Bcc instructions, the operand bits encode an address of an instruction that is to be used to replace the current contents of the

program counter. Replacing the contents of the PC changes the next instruction executed.

1.2 MEMORY AND DATA

Computers have two types of memory:

ROM – Read Only Memory

RAM - normal Read Write Memory

The acronym RAM instead of RWM is standard. It actually standards for "Random Access Memory". Its origin is very old, it was used to distinguish main memory (where data values can be accessed in any order – hence "randomly") from secondary storage like tapes (where data can only be accessed in sequential order).

ROM memory is generally used to hold parts of the code of the computer's operating system. Some computers have small ROM memories that contain only a minimal amount of code just sufficient to load the operating system from a disk storage unit. Other machines have larger ROM memories that store substantial parts of the operating system code along with other code, such as code for generating graphics displays.

ROM memory

Most of the memory on a computer will be RAM. RAM memory is used to hold the rest of the code and data for the operating system, and the code and data for the program(s) being run on the computer.

RAM memory

Memory sizes may be quoted in bits, bytes, or words:

Bits, bytes, and words

Bit a single 0 or 1 data value

Byte a group of 8 bits

Word the width of the primary data paths between memory and the CPU, maybe 16-bit (two byte), 32-bit (four byte) or larger.

Memory sizes are most commonly given in terms of bytes. (The other units are less useful for comparative purposes. Bits are too small a unit of storage. Word sizes vary between machines and on some machines aren't really defined.) The larger memory units like bytes and words are just made up from groups of bits.

All storage devices require simple two-state components to store individual bits. Many different technologies have been used. Some early computers distinguished 0 and 1 bit values by the presence or absence of a pulse of energy moving through a tube of mercury; external storage was provided using paper media like cards or tapes where the presence or absence of a punched hole distinguished the 0/1 bit setting. Later, the most popular technology for a computer's main memory used small loops of magnetic oxide ("cores") that could be set with differing North/South polarity to distinguish the 0/1 bit state. Disks (and tapes) still use magnetic encoding – 0/1 bit values are distinguished by the magnetic state of spots of oxide on the disk's surface.

The main memories of modern computers are made from integrated circuits. One basic circuit is a "flip-flop". This uses four transistors wired together; it can be

set in an on or an off state and so can hold one bit. A more elaborate circuit, with eight flip-flops, can hold one byte. Repeated again and again, these can be built up into integrated circuits that hold millions of bytes. Individual memory chips with as much as 4 million bytes of storage capacity can now be purchased. A computer's memory will be made up out of several of these chips.

Figure 1.5 is a simplified illustration of memory for a machine with a 16-bit (two byte) word size.

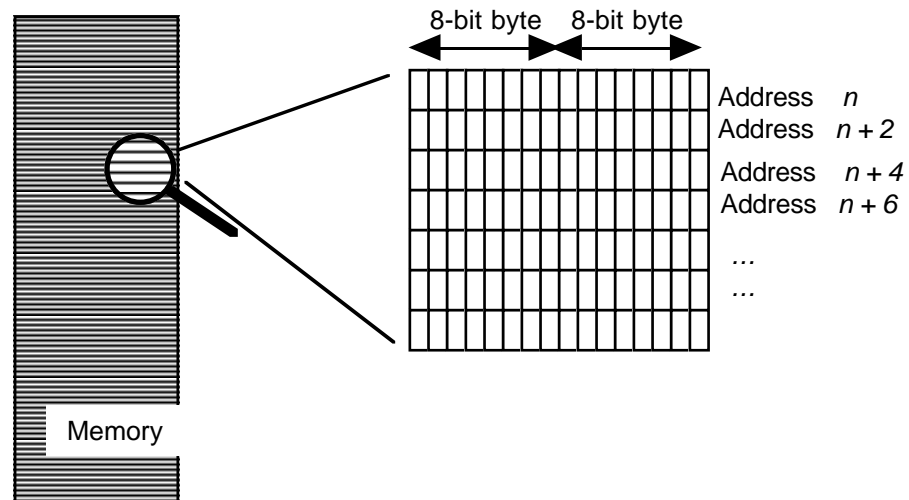


Figure 1.5 Memory organized in words with integer addresses.

Memory addresses

Memory can be viewed as a vector (one dimensional array) of words. The words can be identified by their positions (index number) in this vector – the integer index of a word is its "address". Most computers are designed to allow addressing of individual bytes. (If a request is made for a specific byte, the memory unit may return the entire word, leaving it to the CPU circuits to select the required byte). Because the individual bytes are addressable, word addresses increase in 2s (or in 4s if it is a machine with 4 byte words).

The amount of memory available on a computer has increased rapidly over the last few years. Most current personal computers now have around 8 million bytes of storage (8 megabyte, 8MB); more powerful workstations have from 32MB to 256MB and large time shared systems may have 1000 MB (or 1gigabyte).

Cache memories

"Cache" memories are increasingly common ("cache – a hiding place for provisions, treasures etc"). Cache memories are essentially hidden from the applications programmer; the cache belongs to the computer hardware and its controlling operating system. These work together using a cache to increase performance. Currently, a typical cache memory would be up to 256 KB in size. The cache may form a part of the circuitry of the CPU chip itself, or may be a separate chip. Either way, the system will be designed so that information in the cache can be accessed much more quickly than information in main storage.

The OS and CPU hardware arrange to copy blocks of bytes ("pages") from main memory into the cache. The selected pages could be those with the instructions currently being executed. Most programs involve loops where particular sets of instructions are executed repeatedly. If the instructions forming a loop are in the cache, the CPU's instruction-fetch operation is greatly speeded up. Sometimes it is worth copying pages with data from main memory to the cache – then subsequent data accesses are faster (though data that get changed do have to be copied back to main memory eventually). The operations shifting pages, or individual data elements, between cache and memory are entirely the concern of the CPU hardware and the operating system. The only way that a programmer should be able to detect a cache is by noticing increased system's performance.

All data manipulated by computers are represented by bit patterns. A byte, with 8 individual bits, can represent any of 256 different patterns; some are shown in Figure 1.6.

A set of 256 patterns is large enough to have a different pattern for each letter of the alphabet, the digits, punctuation characters, and a whole variety of special characters. If a program has to work with textual data, composed of lots of individual characters, then each character can be encoded in a single byte. Of course there have to be conventions that assign a specific pattern to each different character. At one time, different computer manufacturers specified their own character encoding schemes. Now, most use a standard character encoding scheme known as ASCII (for American Standard Code for Information Interchange). Although standardized, the assignments of patterns to characters is essentially arbitrary; Figure 1.6 shows the characters for some of the illustrated bit patterns.

Data as bit patterns in memory

Character data

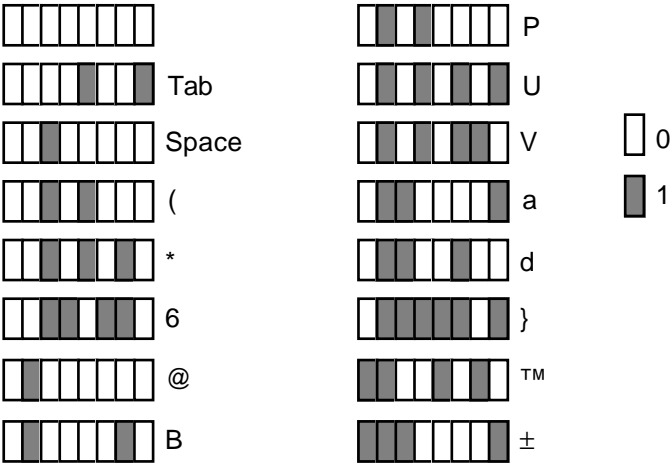


Figure 1.6 Some of the 256 possible bit patterns encodable in a single byte and the (printable) characters usually encoded by these patterns.

This ASCII scheme is mandated by an international standard. It specifies the bit patterns that should be used to encode 128 different characters including all the letters of the Roman alphabet, digits, punctuation marks and a few special control characters like "Tab". The remaining 128 possible patterns (those starting with a 1

ASCII character codes

in the leftmost bit of the byte) are not assigned in the standard. Some computer systems may have these patterns assigned to additional characters like TM, \pm , ϕ , \ddagger .

Numeric data

Bit patterns can also be used to represent numbers. Computers work with integer numbers and "floating point" numbers. Floating point numbers are used to approximate the real numbers of mathematics.

Integer values

A single byte can only be used to encode 256 different values. Obviously, arithmetic calculations are going to work with wider ranges – like -2,000,000,000 to +2,000,000,000. Many more bits are needed to represent all those different possible values. All integer values are represent using several bytes. Commonly, CPUs are designed to work efficiently with both two-byte integers and four-byte integers (the CPU will have two slightly different versions of each of the arithmetic instructions). Two-byte integers are sufficient if a program is working with numbers in the range from about minus thirty thousand to plus thirty thousand; the four-byte integers cover the range from minus to plus two thousand million.

Figure 1.7 shows the common representations of a few integers when using two bytes. The number representations have an obvious regular pattern. Unlike the case of character data, the patterns used to represent integers can not be arbitrary. They have to follow regular patterns in order to make it practical to design electronic circuitry that can combine patterns and achieve effects equivalent to arithmetic operations. The code scheme that provides the rules for representing numbers is known as "two's complement notation"; this scheme is covered in introductory courses on computer hardware.

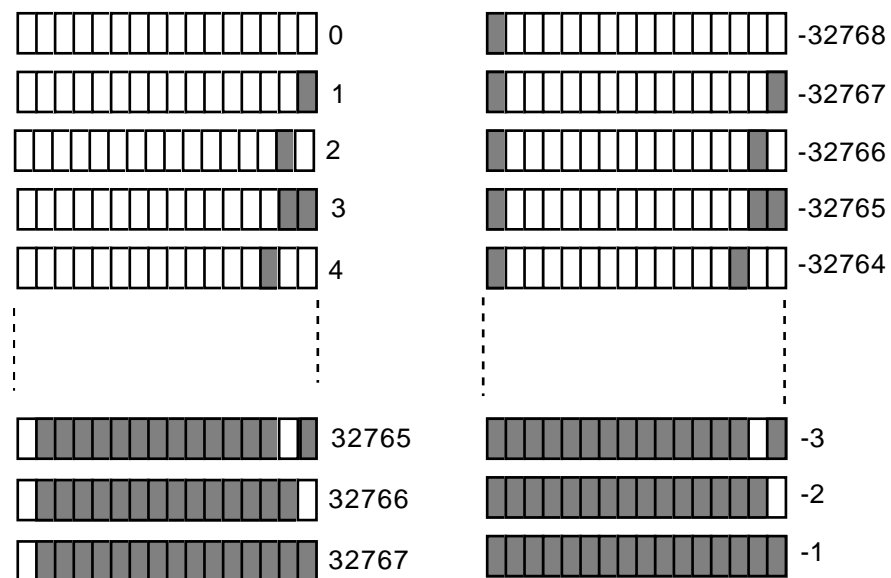


Figure 1.7 Representing integers in two byte (16-bit) bit patterns in accord with the "two's complement notation scheme".

There are other coding schemes for integers but "two's complement notation" is the most commonly used. The actual coding scheme used to represent integers, and the resulting bit patterns, is not often of interest to programmers.

While a computer requires special bit patterns representations of numbers, strings of 0 and 1 characters are not appropriate for either output or input. Humans require numbers as sequences of digit characters. Every time a number is input to a program, or is printed by a program, some code has to be executed to translate between the binary computer representation and the digit string representation used by humans.

The electronic circuits in the CPU can process the binary patterns and correctly reproduce the effects of arithmetic operations. There is just one catch with integers – in the computer they are limited to fixed ranges.

Two-bytes are sufficient to represent numbers from -32767 to +32768 – and it is an error if a program generates a value outside this range. Mistakes are possible. Consider for example a program that is specified as only having to work with values in the range 0 to 25000; a programmer might reasonably choose to use two-byte integers to represent such values. However, a calculation like "work out 85% of 24760" could cause problems, even though the result (21046) is in range. If the calculation is done by multiplying 85 and 24760, the intermediate result 2104600 is out of range.

Integer overflow

Arithmetic operations that involve unrepresentable (out of range) numbers can be detected by the hardware – i.e. the circuits in the ALU. Such operations leave incorrect bit patterns in the result, but provide a warning by setting an "overflow" bit in the CPU's flags register. Commonly, computer systems are organized so that setting of the overflow bit will result in the operating system stopping the program with the error. The operating system will provide some error message. (The most common cause of "overflow" is division by zero – usually the result of a careless programming error or, sometimes, due to incorrect data entry.)

Floating point numbers are used to approximate real numbers. They are mainly used with engineering and scientific calculations. The computer schemes for floating point numbers are closely similar to normalized scientific notation:

Floating point numbers

Number	Normalized scientific representation
17.95	+1.795 E+01
-0.002116	-2.116 E-03
1.5	+1.5 E+00
31267489.2	+3.12674892 E+07

The normalized scientific notation has a sign, a "mantissa", and a signed "exponent" – like '+', 1.795, and E+01. Floating point representations work in much the same way (you can think of floating point as meaning that the exponent specifies how far the "decimal" point need to be moved, or floated, to the left or the right).

A floating point number will be allocated several bytes (at least four bytes, usually more). One bit in the first byte is used for the sign of the number. Another group of bits encode the (signed) exponent. The remaining bits encode the

mantissa. Of course, both exponent and mantissa are encoded using a binary system rather than a decimal system.

Since the parts of a floating point number are defined by regular encoding rules, it is again possible to implement "floating point" arithmetic circuits in the ALU that manipulate them appropriately. Like integers, floating point numbers can overflow. If the number has an exponent that exceeds the range allowed, then overflow occurs. As with integers, this is easy to detect (and is most often due to division by zero). But with floating point numbers, there is another catch – a rather more pervasive one than the "overflow" problem.

Round-off errors

The allocation of a fixed number of bits for the mantissa means that only certain numbers are accurately represented. When four bytes are used, the mantissas are accurate to about eight decimal digits. So, while it might be possible to represent numbers like 0.81246675 and 0.81246676 exactly, all the values in the range 0.81246675.. to 0.812466764... have to be approximated by the nearest number that can be represented exactly i.e. 0.81246676. Any remaining digits, in the 9th and subsequent places in the fraction, are lost in this rounding off process.

Each floating point operation that combines two values will finish by rounding off the result to the nearest representable value. It might seem that loss of one part in a hundred million is not important. Unfortunately, this is not true. Each calculation step can introduce such errors – and a complete calculation can involve millions of steps in which the errors *may* combine and grow. Further, it is quite common for scientists and engineers to be trying to calculate the small difference between two large values – and in these cases the "rounded off" parts may be comparable to the final result.

While integer overflow errors are easily detected by hardware and are obvious errors, round off errors can not be dealt with so simply. Those needing to work extensively with floating point numbers really need to take a numerical analysis course that covers the correct way of organizing calculations so as to minimize the effects of cumulative round off errors.

Instructions – just another bit pattern

As explained in section 1.1, instructions are represented as bit patterns – maybe 16 bits (two bytes) in length, possibly longer. In memory, instructions are stored in a sequence of successive bytes.

All just bit patterns

A few experimental computers have been built where every word in memory had something like 2 or 3 extra bits that tagged the type of data stored in that word. These "tagged memory architecture" machines might have used code 00 to mark a word containing an instruction, 01 if it contained integer data etc. Such computers are atypical. On most computers, bit patterns in memory have no "type", no intrinsic meaning. The meaning of a bit pattern is determined by the circuit of the CPU that interprets it; so if it ends up in the IR (instruction register) it gets interpreted as representing an instruction, while if it goes to an (integer) addition circuit in the ALU it is interpreted as an integer. It is possible (though uncommon) to make programming errors so that data values fetched from memory to be interpreted as instructions or, alternatively, for a program to start storing results of calculations in those parts of its memory that hold the instruction sequence. If a program contains such gross errors, it usually soon attempts an illegal operation (like attempting to execute a bit pattern that can not be recognized as a valid instruction) and so is stopped by the computer hardware and operating system.

1.3 BUS

A computer's bus can be viewed as consisting of about one hundred parallel wires; see Figure 1.8. Some of these wires carry timing signals, others will have control signals, another group will have a bit pattern code that identifies the component (CPU, memory, peripheral controller) that is to deal with the data, and other wires carry signals encoding the data.

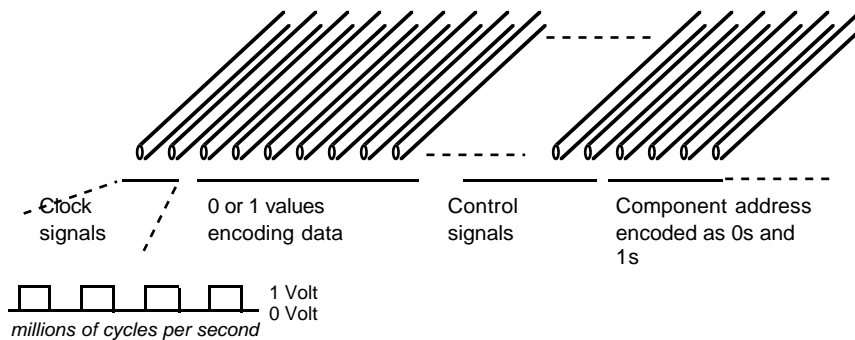


Figure 1.8 Bus.

Signals are sent over the bus by setting voltages on the different wires (the voltages are small, like 0-volts and 1-volt). When a voltage is applied to a wire the effect propagates along that wire at close to the speed of light; since the bus is only a few inches long, the signals are detectable essentially instantaneously by all attached components. Transmission of information is controlled by clocks that put timing signals on some of the wires. Information signals are encoded on to the bus, held for a few clock ticks to give all components a chance to recognize and if appropriate take action, then the signals are cleared. The clock that controls the bus may be "ticking" at more than one hundred million ticks per second.

The "plugs" that attach components to the bus incorporate quite sophisticated circuits. These circuits interpret the patterns of 0/1 voltages set on the control and address lines – thus memory can recognize a signal as "saying" something like "store the data at address xxx", while a disk control unit can recognize a message like "get ready to write to disk block identified by these data bits". In addition, these circuits deal with "bus arbitration". Sometimes, two or more components may want to put signals on the bus at exactly the same time – the bus arbitration circuitry resolves such conflicts giving one component precedence (the other component waits a few hundred millionths of a second and then gets the next chance to send its data).

1.4 PERIPHERALS

There are two important groups of input/output (i/o) devices. There are devices that provide data storage, like disks and tapes, and there are devices that connect the computer system to the external world (keyboards, printers, displays, sensors).

The storage devices record data using the same bit pattern encodings as used in the memory and CPU. These devices work with blocks of thousands of bytes. Storage space is allocated in these large units. Data transfers are in units of "blocks".

The other i/o devices transfer only one, or sometimes two, bytes of data at a time. Their controllers have two parts. There is a part that attaches to the bus and has some temporary storage registers where data are represented as bit patterns. A second part of the controller has to convert between the internal bit representation of data and its external representation. External representations vary – sensors and effectors (used to monitor and control machinery in factories) use voltage levels, devices like simple keyboards and printers may work with timed pulses of current, some devices use flashes of light.

1.4.1 Disks and tapes

Most personal computers have two or three different types of disk storage unit. There will be some form of permanently attached disk (the main "hard disk"), some form of exchangeable disk storage (a "floppy disk" or possibly some kind of cartridge-style hard disk), and there may be a CD-ROM drive for read-only CD disks.

Optical disks CD disks encode 0 and 1 data bits as spots with different reflectivity. The data can be read by a laser beam that is either reflected or not reflected according to the setting of each bit of data; the reflected light gets converted into a voltage pulse and hence the recorded 0/1 data values gets back into the form needed in the computer circuits. Currently, optical storage is essentially read-only – once data have been recorded they can't be changed. Read-write optical storage is just beginning to become available at reasonable prices.

Magnetic disks Most disks use magnetic recording. The disks themselves may be made of thin plastic sheets (floppy disks), or ceramics or steel (hard disks). Their surfaces are covered in a thin layer of magnetic oxide. Spots of this magnetic oxide can be magnetically polarized. If a suitably designed wire coil is moved across the surface, the polarized spots induce different currents in the coil – allowing data to be read back from the disk. New data can be written by moving a coil across the surface with a sufficiently strong current flowing to induce a new magnetic spot with a required polarity. There is no limit on the number of times that data can be rewritten on magnetic disks.

Tracks The bits are recorded in "tracks" – these form concentric rings on the surface of the disk, see Figure 1.9. Disks have hundreds of these tracks. (On simple disk units, all tracks hold the same number of bits; since the outermost tracks are slightly longer, their bits are spaced further apart than those on the innermost

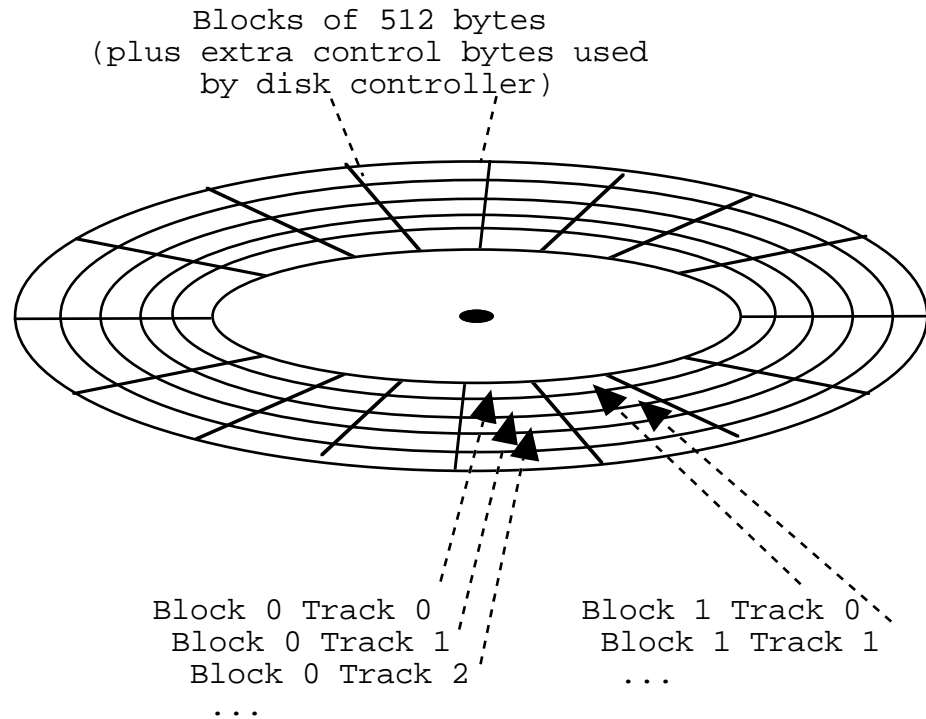


Figure 1.10 Tracks divided into storage blocks.

"Seeking" for tracks

Before data blocks can be read or written, the read/write head mechanism must be moved to the correct track. The read/write head contains the coil that detects or induces magnetism. It is moved by a stepping motor that can align it accurately over a specific track. Movements of the read/write heads are, in computer terms, relatively slow – it can take a hundredth of a second to adjust the position of the read/write heads. (The operation of moving the heads to the required track is called "seeking"; details of disk performance commonly include information on "average seek times".) Once the head is aligned above the required track, it is still necessary for the spinning disk to bring the required block under the read/write head (the disk controller reads its control information from the blocks as they pass under the head and so "knows" when the required block is arriving). When the block arrives under the read/write head, the recorded 0/1 bit values can be read and copied to wherever else they are needed.

Disk cache memory

The read circuitry in the disk reassembles the bits into bytes. These then get transferred over the bus to main memory (or, sometimes, into a CPU register). Disks may have their own private cache memories. Again, these are "hidden" stores where commonly accessed data can be kept for faster access. A disk may have cache storage sufficient to hold the contents of a few disk blocks (i.e. several thousand bytes). As well as being sent across the bus to memory, all the bytes of a block being read can be stored in the local disk cache. If a program asks the disk to

read a block of data that is in the cache, the disk unit doesn't need to seek for the data. The required bytes can be read from the cache and sent to main memory.

Commonly, hard disks have several disk platters mounted on a single central spindle. There are read/write heads for each disk platter. Data can be recorded on both sides of the disk platters (though often the topmost and bottommost surfaces are unused). The read/write heads are all mounted on the same stepping motor mechanism and move together between the disk platters, see Figure 1.11.

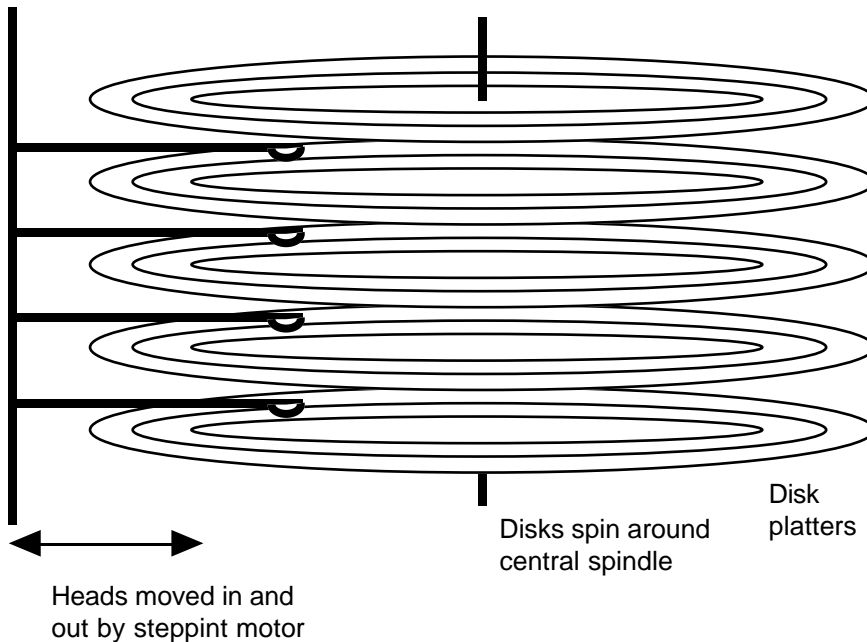


Figure 1.11 Read/write head assemble and multiplatter disks.

A controller for a disk is quite elaborate, see Figure 1.12. The controller will have several registers (similar to CPU registers) and circuitry for performing simple additions on (binary) integer numbers. (The cache memory shown is optional; currently, most disk controllers don't have these caches.) One register (or group of registers) will hold the disk address or "block number" of the data block that must be transferred. Another register holds a byte count; this is initialized to the block size and decremented as each byte is transferred. The disk controller stops trying to read bits when this counter reaches zero. The controller will have some special register used for grouping bits into bytes before they get sent over the bus. Yet another register holds the address of the (byte) location in memory that is to hold the next byte read from the disk (or the next byte to be written to disk).

Disk controller

Errors can occur with disk transfers. The magnetic oxide surface may have been damaged. The read process may fail to retrieve the data. The circuits in the disk can detect this but need some way of passing this information to the program that wanted the data. This is where the Flags register in the disk controller gets used. If something goes wrong, bits are set in the flags register to identify the

error. A program doing a disk transfer will check the contents of the flags register when the transfer is completed and can attempt some recovery action if the data transfer was erroneous.

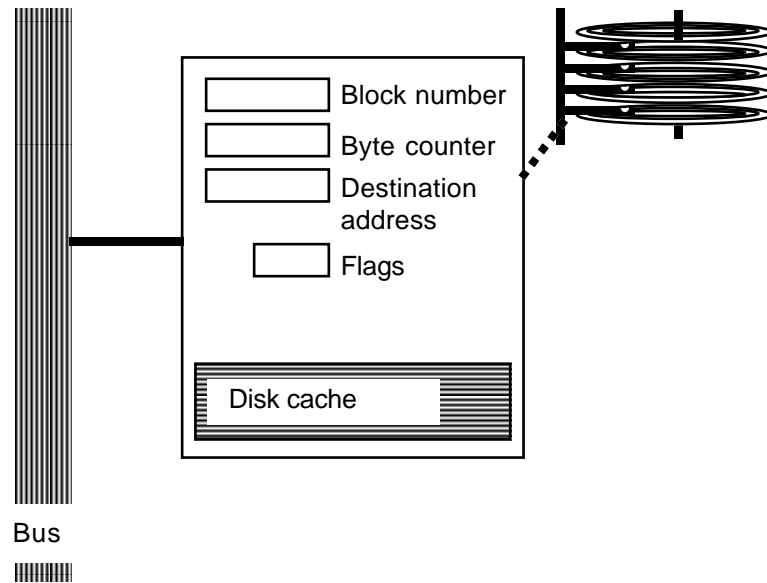


Figure 1.12 Disk controller.

This elaborate circuitry and register setup allows a disk controller to work with a fair degree of autonomy. Figure 1.13 illustrates the steps involved in a data transfer using such a disk and controller.

The data transfer process will start with the CPU sending a request over the bus to the disk controller; the request will cause the disk unit to load its block number register and to start its heads seeking to the appropriate track (step 1 in Figure 1.13).

It may take the disk a hundredth of a second to get its heads positioned (step 2 in Figure 1.13). During this time, the CPU can execute tens of thousands of instructions. Ideally, the CPU will be able to get on with other work, which it can do provided that it can work with other data that have been read earlier. At one time, programmers were responsible for trying to organize data transfers so that the CPU would be working on one block of data while the next block was being read. Nowadays, this is largely the responsibility of the controlling OS program.

When the disk finds the block it can inform the CPU which will respond by providing details of where the data are to be stored in memory (steps 3 and 4 in Figure 1.13).

Direct Memory Access

The disk controller can then transfer successive bytes read from the disk into successive locations in memory. A transfer that works like this is said to be using "direct memory access".

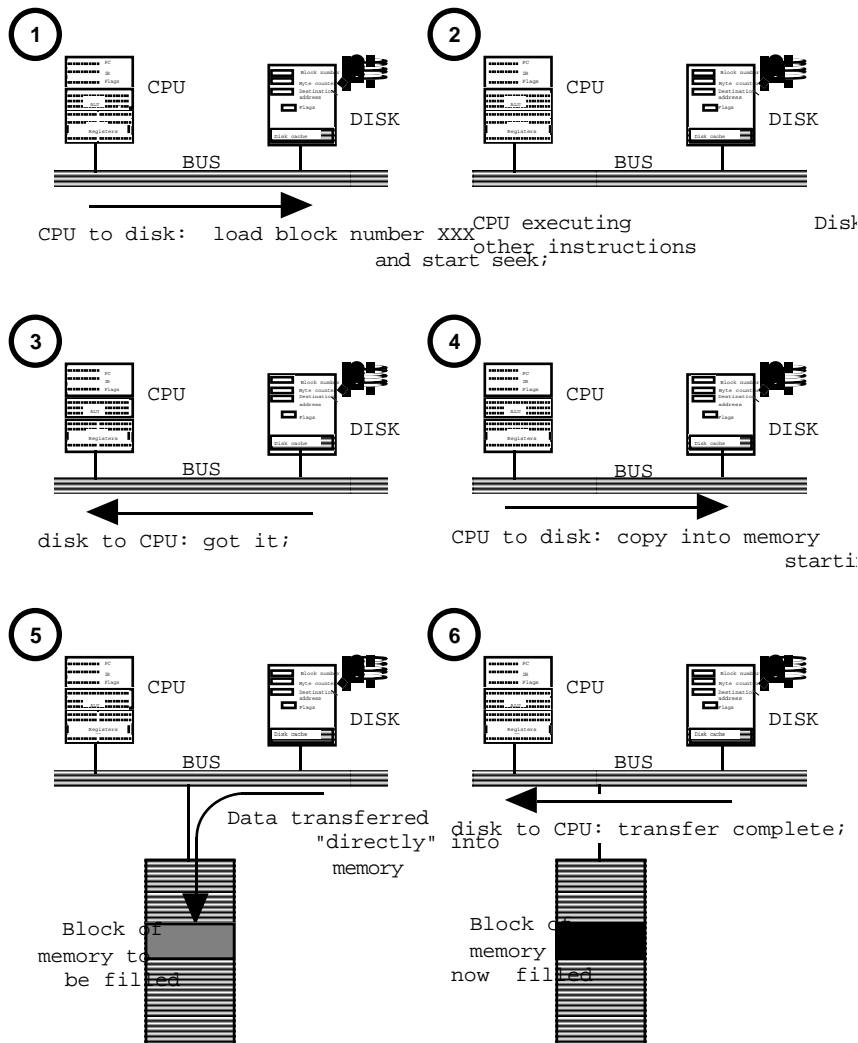


Figure 1.13 A disk data transfer using direct memory access.

The data transfer, step 5 in Figure 1.13, will take a little time (though very much less than the seek step). Once again, the CPU will be able to get on with other work.

When the transfer is complete, the disk controller will send another signal to the CPU, (step 6).

Data files on disk are made up out of blocks. For example, a text file with twelve thousand characters would need twenty four 512-byte blocks. (The last block would only contain a few characters from the file, it would be filled out with either space characters or just random characters). Programmers don't choose the blocks used for their files. The operating system is responsible for choosing the blocks used for each file, and for recording details for future reference.

Files

Figure 1.14 illustrates the simplest scheme used for block allocation. This scheme was sometimes used for small disks on early personal computers (late 1970s and early 1980s). All modern systems use more sophisticated schemes but the essence is the same.

File directory

As shown in Figure 1.14, a few blocks of the disk are reserved for a "file directory". The data in these blocks form a table of entries with each entry specifying a file name, file size (in bytes actually used and complete blocks allocated), and some record of which blocks are allocated. The allocation scheme shown in Figure 1.14 uses a group of contiguous blocks to make up each individual file. This makes it easy to record details of allocated blocks, the directory need only record the file size and the first block number.

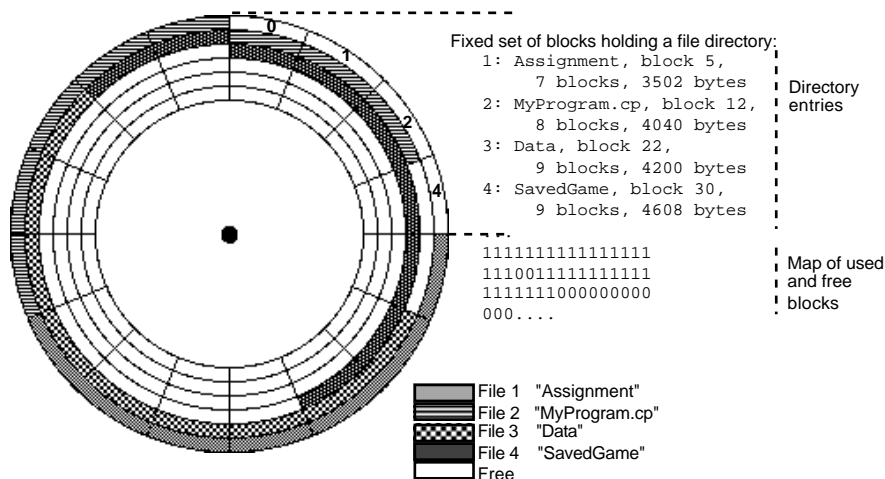


Figure 1.14 Simple file directory and file allocation scheme.

In addition to the table of entries describing allocated files, the directory structure would contain a record of which blocks were allocated and which were free and therefore available for use if another file had to be created. One simple scheme uses a map with one bit for each block; the bit is set if the block is allocated.

Tapes

Tapes are now of minor importance as storage devices for users' files. Mostly they are used for "archival" storage – recording data that are no longer of active interest but may be required again later. There are many different requirements for archival data. For example, government taxation offices typically stipulate that companies keep full financial record data for the past seven years; but only the current year's data will be of interest to a company. So, a company will have its current data on disk and hold the data for the other six years on tapes. Apart from archival storage, the main use of tapes is for backup of disk units. All the data on a computer's disks will be copied to tape each night (or, maybe just weekly). The tapes can be stored somewhere safe, remote from the main computer site. If there is a major accident destroying the disks, the essential data can be retrieved from tape and loaded on some other computer system.

The tape units used for most of the last 45 years are physically a bit like large reel-to-reel tape recorders. The tapes are about half an inch wide and two thousand feet in length and are run from their reel, through tensioning devices, across read-write heads, to a take up reel. The read write heads record 9 separate data tracks; these 9 tracks are used to record the 8-bits of a byte along with an extra check bit. Successive bytes are written along the length of the tape; an inch of tape could pack in as much as a few thousand bytes. Data are written to tape in blocks of hundreds, or thousands, of bytes. (On disks, the block sizes are now usually chosen by the operating system, the size of tape blocks is program selectable.) Blocks have to be separated by gaps where no data are recorded – these "inter record gaps" have to be large (i.e. half an inch or so) and they tend to reduce the storage capacity of a tape. Files are written to tape as sequences of blocks. Special "end of file" patterns can be recorded on tape to delimit different files.

A tape unit can find a file (identified by number) by counting end of file marks and then can read its successive data blocks. Data transfers are inherently sequential, block 0 of a file must be read before the tape unit can find block 1. Files cannot usually be rewritten to the same bit of tape – writing to a tape effectively destroys all data previously recorded further along the tape (the physical lengths of data blocks, interrecord gaps, file marks etc vary a little with the tension on the tape so there is no guarantee that subsequent data won't be overwritten). All the processes using tapes, like skipping to file marks, sequential reads etc, are slow.

Modern "streamer" tape units used for backing up the data on disks use slightly different approaches but again they are essentially a sequential medium. Although transfer rates can be high, the time taken to search for files is considerable. Transfers of individual files are inconvenient; these streamer tapes are most effective when used to save (and, if necessary restore) all the data on disk.

1.4.2 Other I/O devices

A keyboard and a printer are representative of simple Input/Output (I/O) peripheral devices. Such devices transfer a single data character, encoded as an 8-bit pattern, to/from the computer. When a key is pressed on the keyboard, internal electronics identifies which key was pressed and hence identifies the appropriate bit pattern to send to the computer. When a printer receives a bit pattern from the computer, its circuitry works out how to type or print the appropriate character.

Simple keyboards and printers

Figure 1.15 shows, in simplified form, the general structure of a controller for one of these simple i/o devices. The controller will have a 1-bit register for a "ready flag". This flag is set when the controller is ready to transfer data. There will be an 8-bit data register (or "buffer" register) that will hold the data byte that is to be transferred. The controller will incorporate whatever circuits are needed to convert the bit pattern data value into output voltages/light pulses/.... The controller will be connected to the actual device by some cable. This will have at least two wires; if there are only two wires, the bits of a byte are sent serially. Many personal computers have controllers for "parallel ports"; these have a group of 9 or more wires which can carry a reference voltage and eight signal voltages (and so can effectively transmit all the bits of a byte at the same time).

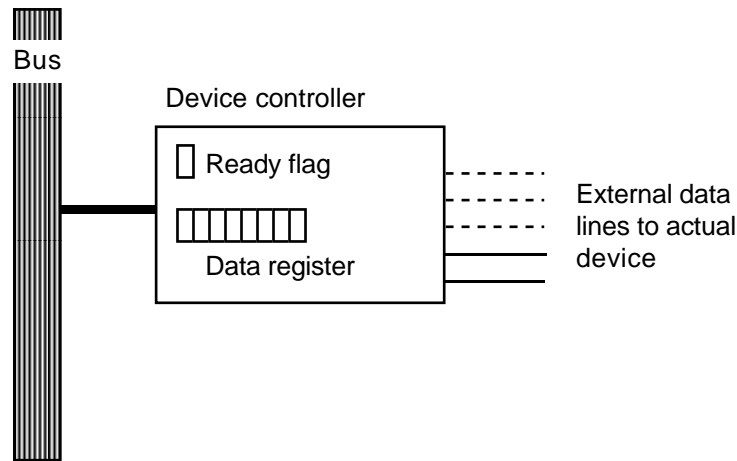


Figure 1.15 Controller for a simple i/o device such as a keyboard.

A simple approach to handling input from such a device is illustrated in Figure 1.16. The mechanism illustrated uses a wait loop – the program code makes the CPU wait until data arrive from the device. The program code would be:

```
repeat
    ask the device its status
until device replies "ready"

read data
```

The "repeat" loop would be encoded using three instructions. The first would send a message on the bus requesting to read the status of the device's ready flag. The instruction would cause the CPU to wait until the reply signal was received from the device. The second instruction would test the status data retrieved. The third instruction would be a conditional jump going back to the start of the loop. The loop corresponds to panes 1 and 2 in Figure 1.16; these two panes just show the exchange of signals caused by execution of the instruction.

When a key is pressed on the keyboard (pane 3 in Figure 1.16) the hardware in the keyboard identifies the key and sends its ASCII code as a sequence of voltage pulses. These pulses are interpreted by the controller which assembles the correct bit pattern in the controller's data register. When all 8 bits have been obtained, the controller will set the ready flag.

The next request to the device for its status will get a 1 reply (step 4). The program can continue with a "read data register" request which would copy the contents of the device's data register into a CPU register. If the character was to be stored in memory, another sequence of instructions would have to be executed to determine the memory address and then copy the bits from the CPU register into memory.

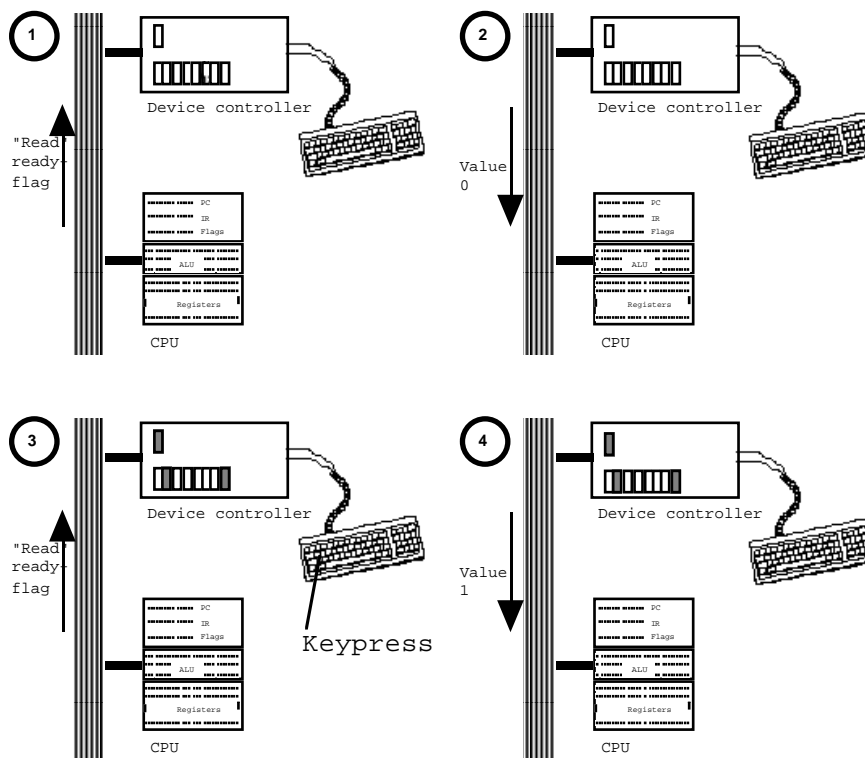


Figure 1.16 Input from a simple character-oriented device like a keyboard.

A wait loop like this is easy to code, but makes very poor use of CPU power. The CPU would spend almost all its time waiting for input characters (most computer users type very few characters per second). There are other approaches to organizing input from these low-speed character oriented devices. These alternative approaches are considerably more complex. They rely on the device sending an "interrupt signal" at the same time as it sets its ready flag. The program running on the CPU has to be organized so that it gets on with other work; when the interrupt signal is received this other work is temporarily suspended while the character data is read.

Visual displays used for computer output have an array of data elements, one element for each pixel on the screen. If the screen is black and white, a single bit data element will suffice for each pixel. The pixels of colour screens require at least one byte of storage each. The memory used for the visual display may be part of the main memory of the computer, or may be a separate memory unit. Programs get information displayed on a screen by setting the appropriate data elements in the memory used by the visual display. Programs can access the data element for each pixel. Setting individual pixels is laborious. Usually, the operating system of the computer will help by providing routines that can be used to draw lines, colour in rectangles, and show images of letters and digits.

Visual display devices

There are many other input and output devices that can be attached to computers. Computers have numerous clock devices. Apart from the high

Clocks

frequency clocks that control the internal operations of the CPU and the bus, there will be clocks that record the time of day and, possibly, serve as a form of "alarm clock" timer. The time of day clock will tick at about 60-times per second; at each tick, a counter gets incremented. An alarm clock time can be told to send a signal when a particular amount of time has elapsed.

A-to-D converters

"Analog-to-Digital" (A-to-D) converters change external voltages ("analog" data) into bit patterns that represent numbers ("digital" data). A-to-Ds allow computers to work with all kinds of input. The input voltage can come from a photo-multiplier/detector system (allowing light intensities to be measured), or from a thermocouple (measurements of temperature), a pressure transducer, or anything else that can generate a voltage. This allows computers to monitor all kinds of external devices – everything from signals in the nerves of frog's leg to neutron fluxes in a nuclear reactor. Joystick control devices may incorporate simple forms of A-to-D converters. (Controllers for mice are simpler. Movement of a mouse pointer causes wheels to turn inside the mouse assembly. On each complete revolution, these wheels send a single voltage pulse to the mouse controller. This counts the pulses and stores the counts in its data registers; the CPU can read these data registers and find how far the mouse has moved in x and y directions since last checked.)

The controller for an A-to-D will be similar to that shown in Figure 1.15, except that the data register will have more bits. A one-byte data register can only represent numbers in the range 0 to 255; usually an accuracy of one part in 250 is insufficient. Larger data registers are used, e.g. 12-bits for measurements that need to be accurate to one part in four thousand, or 16-bits for an accuracy of one part in thirty thousand. On a 12-bit register, the value 0 would obviously encode a minimum (zero) input, while 4095 would represent the upper limit of the measured range. The external interface parts of the A-to-D will allow different measurement ranges to be set, e.g. 0 to 1 volt, 0 to 5 volt, -5 to 5 volt. An A-to-D unit will often have several inputs; instructions from the CPU will direct the controller to select a particular input for the next measurement.

D-to-A converters

A "Digital-to-Analog" (D-to-A) converter is the output device equivalent to an A-to-D input. A D-to-A has a data register that can be loaded with some binary number by the CPU. The D-to-A converts the number into a voltage. The voltage can then be used to control power to a motor. Using an A-to-D for input and a D-to-A for output, a computer program can do things like monitor temperatures in reactor vessels in a chemical plant and control the heaters so that the temperature remains within a required range.

Relays

Often, there is a need for a computer to monitor, or possibly control, simple two-state devices – door locks (open or locked), valves (open or shut), on/off control lights etc. There are various forms of input devices where the data register has each bit wired so that it indicates the state of one of the monitored devices. Similarly, in the corresponding output device, the CPU can load a data register with the on/off state for the controlled devices. A change of the setting of a bit in the control register causes actuators to open or close the corresponding physical device.

EXERCISES

1. A computer's CPU chip is limited in the number of circuits that it can contain. CPU designers have the equivalent of a million transistors from which to build their circuits. A CPU designer can implement circuits that perform many different kinds of data manipulation. Alternatively, a designer can choose to implement fewer distinct operations, instead using their circuits to duplicate elements (allowing two or more operations to proceed simultaneously) or so as to have more high speed registers. One approach gains speed by the program needing to execute relatively few instructions to achieve complex data manipulations; the other approach gains speed by having much faster, though simpler operations. Designers dispute which approach is best.

Research these two approaches – they are known by the acronyms CISC and RISC. Write a brief report summarizing the information that you were able to obtain.

The CPU used in the machine that you will be using for your course will be based on one of these design approaches. Which one?

2. Produce a table containing summary statistics on the machine that you will be using for your course. How much main memory? Does the video display use a separate memory, if so how much? How many colours can the video display use? What is the hard disk capacity? What I/O devices are attached? What is the CPU? ...
3. Charles Babbage, an English mathematician of the early nineteenth century, designed a number of calculating engines. The first, the "difference engine", was a special purpose calculator used to compute tables of functions (e.g. tables giving sines of angles); although Babbage never completed his model, a working version of the difference engine was marketed by a Swedish company in the mid nineteenth century.

Babbage's other machine was the "Analytical Engine". This was a general purpose programmable computing device – a real computer that was to be constructed entirely from cog wheels, gears, pistons and similar mechanical components. Some computer historians suggest that had it been built the Analytical Engine would have been a fully working computer analogous to modern machines, albeit very much slower.

Attempts at building the Analytical Engine failed; the technology available in the 1830s didn't permit construction of mechanical components with the precisions required.

Suppose it had been possible to build these machines. Write a "science fantasy" essay based on the assumption that the Babbage Computer Mark 1 entered commercial production in January 1840. (Possible ideas – a system manager's manual for steam driven computing devices, Charles Dicken's Mr. Scrooge as a database administrator, or a maybe a more serious essay exploring differences in how society might have evolved.)

2 Programs: Instructions in the Computer

Figure 2.1 illustrates the first few processing steps taken as a simple CPU executes a program. The CPU for this example is assumed to have a program counter (PC), an instruction register (IR), a single data register (R0), and a flags register (a few of whose constituent bits are shown individually). Instructions and data values each take up one "word"; addresses are the addresses of words (not of constituent bytes within words).

The program, located in memory starting in word 0, calculates $\sum_{n=1}^{n=3} n$.

<i>memory location</i>	<i>contents</i>	
0	LOAD	N
1	COMPARE	3
2	JUMP_IF_GREATER TO	9
3	ADD	SUM
4	STORE	SUM
5	LOAD	N
6	ADD	1
7	STORE	N
8	GOTO	0
9	STOP	
10	N	1
11	SUM	0

The program consists of a sequence of instructions occupying memory words 0–9; the data values are stored in the next two memory locations. The data locations are initialized before the program starts.

The calculation is done with a loop. The loop starts by loading the current value of N. (Because the imaginary CPU only has one register, instructions like the "load" don't need to specify which register!) Once loaded, the value from variable N is checked to see whether it exceeds the required limit (here, 3); the comparison instruction would set one of the "less than" (LT), "equal" (EQ), or "greater than" (GT) bits in the flags register.

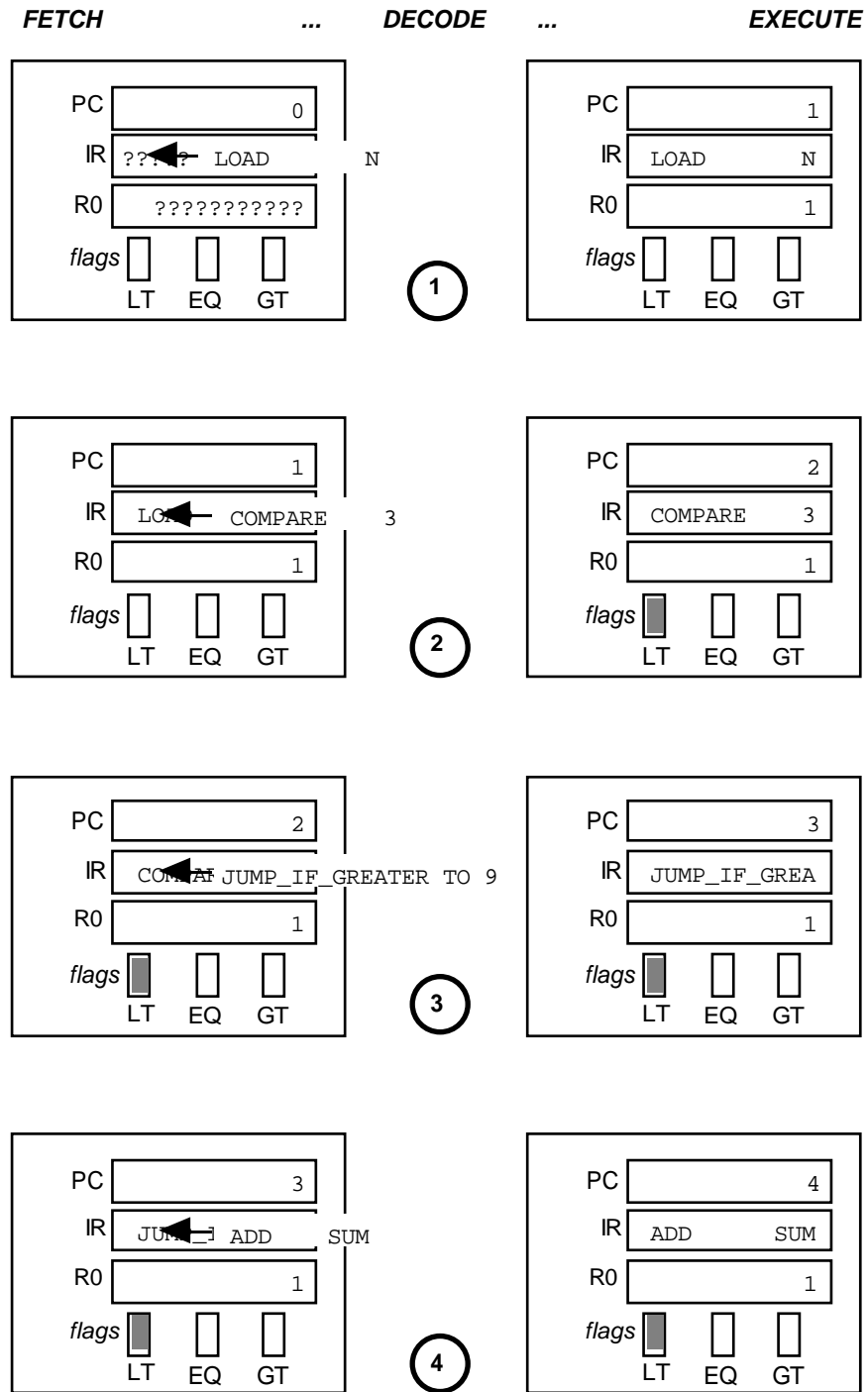


Figure 2.1 Executing a program.

The next instruction, the conditional jump in memory word 2, can cause a transfer to the STOP instruction following the loop and located in memory word 9. The jump will occur if the GT flag is set; otherwise the CPU would continue as normal with the next instructions in sequence. These instructions, in words 3 and 4, add the value of N into the accumulating total (held in SUM). The value of N is then incremented by 1 (instructions in words 5-7). Instruction 8 causes the CPU to reset the program counter so that it starts again at instruction 0.

The first few steps are shown in Figure 2.1. Step 1 illustrates the fetch and decode steps for the first instruction, loading value from N (i.e. its initial value of 1). As each step is completed, the PC is incremented to hold the address of the next instruction. Steps 2, 3, and 4 illustrate the next few instruction cycles.

If you could see inside a computer (or, at least, a simulator for a computer), this is the sort of process that you would observe.

Of course, if you could really see inside a computer as it ran, you wouldn't see "instructions" written out with names like COMPARE or LOAD, you wouldn't even see decimal numbers like 3 or 9. All that you would be able to see would be "bit-patterns" --- the sequences of '1's and '0's in the words of memory or the registers of the CPU. So it would be something a little more like illustration Figure 2.2.

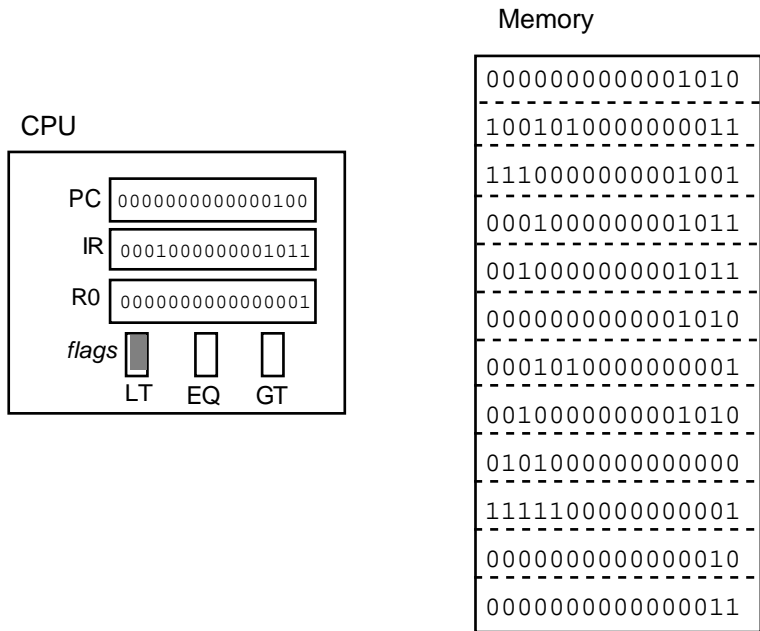


Figure 2.2 View into the machine.

In order to be executed by a computer, a program must end up as a sequence of instructions represented by appropriate bit patterns in the memory of the computer.

There is a very big gap between a statement of some problem that is to be solved by a computer program and the sequence of instruction bit patterns, and data

bit patterns, that must be placed in the computer's memory so that they can be processed by the CPU.

You will be given a problem statement ---

"Get the computer to draw a tyrannosaurus rex chasing some corythosaurus plant eating dinosaurs." (Jurassic Park movie)

"Program the computer to apply some rules to determine which bacterium caused this patient's meningitis." (Mycin diagnostic program)

"Write a program that monitor's the Voice of America newswire and tells me about any news items that will interest me."

and you have to compose an instruction sequence. Hard work, but that is programming.

2.1 PROGRAMMING WITH BITS!

On the very first computers, in the late 1940s, programmers did end up deciding exactly what bit patterns would have to be placed in each word in the memory of their computer!

The programs that were being written were not that complex. They typically involved something like evaluating some equation from physics (one of the first uses of computers was calculating range tables for guns). You may remember such formulae from your studies of physics at school --- for example there is a formula for calculating the speed of an object undergoing uniform acceleration

$v = \text{speed at time } t, \quad u = \text{initial speed},$
 $a = \text{acceleration}, \quad t = \text{time}$

$$v = u + a * t$$

(symbol * is used to indicate multiplication)

You can see that it wouldn't be too hard to compose a loop of instructions, like the loop illustrated at the start this chapter, that worked out v for a number of values of t .

The programmer would write out the instruction sequence in rough on paper ...

```
...
load          t
multiply      a
add           u
store         v
...
```

Then it would be necessary to chose which memory word was going to hold each instruction and each data element, noting these address in a table:

```
start of loop @ location 108
end of loop @ location 120
variable t @ 156
```

Given this information it was possible to compose the bit patterns needed for each instruction.

The early computers had relatively simple fixed layouts for their instruction words; the layout shown in Figure 2.3 would have been typical (though the number of operand bits would have varied).

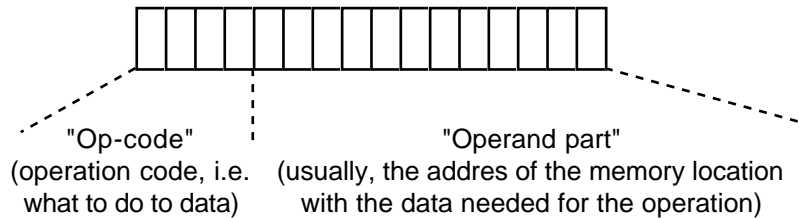


Figure 2.3 Simple layout for an instruction word

The programmer would have had a list of the bit patterns for each of the instructions that could be executed by the CPU

LOAD	0000	ADD	0001
STORE	0010	MULTIPLY	0011
SUBTRACT	0100	COMPARE	0101
...			

The "op-code" part of an instruction could easily be filled in by reference to this table. Working out the "operand" part was a bit more difficult --- the programmer had to convert the word numbers from the address table into binary values. Then, as shown in Figure 2.4, the complete instruction could be "assembled" by fitting together the bits for the opcode part and the bits for the address.

The instruction still had to be placed into the memory of the machine. This would have been done using a set of switches on the front of the computer. One set of switches would have been set to represent the address for the instruction (switch down for a 0 bit, switch up for a 1). A second set of switches would have been set up with the bit pattern just worked out for that instruction. Then a "load address" button would have been pressed.

*Loading on the
switches*

Every instruction in the program had to be worked out, and then loaded individually into memory, in this manner. As you can imagine, this approach to programming a computer was tedious and error prone.

By 1949, bigger computers with more memory were becoming available. These had as many as one thousand words of memory (!) for storing data and programs. Toggling in the bit patterns for a program with several hundred instructions was simply not feasible.

But the program preparation process started to become a little more sophisticated and a bit more automated. New kinds of programs were developed that helped the programmers in their task of composing programs to solve problems. These new software development aids were *loaders* and *symbolic assemblers*.

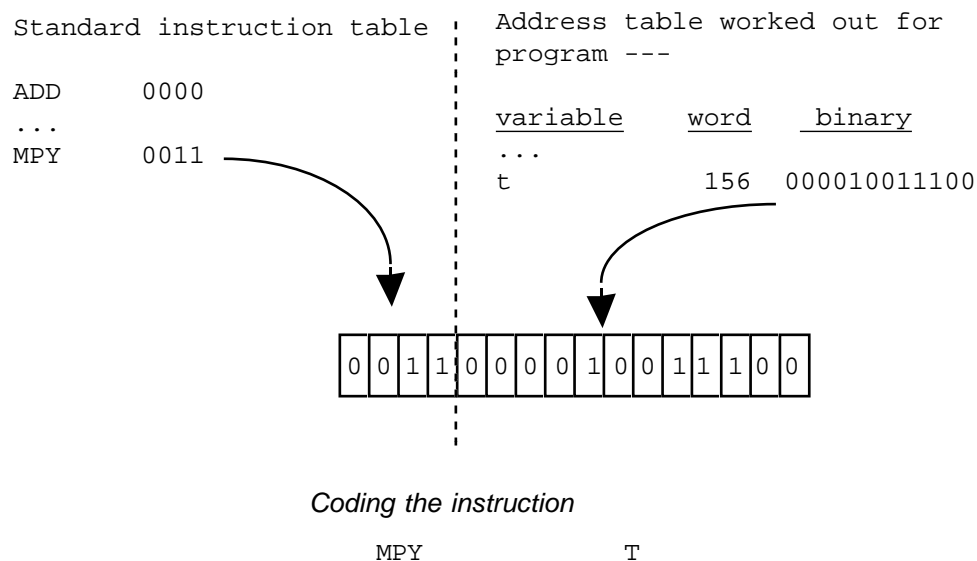


Figure 2.4 "Assembling" an instruction.

2.2 LOADERS

A "loader" is a fairly simple program. It reads in bit patterns representing the instructions (and initialized data elements) needed for a program, and stores these bit patterns in the correct locations in memory.

The bit patterns were originally represented on media like punched cards or paper tapes (a hole in the card at particular position might represent a 1 bit, no hole meant a 0 bit). With the bit patterns were punched on cards, a loader program could read these cards and store the bit patterns, starting at some fixed address and filling out successive words. The cards used typically had 12 rows where bits could be punched, and 80 columns. Depending on the word length of the computer, one instruction would be represented by two to four columns of a card; each card could hold the codes for twenty or more instructions.

The "loader" program itself was typically about 10 instructions in length. It would get toggled into memory using the switches on the front of the computer (using the last few locations in memory, or some other area not used by the program being loaded). The code of the loader would start by noting the address where the first instruction was to be placed. Then there would be a loop. In this loop, columns would be read from cards, instruction words built up from the bits read and, when complete would be stored in memory. As each instruction was stored, the loader would update its record of where to store the next. The loader would stop when it read some special end-marker bit pattern from a card. The person running the machine could then set the starting address for their program using the switches and set it running.

2.3 ASSEMBLERS

By 1950, programmers were using "assembler" programs to help create the bit pattern representation of the instructions.

The difficult creative aspect of programming is deciding the correct sequence of instructions to solve the problem. Conversion of the chosen instructions to bit patterns is an essentially mechanical process, one that can be automated without too much difficulty.

If an assembler was available, programmers could write out their programs using the mnemonic instruction names (LOAD, MULTIPLY, etc) and named data elements. Once the program had been drafted, it was punched on cards, one card for each instruction. This process produced the program source card deck, Figure 2.5.



Figure 2.5 Assembly language source deck

This "source" program was converted to binary instruction patterns by an assembler program (it "assembles" instructions). The source cards would be read by the assembler which would generate a binary card deck that could be loaded by the loader.

The assembler program performs a translation process – converting the mnemonic instruction names and variable names into the correct bits. Assembler programs are meant to be fairly small (particularly the early ones that had to fit into a memory with only a few hundred words). Consequently, the translation task must not be complex.

"Assembly languages" are designed to have simple regular structures so that translation is easy. Assembly languages are defined by a few rules that specify the different kinds of instruction and data element allowed. In the early days, further rules specified how an instruction should be laid out on a card so as to make it even easier for the translation code to find the different parts of an instruction.

Syntax rules Rules that specify the legal constructs in a language are that language's "syntax rules". For assembly languages, these rules are simple; for example, a particular assembly language might be defined by the following rules:

1. One statement per line (card).
2. A statement can be either:
 - An optional "label" at start and an instruction
 - or
 - a "label" (variable name) and a numeric value.
3. A "label" is a name that starts with a letter and has 6 or fewer letters and digits.
4. An instruction is either:
 - an input/output instruction
 - or
 - a data manipulation instruction
 - or
 - ...
 Instructions are specified using names from a standard table provided with the assembler.
5. Labels start in column 1 of a card, instructions in column 8, operand details in column 15.
6. ...

An assembler program uses a table with the names of all the instructions that could be executed by the CPU. The instruction names are shortened to mnemonic abbreviations (with 3 letters or less) ...

LOAD	---->	L
ADD	---->	A
STORE	---->	S
GOTO (JUMP)	---->	J
JUMP_IF_GREATER	---->	JGT
MULTIPLY	---->	MPY
STOP (HALT)	---->	STP
COMPARE	---->	C
...		

Short names of 1--3 characters require less storage space for this table (this was important in the early machines with their limited memories).

**"Two-pass"
translation process**

If an assembly language is sufficiently restricted in this way, it becomes relatively simple to translate from source statements to binary code. The assembler program (translator) reads the text of an assembly language program twice, first working out information that it will need and then generating the bit patterns for the instructions.

The first time the text is read, the assembler works out where to store each instruction and data element. This involves simply counting the cards (assembly language statements), and noting those where labels are defined, see Figure 2.6. The names of the labels and the number of the card where they occurred are stored in a table that the assembler builds up in memory.

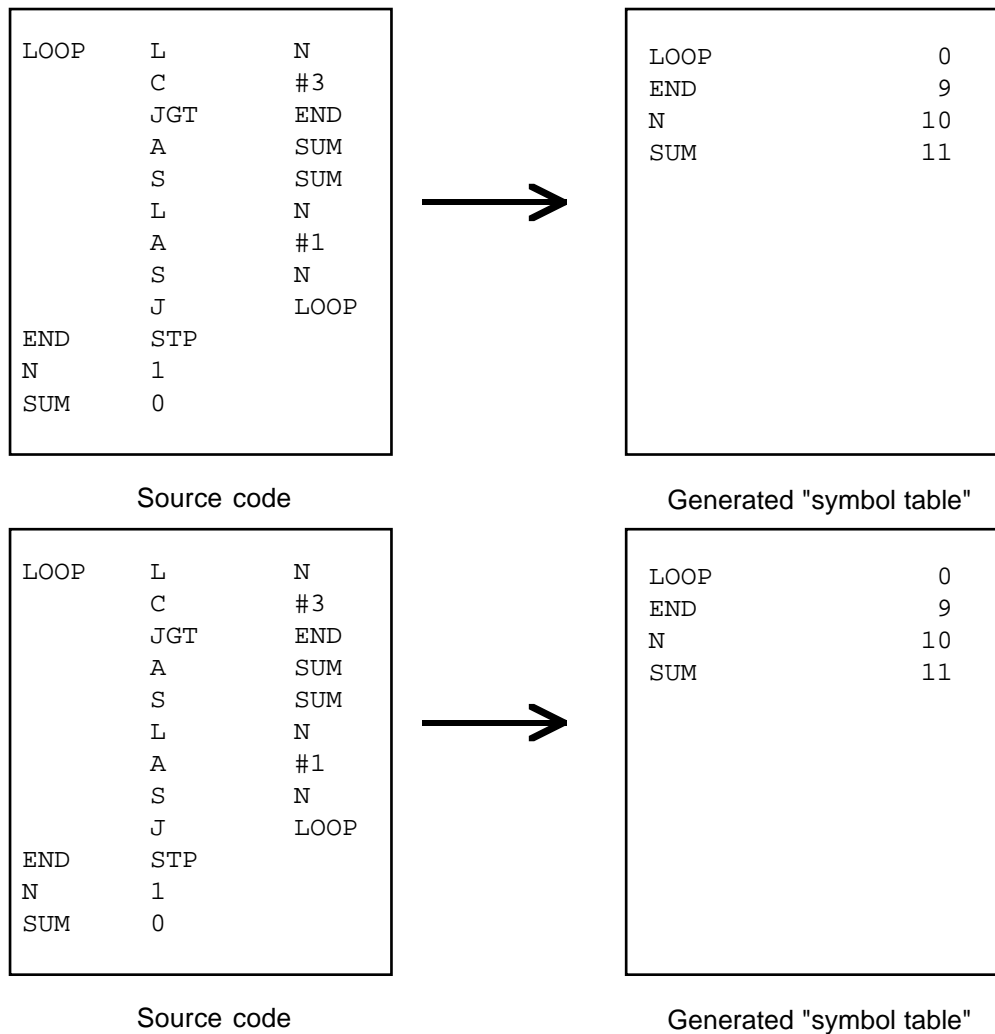


Figure 2.6 Generating a symbol table in "pass 1" of the assembly process.

The second time the source code is read, the assembler works out the bit patterns and saves them (by punching on cards – or in more modern systems by writing the information to a file).

All the translation work is done in this second pass, see Figure 2.7. The translation is largely a matter of table lookup. The assembler program finds the characters that make up an instruction name, e.g. JGT, and looks up the translation in the "instructions" table (1110). The bits for the translation are copied into the op-code part of the instruction word being assembled. If the operand part of the source instruction involves a named location, e.g. END, this name can be looked up in the generated symbol table. (Usually, the two tables would be combined.) Again, the translation as a bit pattern would be extracted from the table and these "address" bits would make up most of the rest of the instruction word.

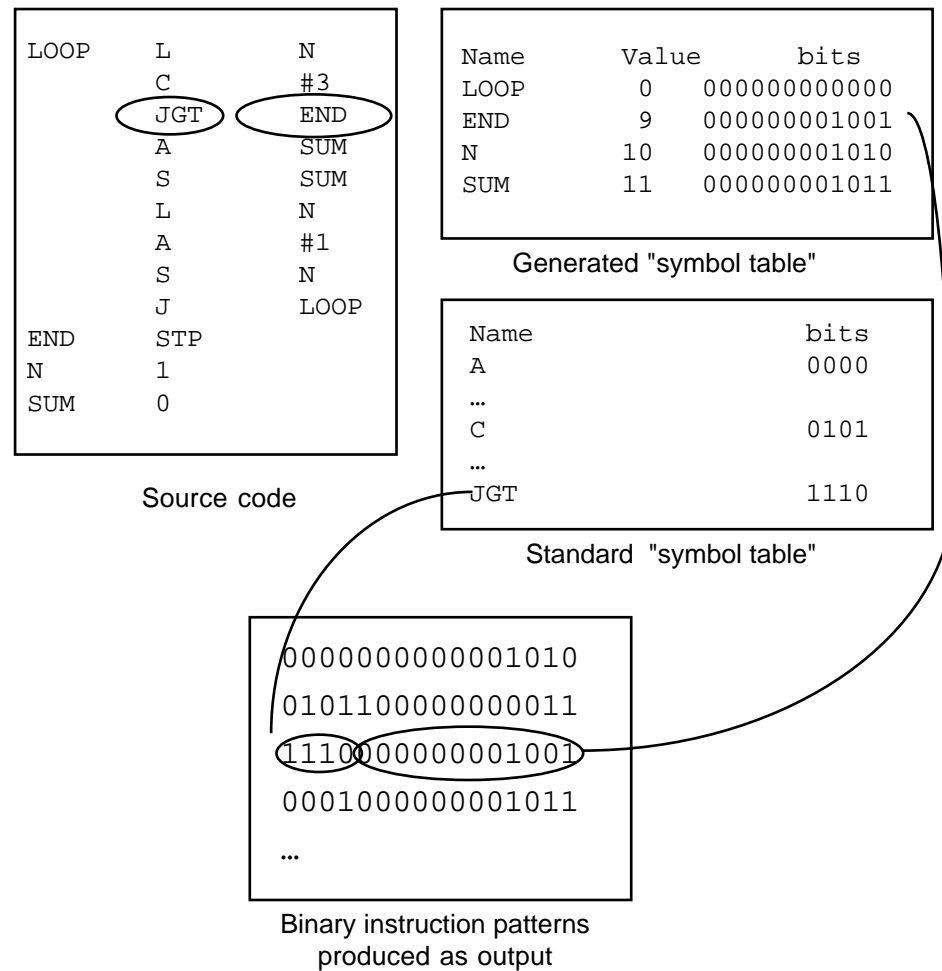


Figure 2.7 Generating code in "pass 2" of the assembly process

Apart from the op-code bits and address bits, the instruction word would have a couple of bits in the operand field for the "addressing mode". These bits would be used to distinguish cases where the rest of the operand bits held the address of the data (e.g. `A SUM`) from cases where the rest of the operand bits held the actual data (e.g. `C #3`).

2.4 CODING IN ASSEMBLY LANGUAGE

With assemblers to translate "assembly language" source code to bit patterns, and loaders to get the bits properly into memory, the programmers of the later '40s early '50s were able to forget about the bit patterns and focus more on problem solving and coding.

Many of the details of coding were specific to a machine. After all, each different kind of CPU has a slightly different set of circuits for interpreting instructions and so offers a different set of instructions for the programmer to use. But there were basic coding patterns that appeared in slightly different forms in all programs on all machines. Programmers learnt these patterns — and used them as sort of building blocks that helped them work out an appropriate structure for a complete program.

Coding— sequence

The simplest pattern is the sequence of instructions:

L	TIME
MPY	ACCEL
A	U
S	V

This kind of code is needed at points in a program where some "formula" has to be evaluated.

Coding – loops

Commonly, one finds places in a program where the same sequence of instructions has to be executed several times. A couple of different code patterns serve as standard "templates" for such loops:

*Do some calculations to determine whether can miss out
code sequence, these calculations set a "condition flag"
Conditional jump --- if appropriate flag is set, jump beyond loop*

	<i>Sequence of instructions forming the "body" of loop</i>
	<i>Jump back to calculations to determine if must execute "body" again</i>

Point to resume when have completed sufficient iterations of loop

This is the pattern that was used in the loop in the example at the beginning of this chapter. It is a "while loop" – the body of the loop must be executed again and again while some condition remains true (like the condition $N \leq 3$ in the first example).

An alternative pattern for a loop is

	<i>Start point for loop</i>
	<i>Sequence of instructions forming the "body" of loop</i>
	<i>Do some calculations to determine whether need to</i>

repeat code sequence again, these calculations set a "condition flag" Conditional jump – if the appropriate flag is set, jump back to the start of the loop	Point to resume when have completed sufficient iterations of loop.
--	--

This pattern is appropriate if you know that the instructions forming the body of the loop will always have to be executed at least once. It is a "repeat" loop; the body is repeated until some condition flag gets set to indicate that it has been done enough times.

Coding– choosing between alternative actions

Another very common requirement is to be able to chose between two (or more) alternatives. For example you might need to know the larger of two values that result from some other stage of some calculation:

```

if Velocity1 greater than Velocity2 then
    maxvelocity equals Velocity1
otherwise maxvelocity equals Velocity2

```

Patterns for such "selection code" have to be organized around the CPU's instructions for testing values and making conditional jumps.

A possible code pattern for selecting the larger of two values would be ...

```

load first value from memory into CPU register
compare with second value
jump if "less flag" is set to label L2
    first value is the larger so just save it
store in place to hold larger
goto label L3
L2 load second value
   store in place to hold larger
L3 ... start instruction sequence that uses larger value

```

Similar code patterns would have existed for many other kinds of conditional test.

Coding– "subroutines"

"Sequence of statements", "selections using conditional tests", and "loops" are the basic patterns for organizing a particular calculation step.

There is one other standard pattern that is very important for organizing a complete program – the *subroutine* .

Typically, there will be several places in a program where essentially the same operation is needed. For example, a program might need to read values for several different data elements. Now the data values would be numbers that would have to end up being represented as integers in binary notation, but the actual input would

be in the form of characters typed at a keyboard. It requires quite a lot of work to convert a character sequence, e.g. '1' '3' '4' (representing the number one hundred and thirty four), into the corresponding bit pattern (for 134 this is 0000000010000110). The code involves a double loop --- one loop working through successive digits, inside it there would be a "wait loop" used to read a character from the keyboard. The code for number input would take at least 15 to 20 instructions. It wouldn't have been very practical to repeat the code at each point where a data value was needed.

If the code were repeated everywhere it was needed, then all your memory would get filled up with the code to read your data and you wouldn't have any room for the rest of the program (see Figure 2.8 A). Subroutines make it possible to have just one copy of some code, like the number input code (Figure 2.8 B). At places where that code needs to be used, there is a "jump to subroutine" instruction. Executing this instruction causes a jump to the start of the code and, in some machine dependent way, keeps a record of where the main program should be resumed when the code is completed. At the end of the shared code, a "return from subroutine" instruction sets the program counter so as to resume the main program at the appropriate place.

"Jump to subroutine" and "return from subroutine" instructions

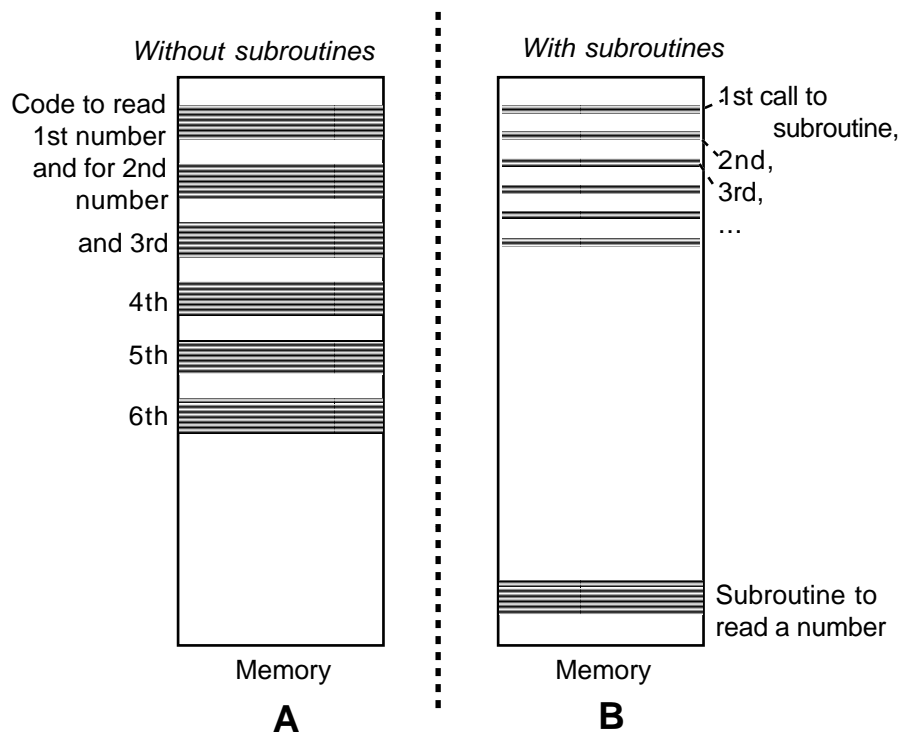


Figure 2.8 Sharing a piece of code as a subroutine.

The form of the code for the "main" program and for the subroutine would be something like the following:

Main program, starts by reading data values ...

START JSR INPUT
As the subroutine call instruction is decoded, the CPU saves the address of the next instruction so that the main program can be repeated when the subroutine finishes

```
S      ACCEL
JSR    INPUT
S      U
JSR    INPUT
S      TFINAL
```

Main program can continue with code that does calculations on input values

```
LOOP   L      T
        C      TFINAL
        JGT    END
        MPY    ACCEL
        A      U
        S      V
        ...
```

and

Subroutine, code would start by initializing result to zero, then would have loops getting input digits

```
"CLEAR"  RESULT
...
WAIT     "KEYBOARD_READY?"
JNE      WAIT
"KEYBOARD_READ"
...
...
```

When all digits read, subroutine would put number in CPU register and return (RTS = return from subroutine) ...

```
L      RESULT
RTS
```

Different computers have used quite distinct schemes for "*saving the return address*" — but in all other respects they have all implemented essentially the same subroutine call and return scheme.

Coding— "subroutine libraries"

Subroutines first appeared about 1948 or 1949. It was soon realized that they had benefits quite apart from saving memory space.

Similar subroutines tended to be needed in many different programs. Most programs had to read and print numbers; many needed to calculate values for functions like sine(), cosine(), exponential(), log() etc. Subroutines for numeric input and output, and for evaluating trigonometric functions could be written just once and "put in a library". A programmer needing one of these functions could simply copy a subroutine card deck borrowed from this library. The first book on

programming, published in 1950, explored the idea of subroutines and included several examples.

Coding – "subroutines for program design"

It was also realized that subroutines provided a way of thinking about a programming problem and breaking it down into manageable chunks. Instead of a very long "main line program" with lots of loops and conditional tests, programs could have relatively simple main programs that were comprised mainly of calls to subroutines.

Each subroutine would involve some sequential statements, loops, conditionals – and, possibly, calls to yet other subroutines.

If a code segment is long and complex, with many nested loops and crisscrossing jump (goto) instructions, then it becomes difficult to understand and the chances of coding errors are increased. Such difficulties could be reduced by breaking a problem down through the use of lots of subroutines each having a relatively simple structure.

Coding– data

Assembly languages do not provide much support for programmers when it comes to data. An assembly language will allow the programmer to attach a name to a word, or a group of memory words. The programmer has to choose how many memory words will be needed to represent a data value (one for an integer, two/four or more for a real number, or with text some arbitrary number of words depending on the number of characters in the text "string".) The programmer can not usually indicate that a memory word (or group of words) is to store integer data or real data, and certainly the assembler program won't check how the data values are used.

In the early days of programming, programmer's distinguished two kinds of data:

Globals

The programmers would arrange a block of memory to hold those variables that represented the main data used by the program and which could be accessed by the main program or any of the subroutines.

Locals

Along with the code for each subroutine, the programmer would have allocated any variables needed to hold temporary results etc. These were only supposed to be used in that subroutine and so were "local" to the routine.

The organization of memory in an early computer is shown in Figure 2.9. The "local" data space for each subroutine was often located immediately after the code, so mixing together code and data parts of a program. Many machines were designed around the concept of "global" and "local" data. These machine

architectures reserved specific memory addresses for global data and provided special "addressing modes" for accessing global and local data elements. Different memory organizations were proposed in the early 1960s in association with the development of the "Algol" languages. These alternative ways of arranging memory are now more popular; they are introduced in section 4.8.

Memory

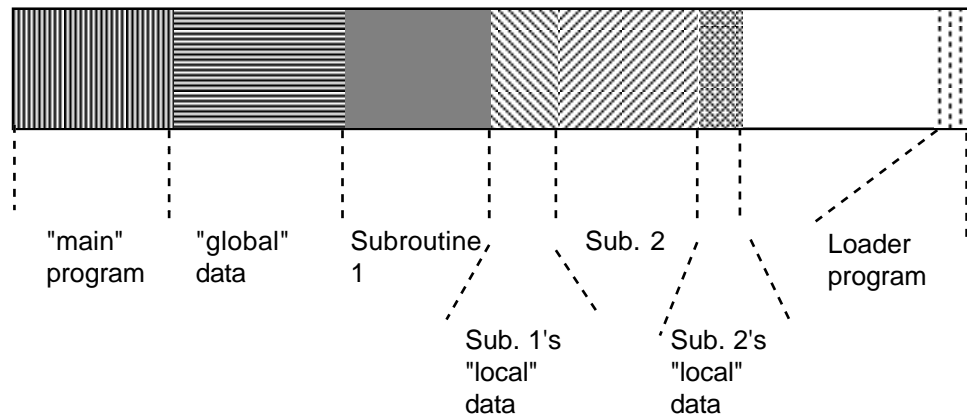


Figure 2.9 Organization of a program in the memory of an early computer.

2.5 FROM ASSEMBLY LANGUAGE TO "HIGH LEVEL" LANGUAGES

In the 1950s, most programs were written in assembly language.

But assembly languages have disadvantages – they involve the programmer in too much detail. The programmer is stuck with thinking about "instructions in the computer". Now sometimes one does want to choose the instruction sequence; people still program in assembly language when dealing with small sections of time critical code. But for most programming tasks, one doesn't really want to get down to the level of detail of choosing individual instructions.

Starting in the mid-50s, "higher level programming languages" began to appear. Initially they did little more than provided packaged versions of those coding patterns that assembly language programmers were already familiar with. Thus a "high level programming language" might provide some kind of "DO- loop" ...

```
DO 10      IT = 1, 4
...
10 CONTINUE
```

Here, the programmer has specified that a sequence of statements (. . .) is to be executed 4 times. It is fairly easy to translate such a high level specification of a loop into assembly language code with all the instructions for comparisons, conditional tests, jumps etc.

When programming in a high level language, the programmer no longer has to bother about individual instructions. Instead, the programmer works at a higher level of abstraction – more remote from machine details. Automatic translation systems expand the high level code into the instruction sequences that must be placed in the computer's memory.

Over time, high level languages have evolved to take on more responsibilities and to do more for the programmer. But in most high level languages, you can still see the basic patterns of "instruction sequence", "loop", "conditional tests for selection", and "subroutines" that have been inherited from earlier assembly language style programming.

3 Operating Systems

3.1 ORIGINS OF "OPERATING SYSTEMS"

Back in the early 1950s there were very few computers. Large universities might have one, as might government research laboratories, and a few big companies. A scientist or engineer wanting to run a program would book time on the machine. When their turn came, they would be totally in control of "THE COMPUTER".

They had to load the assembler (or high level language translator), use it to convert their program to binary instruction format, load the binary version of their program, start their program using the switches on the front of the computer, feed data cards to their program as it ran, etc. It all involved a lot of running around fiddling with card readers (and, later on, magnetic tapes), pressing buttons, flicking switches. Most people didn't do the job very well and wasted a lot of the time that they had booked on the machine.

Organizations that had computers soon started to employ professional "computer operators". Computer operators were responsible for getting maximum use out of the computing equipment. They would run the language translators and assemblers for users, they would load binary card decks and get the programs run. Since they worked full time with the computers, they were familiar with all the operating procedures involved and so made many fewer mistakes and wasted much less time.

Computer operators

Since users no longer directly controlled the running of their programs, they had to write "job control instructions" for the operators. These job control instructions would tell the operators of any special requirements such as the need to mount a particular magnetic tape on which a program was to store some generated data.

Computers were changing rapidly in the early '50s. Peripheral devices were becoming more complex and versatile. Many could operate semi-autonomously. They could accept a direction from the CPU telling them to transfer some data and could organize the data transfers so that these occurred while the CPU continued executing a program. Such changes made for more efficient program execution because the CPU spent less time waiting for data transfers. But such changes also meant that the code to control peripheral devices became a lot more complex. Any erroneous I/O code could seriously disrupt the workings of the computer system and might even result in damage to peripheral devices or to media like magnetic tapes or punched cards.

*"System's code" for
i/o*

Originally, a programmer would have been able to use all of the computer's memory (if possible, the area with the loader program was left alone so that the loader code wouldn't have to be toggled into memory again). The requirements of the new peripheral devices made it advantageous to reserve a part of memory for "systems code" – the specialized code that controlled the new devices. The new organization for main memory is shown in Figure 3.1; part of memory is reserved for the "system's code" (and, also, for system's data – the system needed memory space to record information such as identifiers associated with magnetic tapes on different tape decks).

Memory

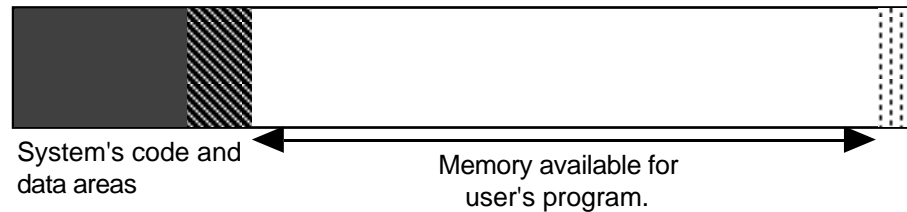


Figure 3.1 Part of main memory is reserved for "system's" code and data.

The "systems" area of memory had subroutines for whole variety of input output tasks. There would have been a subroutines to handle tasks such as reading one character from the control keyboard or one column of a punched card. Another routine would exist to organize the transfer of the contents of a number of words of memory (containing binary data) to a tape block. Programmers could have calls to these "systems subroutines" in the code of their programs. (Later, if you study computer architecture and operating systems you will learn why a specialized variant on the JSR subroutine call instruction was added to the CPU's instruction set. This specialized instruction would have had a distinct name, e.g. "trap" or "supervisor call (svc)". Calls to the systems subroutines were made using this "system call" instruction rather than the normal JSR jump to subroutine.)

Usually, rather than making direct calls to the "system's code" programmers continued to make use of library routines. For example, a program might use a number input routine. This routine would read characters punched on a card (making a call to one of the systems' input routines to get each character in turn) and would convert the character sequence to a numeric value represented as a bit pattern. There would be two such input routines (one for integers, the other for real numbers). There would be corresponding output routines that could generate character sequences that could be printed on a line printer.

These early systems established a kind of three component structure for a running program. The structure, shown in Figure 3.2, is the same as on modern computer systems. The three kinds of code are: the code written for the specific program, library code (provided by colleagues, commercial companies, or possibly the computer manufacturer), and "system's code" (usually provided by the computer manufacturer, though sometimes purchased from other commercial companies).

Memory

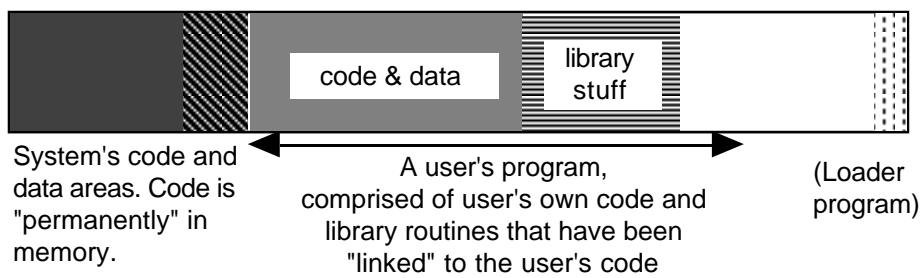


Figure 3.2 System's code, library code, and user's code.

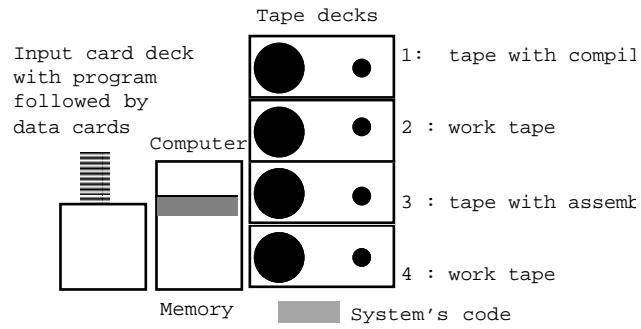
By the late 1950s, memories were getting larger. Many machines had as many as 4000 words of memory (equivalent to about sixteen thousand to twenty thousand bytes in modern terms). Some machines had as much as 32000 words of memory. These larger memories allowed the systems code to be extended in other ways that could improve the utilization and performance of the computer system.

Quite substantial amounts of time tended to be wasted between jobs and between each phase of a job. For example a user might submit a job written in the high level language Fortran along with a deck of data cards to be processed. The operator would have to load the Fortran compiler (translator) and feed in the cards with the program; the translator would output assembly language code that would be punched on cards or written to tape. When the compiler stopped, the operator had to load the assembler program and feed in the generated assembly language source, along with source card decks for any assembly language library routines the user had requested. The assembler would produce binary code (on cards or tape). The operator then had to load the binary and start the program running. Once the program was running, the operator had to feed in the data cards. When the program stopped, the operator had to collect up cards and printed output and then reset the machine ready for the next job. Each of these steps would involve the operator attending card readers, punches, tape decks, and control consoles.

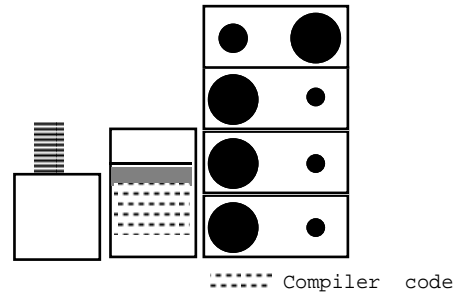
System's code to help the operators

If a computer system only had card input and output, the entire process was very slow. It could take several minutes to load a Fortran translator, run a user program through the translator and punch a deck of cards with the assembly language translation. Most large systems would have had several tape decks which made things a little easier. But it was still an involved process; the operator would have to work at the control switches on the front of the computer, entering instructions that would cause the computer to load the translator or assembler program from tape, then setting the switches again before starting the loaded program.

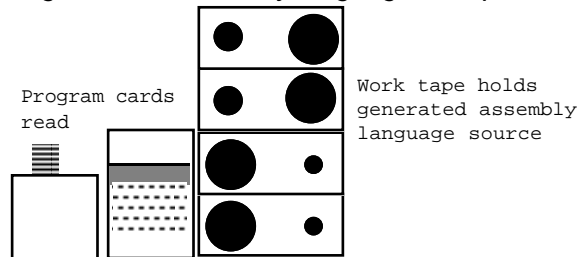
The process is illustrated in the various panes in Figure 3.3. The computer system would have had four or more tape decks; one would have the tape with the code for the compiler, the other would have the assembler. The other two tape decks would hold work tapes where intermediate results could be written. The operator would start by loading the compiler into memory and putting the card deck with program and data on the card reader.



Operator gets computer to load compiler



Operator runs compiler, it reads program and generates assembly language on tape



Operator loads assembler and rewinds tape with generated assembly language

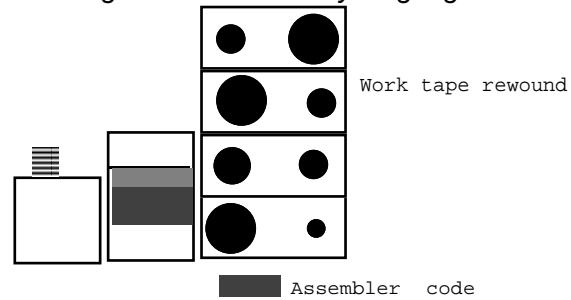


Figure 3.3 Operating the computer.

When the compiler had been loaded, the operator would use the control switches to start it running. The compiler would read the cards with the source code and generate assembly language that would be written to tape. The operator then had to rewind this tape, and load the assembler. The assembler then had to be run; it would read its input from the first work tape and produce the binary code (with the bit patterns for instructions) on a second work tape. When the assembler finished, the operator again had to rewind a tape and then again get a program (now the user's program) loaded into memory. Finally, the user's program could be started. It would read its data cards and produce output on a line printer.

Every time the operator had to do something like enter data on switches or rewind a tape, the CPU would be left idle. Since operators wouldn't know how long it would take to translate a program, or assemble it, or run it, they couldn't be instantly ready to start the next step in the process. So time was inevitably wasted. The CPUs were desperately expensive (CPUs with a fraction of the power of a modern personal computer would have costed millions of dollars) so it was very important to try to keep them working as much as possible.

Most of the operators' tasks involved simple, routine things that could be automated and speeded up. Control routines were written to automate the routine tasks like loading the compiler and rewinding a tape. This control code was added to the systems' i/o code in the reserved area of memory.

These control programs, written to automate routine tasks for the computer operators, were the first "Operating Systems". They worked by reading "control cards" and performing standard actions.

Programs were submitted along with "job control cards" that could be interpreted by the computer's control program. An input card deck (circa ≈1960) would have been something like the following:

*Job control
"languages"*

```

JOB          USERNAME
FORTRAN      ...
C    MY PROGRAM TO COMPUTE HEAT FLOW IN REACTOR
      DIMENSION A(10,10)
      ...
      END
ASSEMBLE     ...
LINK         ...
RUN
      1.2      4.5
ENDJOB

```

The control program used the control cards like the "JOB" card. These would have a keyword (and usually additional information, the JOB card might have identified the user who was to be charged). A job would be terminated by an ENDJOB card; if anything went wrong (like a coding error detected by the compiler or an arithmetic error during execution) the operating system could tidy up by reading all remaining cards in the card reader until it found the ENDJOB.

The operating system code handled each control card in a standard way. When it read a JOB card, it would print an accounting message on the operator's terminal. A FORTRAN card was processed by loading the FORTRAN compiler into memory and then "calling" it to read all the FORTRAN source cards. The operating system

would also arrange for an output tape to be ready for the compiler to use. If there were no compile errors, the operating system organized the rewinding of the output tape and the loading of the assembler. The other steps, like the linking process where "library" code was read from tapes were handled in similar fashion. If the process resulted in successful construction of an executable program, this was then "RUN" so that it could process its data cards. The FORTRAN compiler, the ASSEMBLER, the LINKER, and the user program were all executed rather as if they were subroutines called by the controlling operating system.

3.2 DEVELOPMENT OF OPERATING SYSTEMS

By the early 1960s, the larger computer systems had the form shown in Figure 3.4. Disks were not common, most systems made do with tape decks. An early disk unit would have been about five feet high, would have had a multi-platter arrangement with disks about 18 inches across. A complete disk unit would have held about as much data as a modern floppy and would have cost – well certainly more than a new car, probably something close to the price of a small suburban house.

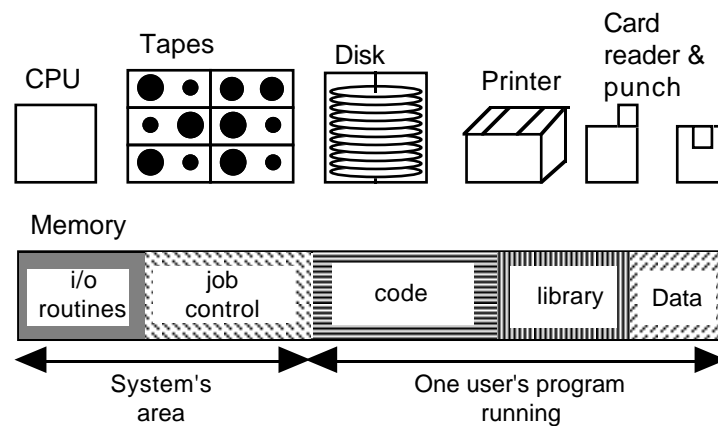


Figure 3.4 "Mainframe" computer system of early 1960s.

Most of the computers of the time got by with just tapes, but some had disks. Disks were a lot faster than tapes (faster transferring data, no need to read all the data sequentially because the disk control could move the disk heads to the blocks containing the specific data required, no need to "rewind", ...). A disk unit could substitute for several tapes. Space on the disk would be partitioned: e.g. blocks 0-159 holding the FORTRAN compiler, 160-235 holding the COBOL compiler, 236-263 with the RPG compiler, 264-274 for the ASSEMBLER, 275-304 containing the FORTRAN subroutine library, 305-310 for the LINKER, 311-410 for "workspace 1", 411-511 for "workspace 2". But even if a disk was being used, the operations remained much the same as described for the tape-based machines.

Part of the machine's main memory would be reserved for the primitive Operating System. This would have consisted of a set of standard i/o subroutines that controlled the various peripheral devices and the subroutines that interpreted the job control cards. In addition to this Operating System (OS), the computer would have a number of pieces of software supplied by its manufacturer:

- FORTRAN compiler (translator for a programming language intended for use by scientists and engineers)
- FORTRAN subroutine libraries (by now these would contain subroutines for doing things like sorting, complex mathematical operations like 'matrix inversion', as well as more standard things like converting digits to numeric values or calculating SINE(), LOG() etc)
- COBOL compiler (translator for a business programming language)

and others.

The job control component of the OS minimized the time needed by the operators to set up the machine between successive jobs, and the time spent organising successive phases of an individual job. The system's i/o routines could arrange for i/o to proceed simultaneously with computations. Together these improved the efficiency of machine usage. But there were problems. For example:

- CPU usage was still relatively low. Even with smart peripherals running simultaneously with computations, most programs spent a large amount of time (often 50% or more) in "wait loops" waiting for input data to be read from devices, or waiting for output data to be printed.
- Programmers were inconvenienced by slow turn around and lack of opportunity to test and debug their programs. (It was common to have to wait 24 hours for results to be returned, and the 'results' might be simply a statement to the effect that the compiler had found the second card of the program to be wrongly punched!)
- Some programs needed more memory than could be obtained.

These problems, and other related problems, were dealt with in a series of developments undertaken throughout the 1960s and early 1970s that lead to more sophisticated operating systems.

3.2.1 Multiprogramming

The most pressing problem was the "waste" of CPU cycles when the single job in the computer's memory had to wait for an i/o transfer to complete. The CPU was by far the most costly part of the computer system of this period. The prices of other components like the peripherals and memory were falling. Larger memories were practical. The availability of larger main memories provided the solution to the problem of keeping the CPU busy. If a single job in main memory couldn't

keep the CPU occupied all the time, then have two or more jobs in memory (Figure 3.5).

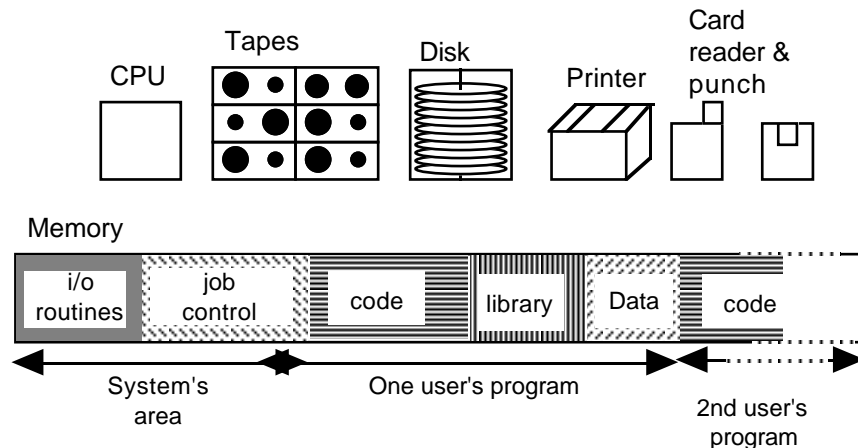


Figure 3.5 Early multiprogrammed system.

The computer's memory was be split up into "partitions".

- One held the OS.
- Another (tiny) part would hold the 'boot' loader (the name 'boot' loader apparently relates to some argument that starting the computer using this loader to read the OS from tape/disk was like 'picking oneself up by one's bootstraps'. ? !)
- The rest of memory was split up among two, three or more program partitions.

A user's program would be run in a partition in much the same way as it had previously been run using all memory not allocated to the OS.

In the early versions of this scheme, only the CPU and the OS were shared. Each memory "partition" had to have its own assigned peripheral devices – a line printer, a card reader, two or more tape decks, and a separate allocation of space on disk (if there was a disk). The OS had to be more elaborate because it had to keep track of each of the different jobs and of the way peripheral devices had been allocated to jobs. If a job asked the OS to perform some i/o operation that involved delays, the OS had to switch the CPU to execute instructions for a job in a different memory partition. Really, the more elaborate OS was using a computer with a large memory to simulate the working of two (or more) smaller, slower computers!

The "partitioning" of memory was adjustable. The operators could enter commands at the control terminal that would get the OS to rearrange the way memory was used. For example, in the morning the operators might arrange to have two medium sized partitions to run average jobs. In the afternoon, when lots of programmers had submitted programs that needed small test runs, the operators could reconfigure the machine to have three small partitions. Overnight, the

machine would be used with all the non-OS memory in a single partition so as to allow bigger programs to run.

The human operators were involved in the relatively slow, and infrequent, process of reconfiguring the machine for a different degree of "multiprogramming". The process of switching the CPU among the different programs was all done automatically, in mere millisecond times, under the control of the code in the OS. Switching would normally happen only when a running job made a request for i/o to the OS; a request like "card reader, read a card and store all 80 characters in this portion of memory ...", or "line printer, this portion of memory holds the 132 characters to be printed on a line, please print them", or "tape drive 8, transfer your next data block into memory starting here ...". The OS would get the device to start the data transfer, and arrange to be notified when the transfer was complete. But the requesting program usually wouldn't be able to continue until the transfer was done – so the OS switched the CPU to work for another program.

Essentially all operating systems now use some form of multiprogramming system. Support for multiprogramming may be limited in an OS for a personal computer (after all a personal computer is to be used by just one person and that person can probably only concentrate on one thing at a time); a personal computer may have many different programs concurrently in memory but the CPU may only get switched when the user selects the window associated with a different program. Larger machines use elaborate multiprogramming schemes. Human intervention is no longer needed to change the degree of multiprogramming. The job control portions of the OS now include code for elaborate algorithms that work out how many jobs should be allowed to run to get best performance. Rather than fixed memory partitions, these modern multiprogramming systems can reorganize memory whenever this is found to be necessary.

3.2.2 Timesharing

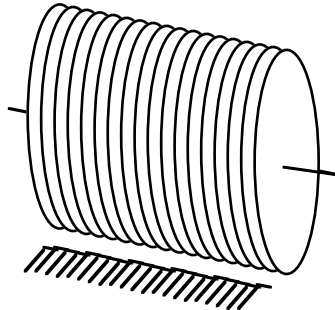
Multiprogrammed systems improved the usage of the CPU – but didn't do that much for the programmers.

Programmers were lucky to get one test run of their programs each day. While this may have encouraged a meticulously careful style of coding it really didn't do much for the productivity of the average programmer. Programmers needed a system where they could edit their code making additions and corrections, give the new code to a language translator (compiler) and either rapidly get any coding errors identified or get a program that could be run. If a program was successfully compiled, programmers needed to make small test runs that allowed their programs to be tested with small data sets.

"Timesharing" systems, pioneered at MIT in Boston USA starting around 1960, provided programmers with more appropriate development environments. It was the development of disk (and drum) storage that made these new systems possible. (Drums have ceased to be used. They are a variation on disks in which data is recorded in tracks on a cylinder rather than on a flat platter. Each track had its own read-write head, see Figure 3.6. Drums were a lot faster than disks, but the

multiple read-write heads made them expensive. They have gone out of fashion because their maximum storage capacity is a lot less than a disk.)

Metal cylinder, surface covered by
magnetic oxide, tracks on surface of
cylinder



A read/write head for
each track

Figure 3.6 Drum storage.

The CPU of MIT's original time-sharing computer was a slightly modified version of the largest computer then commercially available – an IBM 7094. The machine had two blocks of memory, both with 32K (i.e. 32768) words. One block of memory held the operating system. The other held the program and data for a single user. The computer had several very large disks with a total capacity of several million words of storage (≈ 50 Megabyte in modern terms?).

The computer was intended to support up to 32 people working simultaneously. Users worked at 'teletypes' (a kind of electric typewriter) that were connected to the computer via a special controller (see Figure 3.7).

Only one user's program was actually running at any time. It was the program that was currently in memory. The other 'users' would be waiting. Their programs would be saved on the drum.

A user's program was allowed to run until either it asked the OS to arrange some input or output (to the user's teletype or to a disk file); or until a 'time quantum' expired. (A time quantum might be ≈ 0.1 seconds.) When a program asked for i/o or used up its time quantum it would be '*swapped out to the drum*'. 'Swapping out' meant that the content of user memory was written to the drum. A different user's program would then be 'swapped in'. This program would then be allowed to run until it exceed its time quantum or asked for i/o. The second program would then get swapped out and a third program would be swapped in. Eventually, it would be program one's turn to run again.

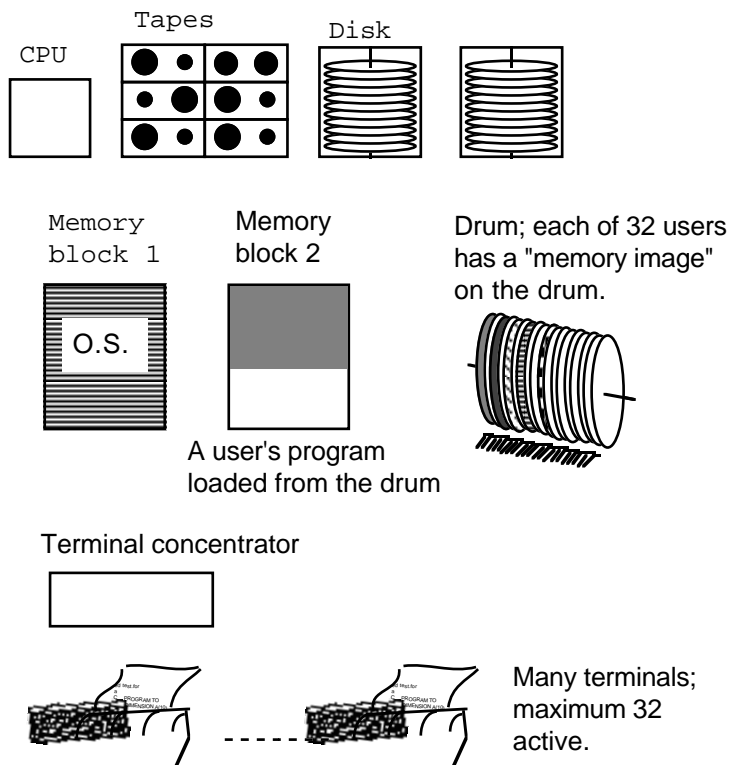


Figure 3.7 MIT's pioneering "time-sharing" computer system.

Even if the system was very busy with lots of users, each individual user would get ≈ 0.1 seconds worth of CPU time every three seconds. So it would seem to an individual user that their program was running continuously, albeit on a rather slow computer.

Operating systems for personal computers don't use timesharing. (They are personal computers because you don't want to share!) But most other operating systems (e.g. the Unix system you may use in more advanced CS subjects) incorporate sophisticated versions of the timesharing scheme pioneered by MIT.

Modern computer systems have much more memory and so don't have to do quite as much "swapping"; but swapping is still necessary. (Swaps are between memory and disk as the machines won't have drums; but quite often there will be a special small high speed disk that is used only for swapping and not to hold files.) If you get to use Unix, you will notice when you've been "swapped out"! (The machine seems to ignore you.)

There are obvious similarities in "multiprogramming" and "timesharing"; both involve the OS controlling the use of a computer system so that it appears as several computers (lots of little ones, or lots of slow ones). Multiprogramming and timesharing evolved to meet different needs; multiprogramming was to optimize CPU usage, timesharing was an attempt to optimize programmer usage! The

switching of the CPU and swapping of programs after a time quantum is quite wasteful; during the switch the CPU isn't doing any useful work. In the '60s when CPU power was at a premium, timesharing systems were not common.

Modern OSs combine timesharing with "background batch streams" (successor to multiprogramming) to get the best of both approaches. If no interactive user needs the CPU, the OS allocates it to a long running, multi-programmed "background job".

3.2.3 File Management

Users of the MIT system were able to keep their own files on disk. Operating systems had to be extended quite a lot before they could allow users to have their own files like this.

The first disks were limited in capacity and all their storage capacity was needed for systems software – compilers, assemblers, subroutine libraries, work areas for the compilers etc. As suggested in an earlier example, the system's administrator could allocate the space; e.g. "FORTRAN compiler in blocks 0..159" etc. This was adequate when there wasn't much disk space and whatever there was had all to be used for systems software. As disks got larger, it became possible to allow users to have some space. But this space couldn't be allocated to users in the same way as the system's administrator had allocated it for systems software. It was no good telling a user "You can have blocks 751 to 760" because they'd make mistakes and use blocks 151 to 160 (and so overwrite the compiler!).

Every attempt by a user program to read from or to write to the disk has to be checked – another task for the OS. The OS has to keep records of the disk space used by each user. The simple form of "file directory" (illustrated in Figure 1.14) has to be made more elaborate to suit a shared environment. In shared environments, the disk directories contain details of file ownership and access permissions as well as the standard information like file name and block allocation. The time-shared computer systems started to have files owned by the operating system with lists of the names of users; the main directory on the disk would have an individual directory for each user. These individual directories contained details of the files belonging to that user. The OS could impose restrictions on the number of files, or total space allocated to an individual user. Users could specify that they wanted files to be private, or to be readable by other users in a cooperative work group.

Once the OS had acquired records of users, it could keep records of jobs run, CPU time consumed and so forth. Previously, such accounting information had had to be manually logged by the computer operators. By the mid to late 1960s, these accounting tasks had become another responsibility of the OS; the operators merely had to run a program that printed out the bills that had been calculated using the information that the OS had stored.

3.2.4 Virtual Memory

In the late '50s and early '60s, computer memories were still quite small; it was rare for a computer to have more than 32K (32768) words of main memory (in modern terms, that is about 120,000 to 160,000 bytes). Some of this memory had to be reserved for the OS. This meant that programmers were limited in the amount of code and data that they could get in memory.

Once disks became available, schemes were devised that allowed programmers to have programs and data that exceeded the machine's main memory capacity. When the programming was running, only parts of its code and data would be in main memory, the rest would be on disk. Initially, individual programmers were responsible for organizing their program code and data so that both could be stored partly in memory and partly on disk. As shown in Figure 3.8, the programmer had to break up a program into groups of subroutines that got used in the same "phases" of a program.

"Overlays"

A program would typically start by using one group of routines (some taken from a library) to read in data, then another group of routines would do calculations needed to set up for the main processing phase, a different group of routines would do the main calculations, another calculation phase would follow to get results that were to be printed, and then some routines would be used in an output phase. At any one time, only a few of the routines needed to be in main memory.

The program would have some fixed amount of space for code in main memory. The complete program was stored on disk (spread over many disk blocks). When the program was run, a small part of this code would have to be in main memory. This chunk of code would have included the main program, and any frequently used subroutines. The main program had to have extra code to load the different 'overlays' as each was needed. It would start by loading the 'overlay' with the input routines (Figure 3.8A). Once the main program had loaded the input 'overlay' it would call the input routines; these would read the program's data. When the input phase was complete, the main program would resume and arrange the loading of code for phase 1 of the calculations. The phase 1 routines would then be called (Figure 3.8B). The main program would continue by loading the overlay with the routines for the next phase. The phase 2 routines would then be executed (Figure 3.8C). The program would continue in this way, loading each bit of code as it was needed.

As well as bothering about shuffling the code between disk and main memory, the programmers had to devise equally complex schemes to shuffle their data. It was very common to have far more data than would fit in main memory. For example, an engineer who wanted to model heat flow in a metal beam might need to work out the temperature at successive time intervals at a number of points on the beam. To get meaningful results, the engineer would have needed a fairly fine grid of points, e.g. modelling 1000 points along the beam and 100 points across the beam. But you can't fit 100,000 real numbers into main memory if you've only got 32K words for the OS, your program and your data.

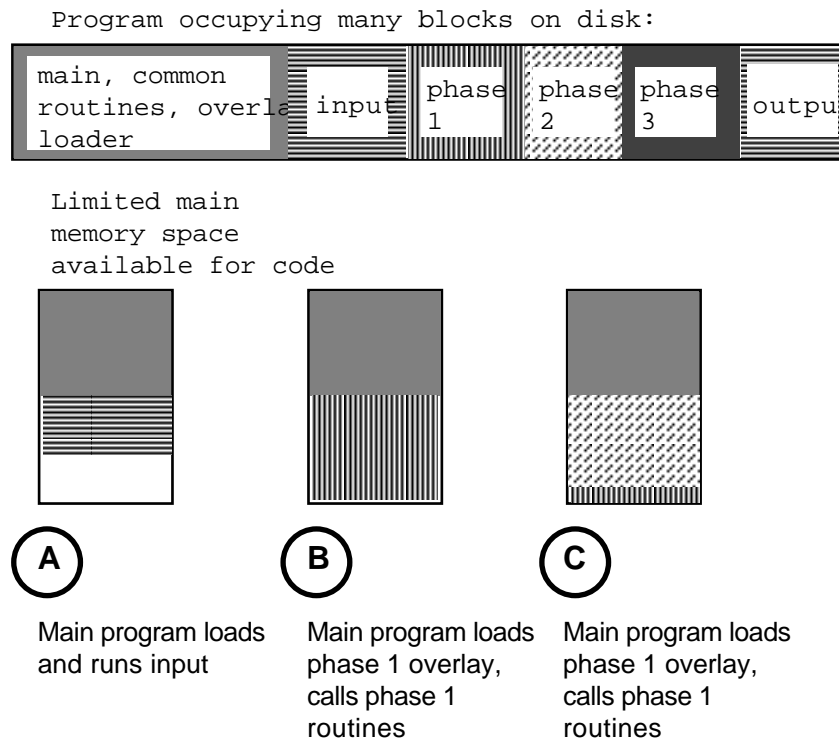


Figure 3.8 Program "overlay" scheme.

When shuffling code, programmers had standard 'overlay' models to follow. When shuffling data, they were on their own and had to invent their own schemes (because the best way of shuffling data between memory and disk depends on the frequency of access and the access patterns used in a particular program).

This need to bother about shuffling code and data around was a distraction. It didn't have anything to do with solving a problem (like working out something about heat flow in beams); it was simply a matter of fighting with limitations of the computing systems available. The programmers just wanted big programs with big "address spaces", but they were being forced to think in terms of a two level storage system – the main memory, and a secondary disk memory.

Another extension to the computer's operating system hid the sordid details of the "two level store" from the programmers. With a little help from some special extra hardware in the CPU, an OS can be made to fake things, so that it seems as if a computer has a very much larger memory than it really does possess.

The OS can arrange to know which parts of a program's code and data are in memory and which parts are on disk. When a chunk of code or data not in memory is needed, the OS can shuffle things around to get that code or data into memory. The programmer need never know that this is happening.

Paging

The Atlas computer built by the University of Manchester, England, around about 1960 was the first with what was then called a "one level store". It combined

16K words of real memory and storage on a drum. The OS, and special "paging" hardware in the CPU, faked things so that the Atlas machine seemed to be a computer with one million words of memory.

It took more than ten years before these memory management schemes became widely used on large computers. The name for the scheme was changed to *virtual storage* or *virtual memory*. (The use of the word virtual is slightly odd. It is really the same usage as in optics where one talks of "virtual images" being produced by magnifying glasses and by mirrors. The virtual store isn't real store – it just looks that way. The term has acquired wider usage, so multiprogramming is now sometimes discussed in terms of *virtual processors* – the OS makes it look as if you have many processors. "Virtual reality" is the latest extension.)

Virtual memory

In addition to special "paging" hardware that is needed in the CPU, support for virtual memory requires a lot more code in the OS. But now it is all commonplace; the CPU chips and OSs even for personal computers started to support virtual memory in the late 1980s.

3.2.5 "Spooling"

Developers of Operating Systems found yet another way in which they could make the OS fake things so that the computer seemed different from what it really was.

A program that wants to print data can ask the OS – "please send this information to the line printer". But what happens if you have a multiprogrammed computer, where the execution of several programs is being interleaved, and more than one of the programs wants to print data?

It obviously won't do to have a program print a line or two then stop, have another program start and print a line, then the first program resume. Interleaved printouts aren't much use.

The first solution was a little expensive; you had to have many peripheral devices. If you intended to have 4 memory partitions, for 4-way multiprogramming, then you had had to buy four separate printers. Although printer manufacturers liked this scheme, it was not widely popular.

Eventually, once most systems were equipped with disks, the OS developers found a work around. Programmers never send output directly to devices like printers – they merely think they do. The OS fakes it so that each program that wants to print thinks that it has its own exclusive printer ("virtual printers").

Rather than send output directly to a device like a printer, the OS would collect ("spool up") all the output from the program and save it in a temporarily allocated file on disk. Each running program could be allocated a separate area on disk to store its output temporarily. When a program finished, the file containing the output that it had written was tidied up by the OS and then transferred to a queue. A little printer control program would form another part of the OS. It would run the printer(s) attached to the computer. For each printer that the system had, the print control program would take a file from the output queue and arrange to get it printed in one piece. As each print file finished, the print control program would arrange to take the next file from the head of the output queue.

3.2.6 Late 1960s early 1970s system

By the late 1960s early 1970s, Operating Systems were becoming large and complex, see Figure 3.9. The figure shows some of the software components that would have made up the OS in memory (the relative sizes of components are not significant).

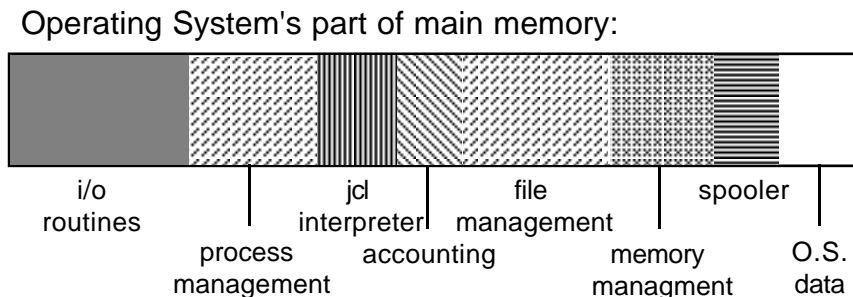


Figure 3.9 OS. of late 1960s early 1970s.

***"Device drivers" and
"interrupt handlers"***

There would be a number of i/o handling routines. These would include "device driver" routines that organize data transfers to/from peripheral devices. Other routines would deal with 'interrupts' that devices use to indicate when they've completed a data transfer. The OS would respond to interrupts by updating its records of which programs were waiting for i/o.

***"Process
management"***

Another large chunk of code would be devoted to "process management", i.e. the scheduling of the allocation of the CPU, the processor, to programs. This code kept track of which programs were ready to use the CPU and which were waiting for i/o from terminals, or disks, or other devices. If the system supported interactive users with 'time sharing', this scheduling code had to make sure that no program ran for too long without stopping to let other programs have some CPU cycles. There would be complex rules to apply to determine which to pick if there were several multiprogrammed jobs ready to run. The system would typically have a large number of jobs queued up on disk, these queued jobs would have been read earlier from cards. Another part of "process management" code would be run regularly to check whether any queued job should be started.

***Job control
interpreter***

The system would have some routines for interpreting the cards with "Job Control Language" (JCL) statements that specify the various processing steps a job required. "JOB" cards would have information that the JCL interpreter had to check to identify the job's owner (so that accounts could be kept and file use could be checked). Other JCL cards would specify things like the compiler (translator) to be used, or files needed to hold data.

File management

File management code would take up another large chunk of memory. This code would allow users to create files to store programs or data on disk; checks would have to be made to restrict users to allowed disk quotas, other checks would be made to see that only authorized users could access a file, and the code would have to make certain that disk blocks were allocated properly so that files didn't get

mixed up. File management code also had to look after things like providing temporary files for 'spooling' program output, special files for "swapping" time-shared programs, and other special files that modelled "virtual memory" for big programs.

Memory management code would organize the allocation of main memory to user's programs. This might be something relatively easy, like maintaining a fixed number of 'partitions' in which programs could run. Where used, things like "virtual memory" made memory management a lot more complex.

Memory manager

There would be some code used to keep accounts so that people could be charged for their computer usage. The OS code would also include minor specialized components like the "spooler".

Spooler and other minor components

All of the code components in the OS would have needed their own tables of data, these data tables would have taken up another area of memory.

OS data areas

3.3 NETWORKING

Originally, computers were located in isolated, air-conditioned computer rooms and they communicated only through their card readers, line printers, and the operators control terminal. Gradually, they began to be linked up. It was limited at first:

- machines like the MIT time-share machine would be connected to many terminals in surrounding buildings,
- a university with a central computer might have extra line printers and card readers located off campus at research centres; these would be linked to the computer by specially installed non-standard telephone lines.

The first real networked systems would have been the US's SAGE air defence system and the air-line seat reservation systems. These were fairly specialized. The main computer only ran a single program. SAGE had a program that kept track of aircraft movements; the seat reservation systems looked after files that contained details of bookings for each scheduled flight in the next couple of weeks.

The main computers were linked by special high-speed "leased line" telephone connections to smaller computers in other towns. These smaller computers acted as concentrators for messages coming from many terminals, either directly connected terminals or terminals linked via slow-speed telephone connections, see Figure 3.10.

During the 1960s, a number of organizations like large banks and airlines started to "network" their computer systems. But the scope was always fairly limited. The organizations had a main computer that ran one program – a seat reservation program, or a bank accounting record keeping program ("transaction programs" they keep track of individual transaction like the cashing of a check). The remote computers joined via special telephone lines just provided input data for transactions. But, a much more innovative scheme for networking appeared around 1968.

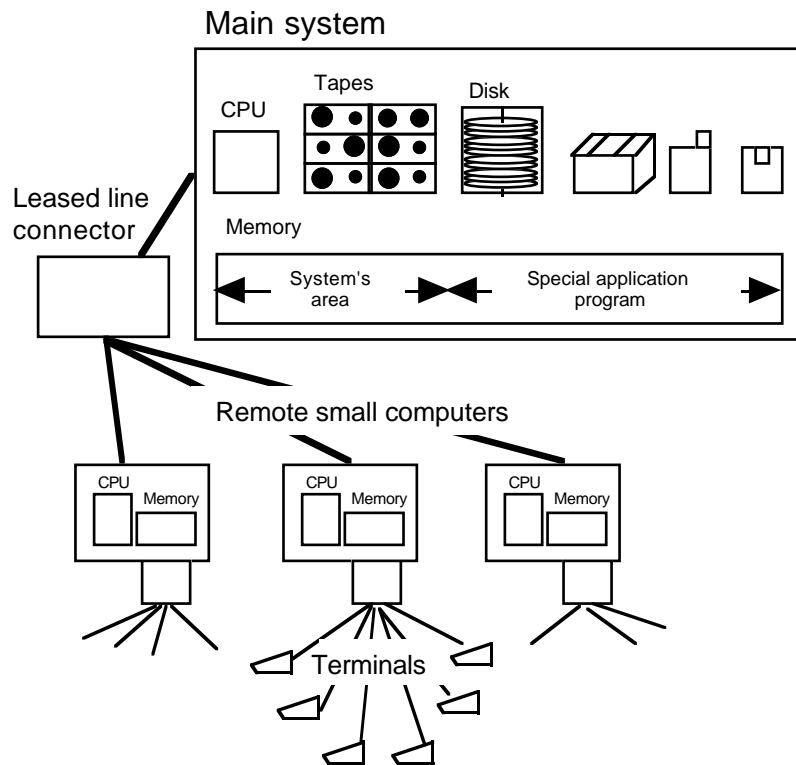


Figure 3.10 Early networked system, such as the airline seat reservation systems.

The "ARPA" net (Internet)

The US Department of Defence (DOD) sponsored a lot of research at universities and research institutes through its "Advanced Research Projects Agency" (ARPA). DOD-ARPA paid for large CDC computers for scientific computing at labs. like Oak Ridge, Los Alamos, and Livermore. It paid for DEC-6 and DEC-10 systems for research into Artificial Intelligence at places like MIT. There were ARPA funded scientists and engineers at many places remote from the centres with the ARPA computers. ARPA wanted to improve access to the shared computers by this wider community of users. This was achieved through the creation of the "ARPA network" (ARPAnet), which is the ancestor of the modern Internet.

The ARPAnet used small computers, IMPs (Interface Message Processors) connected by special telephone lines that allowed high speed data transfers. A particular IMP could be connected directly to one of the main CDC or DEC PDP-10 computers that were ARPA's shared computer resources. Other IMPs simply acted as terminal concentrators, connecting several local terminals to the network.

Scientist and engineers, who were allowed to use an ARPA computer, would have access to a terminal that was connected to an IMP located in their university or research laboratory. At this terminal, they could "login" to any computer on which they had an account --- e.g. `RLOGIN HOSTNAME=MITDEC10`

USER=WILENSKY. If login was permitted, the chosen host computer would send a response message back to the IMP with details of the job that had been started for the new user. The IMP would record the details – "user on terminal 3 @ me connected to job 20 on host MITDEC10." Subsequently, each line typed by the user was packaged by the IMP.

A packaged message had address details as well as the user's data e.g. if the user entered a "dir" command to list a file directory, the IMP would create a message like "3,IMP16,20,MITDEC10,dir" (terminal 3 @ me to job 20 @ MITDEC10, content 'dir').

An IMP didn't keep open a permanent connection to a remote host. Each message between user and host computer would be routed individually. Each IMP had a map of the network and the IMPs exchanged messages that described current traffic on links and identified areas with congestion. Using this information, an IMP could determine which link to use to forward a message. An IMP that received a message would check whether it should deliver it to a local host machine or forward it. Messages were stored in the IMPs until there was an opportunity to forward them across the next link. A message would make hops from IMP to IMP until it arrived at its destination.

The message would be taken by the host, and the contents delivered to the user's job. Output from the job would be packaged by the local IMP. The response message would work its way back through the network, the return route might differ from the original route.

The original intent of the ARPAnet was simply to provide this fairly flexible "remote login service" so that expensive ARPA computers could be shared by users all over the US. Very soon additional uses were found.

As well as having accounts and file allocations for individual users, most computer OSs have some form of "group accounts"; a systems administrator can add an individual user to any number of these groups. Once the ARPA net was running, researchers located at different places in the US could form a "group" on a particular research computer and work on shared files. Many found that the easiest way to contact their coworkers was to leave messages in shared files (avoiding problems with nobody answering phones, difficulties of different time zones etc). This kind of messaging led to primitive "electronic mail" (email) systems.

These mailing systems were regularized. Each user of a given host machine was allocated a "mail box" file; other users could append messages to this file. News services started about 10-15 years later.

Within a few months of its establishment, the ARPA net went international making use of a telecommunications link via one of the first geostationary satellites to link up with a European subnetwork.

Starting with just a few machines in the late '60s, the ARPAnet grew to several hundred machines by the late 1970s, several thousand machines by the late 1980s (with a name change to Internet – it went between other more localized networks) and now the network has thousands of machines. Nowadays, most host machines manage without IMPs, doing message packaging and routing for themselves.

In the mid-1970s, different types of computer network started to appear. By this time, computers were much less costly; and while it was common for an organization to have a large "mainframe" machine, this would be supplemented by

Message packets

*Store and forward
message delivery*

Remote login services

*Collaborative group
working and email*

Local networks

a large number of "minicomputers" of various forms. Each minicomputer would have some disks, a display screen and keyboard, but they typically wouldn't all have printers, tape units etc. Organizations set up "local networks" that simply joined up the different computers in any one office or department so as to allow them to share things like printers, and have some shared files. These shared file systems allow individuals working on different minicomputers to work together on a large project.

These networks use messages to pass data in way somewhat similar to the wide-area networks. For example if a user at one minicomputer typed in a command that specified that a file was to be copied from a local disk to a shared disk on another computer, the operating system on the mini would exchange messages with the computer that controlled the shared files; the first few messages exchanged would create the new file, subsequent messages would copy the existing file chunk by chunk.

***Broadcasting
message packets***

The message passing mechanism was rather different from that used on the wide area networks. There are a variety of local network schemes ("token ring", "ethernet") with ethernet being currently the most popular. All the computers on an ethernet are connected to something like a co-axial cable. Messages are "broadcast" on this cable. All the computers can read messages at the same time, but only the computer to which a message is addressed will actually deal with the data.

Ethernet networks

Modern personal computers typically have hardware interfaces and software in their operating systems that allow them to be connected to "ethernet" style networks. Each individual ethernet network has to be limited in size, but different local networks can be joined together by "bridges" and "routers" that can forward messages from one network to another. (A bridge or router serves much the same purpose as an ARPAnet IMP; it is a simple computer that runs a program that organizes the forwarding of messages.) Often local networks will have "bridges" that connect them to the worldwide Internet.

3.4 MODERN SYSTEMS

The operating systems written for the mainframe computers of the late 1960s and early 1970s did succeed in delivering efficient CPU usage through "multi-programmed batch computing", as well as supporting interactive timesharing. But these systems were often difficult to use (the "job control language", JCL, statements that had to be written to run a job were hard for the average user to understand, and the job control mechanisms were quite limited). The systems were proprietary; each computer manufacturer supplied an operating system for their computers, but these different operating systems were quite different in their styles of JCL and the facilities that they provided. Such differences hindered those users who had to work with more than one kind of computer.

Further the code for these OSs was often poor. Most of the systems had not been designed, they had just grown. More and more features would have been added to an original simple multiprogrammed OS. Consequently, the systems were difficult to maintain.

Modern operating systems started to appear in the 1970s. Their designers had different objectives. With hardware costs declining, there was no longer a need to try to maximize machine usage. Some computer power could be "wasted" if this made the system easier to use and so made the users more productive. Unix was one of the first of the modern systems; its designers sought to create a system that would make programmers more productive. Later systems, like the Macintosh OS, have focussed on the needs of other less sophisticated users.

3.4.1 UNIX

Unix started to be developed around 1969, with the first published description appearing in 1973. It was developed at an ATT research lab. and was made freely available to universities, encouraging its widespread adoption.

Unlike earlier operating systems which had been written in assembly language, the code for Unix was largely in C. The use of a high level language made the code much easier for programmers to understand and maintain. Further, the Unix OS was designed! The programmers who developed Unix started with a clear idea of how their OS was to work and what services it was to provide.

Unix was more limited in its aims than many other OSs of the time. It was intended solely to provide a good environment for timeshare style program development. Other OSs were attempting to do timesharing, and database transactions, and run large jobs, and ...; but such different uses of a computer tend to conflict resulting in poor performance in all areas.

The design for Unix modelled the OS in terms of several layers (when describing the design, someone made an analogy with a nut or an onion and introduced terms like "kernel", "shell" etc --- these names have stuck):

- the innermost layer (the "kernel") has the code for the i/o handling routines ("device drivers") etc;
- another layer contains the code for process management, file management, and memory management;
- further layers contain code for looking after wide area and local networks and so forth;
- the next layer out comprised large numbers of useful utility programs – programs for copying files, comparing files to find differences etc
- the outermost layer (the "shell") was the job control language interpreter, but this JCL interpreter was much more flexible than any that had been proposed previously.

Unix was originally written for a particular kind of computer (the "PDP11/20") manufactured by Digital Equipment Corporation (DEC). But, the relatively clean design of the system, and the use of a high level language, made it possible for the system to be adapted to other computers (only the "device drivers" and other really low level code had to be redone). Unix was moved to related but more powerful computer architectures (DEC's VAX series of computers) and to totally different computer architectures. During the 1980s, Unix was adapted to run on computers

as diverse as the modern Cray supercomputers down to personal computers with Intel-80386 CPU chips.

The US Department of Defence's Advanced Research Projects Agency (ARPA) sponsored development of Unix at the University of Berkeley. The Berkeley developers added features to support virtual memory and networking (both wide area and local networking). Late in the 1980s, many computer companies, ATT, Berkeley University, IEEE, etc got together and established standards for all Unix systems.

Unix thus has the advantage of being a system that is non-proprietary, is widely available, and is effective in its original role of supporting program development. Most students continuing with computing studies will eventually get to work with some Unix systems. Modern Unix systems have been expanded so as to handle tasks other than the "programmers' workbench" of the original design. These extensions (to handle large databases, some transaction processing and so forth) were demanded by customers. In some respects, these extensions detract from Unix which no longer has a quite the simplicity and elegance of its early forms.

3.4.2 Macintosh OS

The Macintosh OS (1984), and things like Windows 3 (late 1980s), represent more modern operating systems, having evolved in the ten to fifteen years after the start of Unix.

The important ideas in the Mac OS (and later systems meant to work in similar style) were developed at Xerox's Palo Alto Research Centre during the 1970s and early 1980s (Apple started the Mac OS by getting a licence to use Xerox's ideas).

Starting around 1972, Xerox PARC had had a project that aimed to explore what the "office of the future" would be like. Obviously, the office workers were going to make heavy use of computers. The Xerox researchers realized that the old systems were inappropriate.

The old systems had the computer as sort of oracle, surrounded by priests (the system's programmers and system's administrators) and neophytes (the computer operators); even the newer Unix systems had to have "system's gurus" to attend them and keep users at bay.

In an "office of the future", individual's would have their own computers, and these therefore would have to have operating systems that did not need priestly ministrations from gurus or others.

Unlike other developers of that period, the Xerox group realized that the cost of CPU power was going to drop dramatically. Consequently, it wasn't going to be important to keep the CPU efficiently employed, what was going to be important was the efficient use of time of the office workers. So it was going to be worthwhile "wasting" CPU cycles with the computer doing extra work if this would simplify the tasks of the user.

Given these premises, the Xerox group focussed on what they thought would be the needs of users; they identified factors such as:

- visual displays for "high-bandwidth" communication (show the user what programs and files are available for use etc);
- direct manipulation (use of mouse pointer, selection of object represented visually [as an "icon"], picking a command from a menu – the "point-and-click" interface rather than the "remember-and-type-command" interface of Unix and older systems);
- consistency (every program working in a similar manner);
- intercommunication (e.g. easy transfer of pictures, text and other data between programs).

Xerox developed a variety of experimental systems embodying the features that they felt would empower users and make computers more useful. However, Xerox never really brought these experimental systems to the level of practical, affordable products.

Steve Jobs and others at Apple in the early 1980s recognized the importance of the Xerox ideas and worked to make them practical. The Mac OS of 1984 was the first system that could really deliver computer power to all users.

4 Why have "high-level" languages?

4.1 LIMITATIONS OF ASSEMBLY LANGUAGE AND ORIGINS OF HIGH LEVEL LANGUAGES

Chapter 2 introduced to the idea of a program as a sequence of instructions in the memory of the computer.

It is possible for a programmer to write programs in this form. The programmer writes an "assembly language" program. The source text of this program is translated into binary by an assembler. The binary information representing instructions is put into the computer's memory by a loader.

Generally, programmers avoid assembly language, working instead with one of the numerous "high level languages".

Assembly language programming has two main disadvantages:

- it involves far too much detail;
- it is necessarily specific to a particular make of computer.

*Problems with
assembly language
programming*

The excessive detail comes from the requirement that the programmer choose the exact sequence of instructions and data movement operations needed for each task. Since different makes of computer have slightly different instruction sets, a program written in terms of the instruction set for one computer cannot be used on a different kind of computer.

As noted in Chapter 2, assembly language programs are built up from small fragments of code that follow standard patterns ...

- Sequence pattern, instructions for evaluating an expression like

$$v = u + a * t$$

- Loop pattern, dealing with a group of operations that must be executed several times

- Selection pattern, dealing with choices
- Subroutine pattern, breaking down a complex process into simpler sections.

It is easier if the programmer can use concise statements to specify the required patterns instead of having to write the corresponding instruction sequences.

e.g.	this expression is easier to read	than this code
1)	$V = U + A * T$	<pre>load t mpy a add u store v</pre>
2)	<pre>T = 0.0 DO 10 I=1,10 DIST = U*T + 0.5*A*T*T 1100 WRITE(5,500)T, DIST T = T + 1.0 10 CONTINUE</pre>	<pre>move fzero, t move #1, i compare i, #10 jgt 1101 fload t 30 more instructions</pre>

Assemblers made it possible for programmers to work in terms of readable text defining instructions rather than bit patterns. The assembler program does the translation task converting the text instructions into the bit patterns.

Compilers

High-level language translators (compilers) allow programmers to write statements. A compiler can translate a statement into an appropriate instruction sequence (which can then be input to an assembler).

Just like assemblers, compilers rely on rules that define the allowed forms for the different possible statements. The rules defining the form of an assembly language are very simple:

- a line of an assembly language program can be
an optional label and an instruction
or
a variable name and an initial data value
- an instruction can be
an i/o instruction
or
a data manipulation instruction in the form

instruction-name operand
or
...

Translation involved little more than looking up the instruction name in a table to get the corresponding bit pattern.

Syntax rules for high level languages

The rules defining the form of a high level language are naturally a lot more complex than those for an assembler because we want a high level language to

allow the direct use of more complicated constructs. These rules defining the allowed forms in a language define the *syntax* of that language.

4.2 HIGH LEVEL LANGUAGES CONSTRUCTS

4.2.1 Statements

A language will typically allow for a number of different kinds of statement:

Statement types

- assignment statements (define a new value for a variable in terms of some expression that combines values of other variables, e.g. $V=U+A*T$)
- selection statements (provide something that allows the programmer to indicate selection of one operation in a set of two or more alternatives)
- iteration statements (loop constructs)
- subroutine call statements
-

Most high-level languages still exhibit relics of their assembly language ancestors. A program is still a sequence of statements that are normally executed in sequential order (with appropriate adjustments for subroutines, loops and selection statements). Each statement may expand to many instructions, but it is still the same "instructions in the machine" model of computations.

The people who invented a particular high-level language will have chosen the forms of statements allowed in that language. The chosen forms will represent a trade off among various factors. They have to allow programmers to express complex constructs succinctly (and therefore they may be moderately complicated) but at the same time they have to be simple for an automatic translation program to recognize and interpret.

The first few examples in this section use FORTRAN statements. **FORTRAN** is about the oldest of the high level languages (first used in 1956) but it is still quite widely used (the language has been updated a few times).

The forms of statements in the earliest dialects of FORTRAN are obviously related to simple, standard patterns of machine instructions as would have been familiar to assembly language programmers in the 1950s. (Some critics of the continued use of FORTRAN point to the absurdity of still writing code for the assembler used on an IBM704.) For example, the early FORTRAN dialect provided programmers with a simple "arithmetic if" test as one of its few conditional constructs ...

```
C      CHANGE MAXTEMP IF TEMP GREATER THAN CURRENT MAXTEMP
      DIFF = MAXTEMP - TEMP
      IF(DIFF) 150, 160, 160
150    MAXTEMP = TEMP
160    CONTINUE
```

In FORTRAN, lines starting with a 'C' were comment lines that a programmer could use to explain what was being done in the code. The 'assignment statement'

on the second line (`DIFF = ...`) calculates the difference between the current maximum temperature value and some newly calculated value. `DIFF` will be positive if the current value of `MAXTEMP` is already larger than new value. This code uses an 'arithmetic if' statement. The part `IF(<variable name>)`, like `IF(DIFF)`, had to be translated into a sequence of instructions that would test whether the value of the variable was negative, equal to zero, or positive. The rest of the 'arithmetic if' statement comprises three label numbers. These label numbers identify the statement that should be executed if the value was negative, 0, or positive.

In the example program, if `DIFF` is negative, the `MAXTEMP` value must be changed. This change is done by the code (starting at) statement 150. If `DIFF` is zero or positive, then `MAXTEMP` is equal or greater than the new value, and so no change is needed. Rather than execute statement labelled 150, the CPU executing the program can jump to statement 160.

The simple structure of the "arithmetic if" statement made it easy to define

- 1) rules that a compiler could use to recognize such a statement

and

- 2) a small code template that the compiler could use to generate code, e.g.

```

IF(<variable>) <label1>, <label2>, <label3>

translate to
fload      <variable>      (loads real (float) number
                             into a CPU register)
ftest
blt  <label1>              (if "less" flag set ,
                             branch to label1)
beq   <label2>              (if "equal" flag set ,
                             branch to label2)
jmp   <label3>              (otherwise go to label3)

```

The arithmetic if statement was one of the simplest in the FORTRAN language. Other constructs, like `DO ...` (the construct used to form iterative loops), required much more elaborate code templates for their translation (as well as more complicated recognition rules). The template for translating a `DO ...` construct would have needed to use something like 20 assembly language statements specifying the various instructions needed. These would have come in two groups; one group at the start of the iterative code, the other group at the end.

These high level `IF()...` and `DO ...` constructs saved the programmers from having to sort out all the specific instructions required, so giving them more time to think about the best way of coding a problem.

Productivity gains

Numerous empirical studies have shown that programmers generate about the same number of lines of code per day irrespective of the language that they use. It is a surprisingly low number of lines (lets say '50' to protect the programmers' dignity). If you program in assembly language, you get one instruction per line so you are getting about 50 instructions per day. If you program in FORTRAN (or another higher level language), your one line of code would generally translate into

five or more instructions, so 50 lines of code meant that you were coding at rate exceeding 250 instructions per day. High level languages give a big boost to programmer productivity.

In addition to getting improved programmer productivity, high-level languages give you machine independence. There might have been other computers around that didn't have a "test" instruction, nor instructions like "branch if less flag set" (in fact, the other CPU might not have had any status flags). A programmer using a high level language like FORTRAN didn't have to bother about such differences; the compilers took care of such details.

**Machine
independence**

The "arithmetic if" statement illustrated earlier would have had to have a quite different translation into machine instructions ...

```
IF(<variable>) <label1>, <label2>, <label3>
```

might, on the second machine, be translated using the template

float	<variable>	(loads real (float) number into the CPU's float register)
spfa		(skip, i.e. miss out next instruction, if the float register contains +ve value)
jmp	<label1>	(deal with -ve case)
snfa		(skip if register value non zero)
jmp	<label2>	(deal with zero case)
jmp	<label3>	(deal with +ve case)

If you had equivalent FORTRAN compilers on both computers, you weren't bothered by such differences. You could rely on the compiler for a particular machine to use a template with the appropriate set of assembly language statements that used the correct instructions for that machine.

4.2.2 Data types and structures

A high-level language and its compiler do more than make it easier for the programmer to code the data manipulation steps.

High-level languages allow a programmer to declare restrictions on how particular data elements are to be used. After a compiler has read such declarations, it will remember them and when it sees reference to variables later in the code it will check whether the restrictions are being obeyed. Such checks cannot be done in assembly language.

When writing in assembly language, a program can only define a variable as "something" that occupied a particular location in memory. If part of the program treats that variable as if it held a real number, and another part treats the variable as an integer number, the assembler program won't detect any errors. It will produce executable binary code which will give results that are almost certain to be incorrect.

High level languages mostly use explicit variable declarations that define the "type" of a variable (sometimes, a language may rely on a naming scheme where

the type of a variable can be inferred from its name). Thus, in some versions of FORTRAN one could have declarations for variables like the following ...

```
REAL MAXTEMP, MINTEMP
INTEGER TIME
...
```

Such declarations were usually put at the start of a program (or subroutine) or sometimes in the code at a point just before the place where a variable was first used. Such declarations allowed the compiler to check that, for example, MAXTEMP was always treated as a real number.

The original 1956 FORTRAN compiler also introduced mechanisms that allowed a programmer to define "structure" in their data. "Data structures" allow a programmer to express the fact that there may be some conceptual relationship between different data variables.

Ideas of data structures have been developed substantially over the years, the original FORTRAN idea was quite limited; it was simply a way of allowing the programmer to express ideas about "arrays" (or "matrices"). Taking again the example of an engineer writing a program to model heat flow along a metal beam. It would be necessary to calculate temperatures at several points along the beam and across the beam. For example a small study might need ten divisions along the beam, with four positions across the beam at each division (Figure 4.1).

	1	2	3	4	5	6	7	8	9	10
1										
2										
3										
4										

Figure 4.1 An array, or matrix, of places where an engineer wants to analyze properties of a steel beam that gets represented as an array of data values in a program.

The program would have to calculate forty different values; but the engineer programmer really wouldn't want to think of these as being 40 different variables (BEAM1, BEAM2, ..., BEAM40) because that would hide the nature of the physical system that the program is supposed to model.

A programmer working in assembly language would not be able to specify anything clearer than a need for space in memory to store 40 numbers. A programmer using FORTRAN could specify an "array structure" that helped make it clearer that these 40 numbers really did represent something involving 4 "rows" of ten values each ...

```
DIMENSION BEAM(4,10)
```

As well as helping to clarify the meaning of the data, such a declaration of an "array structure" allowed simplifications of code. Rather than a whole set of different formulae to calculate the value for BEAM1, BEAM2 etc, the program could be written using nested loops and a single formula that calculated the value for BEAM(I,J) (with I and J being variables that ran through the range of row and column positions)...

```
DO 10 I = 1, 4
DO 9 J = 1, 10
...
BEAM(I,J) = ....
9 CONTINUE
10 CONTINUE
```

4.3 EVOLUTION OF HIGH LEVEL LANGUAGES

The original FORTRAN compiler of 1956 started a massive amount of development work on "programming languages". That early FORTRAN was a vast improvement over assembly language, but was also limited and flawed. Subsequent languages have tried to remove limitations, add expressive power, and employ variations on the basic "sequence", "iteration", "selection", and "subroutine call" patterns that provide fewer opportunities for errors.

Reducing opportunity for errors:

That FORTRAN arithmetic if statement is confusing; e.g. :

```
IF(NUM) 150, 160, 170
```

If NUM is positive we go to statement 150, or would it be statement 170?

As the different pieces of code are scattered around in the text of the program, and reached by jumps backwards and forwards, it is very difficult for someone to read the code and get a clear idea of what is supposed to be happening.

The complete FORTRAN example given earlier was supposed to replace MAXTEMP with some new value TEMP if this was larger. The FORTRAN was

```
DIFF = MAXTEMP - TEMP
IF(DIFF) 150, 160, 160
150 MAXTEMP = TEMP
160 CONTINUE
```

More modern languages (Pascal, C etc) allow a much clearer expression of the same idea: e.g. in Pascal one can have something like ...

```
if(temp>maxtemp) then maxtemp := temp;
```

This clearer style makes it much less likely that a programmer will make a mistake when coding.

Greater expressive power:

The original FORTRAN only had "counting loops" e.g.:

```
DO 100 I= 1, 25
... (body of loop doing some work)
100 CONTINUE
```

This is fine if you know that a loop has to be executed 25 times (with the variable I counting from 1 to 25).

But often there are other conditions on which you want to iterate, e.g.:

- loop reading commands until user types 'quit';
- repeatedly process successive data elements until a negative value is found;
- ...

Later languages have their own variations on the counting loop construct; in most languages this will be a "for" loop, e.g. Pascal's

```
for i:=1 to 25 do begin
...
end;
```

But the more modern languages will have other constructs such as "while" loops and, maybe, "repeat" loops; e.g. a Pascal "while" loop to process non-negative input values:

```
readln(val); (* read first value *)
while(val>0.0) do begin
... (* code to process value *)
readln(val); (* get user to input next value *)
end;
```

More data structures and more compile type checking

FORTRAN is pretty much limited to arrays of simple real (or integer) numbers. Modern languages provide other built in types (e.g. character data for text processing programs, "boolean" data – true/false values) and give the programmer many more ways of building elaborate data structures from simple built in data types.

A large part of the second and subsequent programming subjects in most computing science courses are largely devoted to exploring the definition and use of data structures.

The compilers (particularly those for advanced languages like C++) do lots of checking to make certain that any structures defined by a programmer are used only in appropriate ways.

4.4 FORTRAN

FORTRAN was the first of the high level languages. In 1956, IBM started supplying a FORTRAN compiler to customers of its 704/709 series of computers (these were machines intended for scientists and engineers).

The name FORTRAN came from "FORMula TRANslator" --- that is how the developers thought of their compiler, it "translated" scientific and engineering formulae into assembly language code that specified the appropriate sequence of instructions. IBM didn't claim any exclusive right to the FORTRAN language and fairly soon other computer manufacturers made their own FORTRAN compilers (along with compilers for similar, but less successful languages as invented by companies or universities).

Since one of the reasons for using a high level language was machine independence, it was recognized that different FORTRAN compilers would have to be standardized so that all used exactly the same kinds of statements and data declarations. Standardization initially involved semi-formal agreements among the computer companies but later the US government's "American Standards Association" became involved. By the late 1960s, there was a government approved standard FORTRAN dialect (ASA FORTRAN IV).

Standardization

Standard FORTRAN was subsequently extended to incorporate ideas invented for other languages.

Program organization

FORTRAN defined a program as consisting of a "main program" along with any number of subroutine units.

Subroutines were divided into "functions" and standard "subroutines". Functions were short routines that simply calculated a value e.g. finding the sine() or cosine() of some angle given in radians.

Some compilers could handle a file (actually a deck of cards in those days) that contained more than one routine; but really each separate routine was thought of as a separately compilable unit. This support for separate compilation encouraged the development of libraries of FORTRAN subroutines that had been written to handle subtasks that occurred frequently in engineering applications

*Independent
compilation of
subroutines*

A "linking loader" was used to put together a FORTRAN program from its separately compiled parts and library routines, see Figure 4.2.

Linking loader

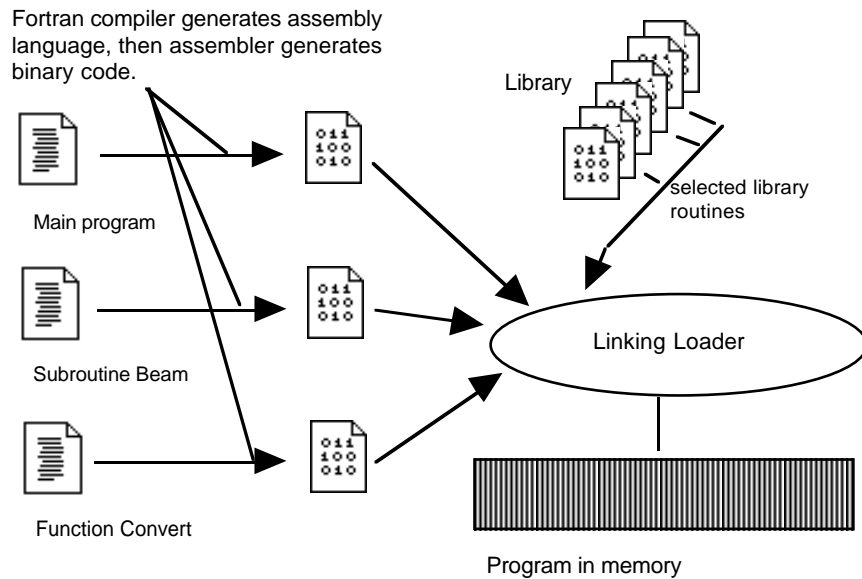


Figure 4.2 Linking a program from separately compiled and library files.

All the data variables used by a program were defined by declarations in the main program, its functions, and its subroutines. The compiler and link-loader worked together to choose fixed memory locations that would be used for these variables. In early systems, the space reserved for variables was usually located just after the code of a unit (see Figure 4.3). This "static" allocation scheme is a bit restrictive and later languages offer alternative, more flexible but more complex schemes.

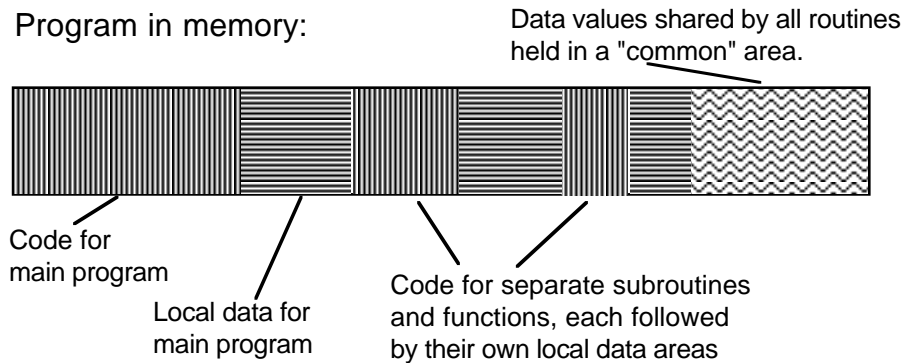


Figure 4.3 Simple arrangement of code and data areas in memory (as used by early FORTRAN compilers and similar systems).

Although generally a good way of putting together a program out of parts, the original FORTRAN scheme didn't allow for much checking for correctness. For

example, a subroutine or function might need to be given a real number as input (e.g. FUNCTION SINE(X) wants a real value for X) but by mistake the programmer writing the main program might have called the function asking it to work with an integer value. Such an error would pass undetected, the program would be built and would run but would give incorrect results. Later languages have explored alternative ways of putting programs together, ways that do allow checks for consistency.

Each individual unit (main program, subroutine, or function) would consist of data declarations and code statements. The data declarations named individual data elements

Data declarations

```
REAL MAXTEMP
INTEGER TIME
```

Simple variables

and structured composites of simple data elements (the arrays):

```
DIMENSION BEAM(4,10)
```

Arrays

The statements included:

Statements

- assignments:

```
DIST = U*T + 0.5*A*T*T
```

- conditionals:

```
IF(MAX) 500, 510, 520
```

(there were several other forms of IF statement apart from this arithmetic IF),

- iterative loops:

```
DO 100 I=1,50
...
100 CONTINUE
```

and

- calls to subroutines:

```
CALL MODELHEATFLOW
```

Arguments for routines:

Main programs and subroutines have to communicate.

A subroutine will have been written to take some data values as input and perform various calculations that change those data. But which data elements does a routine work with?

Sometimes, a subroutine will only work on one set of data. In this case these data elements can be declared by the main program as being in some "common" area available to all routines.

But often you want a routine (or function) to work on different data elements, e.g. you might want to calculate the sines of several different angles (represented by several different data elements in the main program). In these situations, the calling program must have some way of telling the subroutine about the specific data element(s) it is to work with.

***"Passing arguments
to a subroutine"***

There are many ways that a calling program can tell a subroutine what data to work with (you will be meeting some of these alternative ways when you start programming). The scheme used in FORTRAN relies on the main program telling the subroutine where in memory the data can be found (it does this by "passing" the addresses of the arguments to the subroutine or function).

For example, a subroutine for sorting elements in a vector with up to 100 elements would need to be told which array to sort and how many elements it actually contained. Such a subroutine would specify this in its header:

```
SUBROUTINE SORT(DATA,N)
  DIMENSION DATA(100)
  ...
END
```

A main program might contain more than one call to this sort routine, with different arguments in each call, so that it could get the contents of more than vector sorted ...

```

DIMENSION HEIGHT(100), RANGE(100)
...
...
CALL  SORT(HEIGHT,NCOUNT)      pass address of
                                vector HEIGHT and
                                variable NCOUNT to
                                sort subset of data
                                in vector
CALL  SORT(RANGE, 60)          pass address of
                                vector RANGE, and
                                address of a compiler
                                invented variable set
                                to contain 60.
```

I/O package

FORTRAN came with a fairly good library of input and output routines. These allowed programs to save data to tape (and disk), to read previously saved data from tape, to read "cards" with data, and to print results on a line printer.

***"Formatting" of
output***

The routines for reading "cards" and printing provided for elaborate "formatting" of data (i.e. there was a way of specifying the layout, or "format", of data on a printed page).

Use of language

For something that started as a little project to help engineers write assembly languages on an early computer, FORTRAN has had an enormous success.

For those who have to code up engineering calculations, FORTRAN provides just about everything needed and it is not hard for compilers to generate good assembly language code. Consequently, the language is still widely used by engineers. The CPUs for the "supercomputers" used for large engineering calculations have often been designed with FORTRAN in mind. (Modified dialects of FORTRAN are sometimes the only programming language for special parallel supercomputers).

But for anything other than engineering calculations, FORTRAN is clumsy and most programmers prefer to work with other more sophisticated languages.

4.5 BASIC

Basic was originally developed at Dartmouth University in the USA in the early 1960s.

It was intended to be a simplified version of FORTRAN-II (a revised FORTRAN dialect of the early 1960s) that could be used for small programs entered and run interactively on an early time-shared computer system. (It simplified FORTRAN in a number of ways such as by ignoring the differences between real and integer values, and by restricting the number of data variables you could have.)

It was also an interpreted language. BASIC code was not translated into assembly language and thence to loadable binary. The BASIC interpreter would start by simply checking that it could recognize each statement in the program. (The statements were similar in type to those of FORTRAN with assignments, IF tests, loops, and "GOSUB" calls.) The program would then be run with the interpreter stepping through the successive statements.

As each statement was reached, the interpreter would reanalyze it and then immediately carry out the operations required. So, for a statement like:

```
110 LET V = U + A * T
```

the interpreter would get the value for variable U, save the value temporarily, get the value of A, multiply by the value of T, add the saved temporary value and store the result in V.

An interpretive system is far less efficient than a compiled system. A BASIC interpreter might have to execute tens of thousands of instructions to evaluate something like $V=U+A*T$ (whereas only about four or five instructions of compiled code are required). So one would never use an interpretive approach for large scale numeric calculations.

But an interpreter has some advantages. Interpreters avoid the complexity of the separate compilation, assembly, and link-loading steps that are needed by a

*Interpreted
languages*

*Costs of an
interpretive system*

*Advantages of an
interpretive system*

compiled language. Interpreters can often detect "syntax errors" (where a statement is not recognizable) immediately after the user types in a line of the program; the errors can be identified and the user gets a chance to correct them. Often, interpreters can provide extensive run-time checks so if something goes wrong while a program is executing, it doesn't just stop, instead, the interpreter generates some explanation as to what is likely to be wrong. Again, this makes it simpler for the programmer to find and correct errors.

BASIC popular as an introductory language

Such advantages seemed to make BASIC an attractive environment for students and school children to learn how to program. For a long time (from 1962 until the mid 1980s) BASIC was used in introductory courses but it gradually fell from favour. The structure of BASIC was based on FORTRAN, and it shared FORTRAN's faults of error prone constructs, and a style of code with too many jumps back and forth between labelled statements. Programs written in this style are difficult to get to work correctly, and even more difficult to maintain.

Most programmers prefer working with programming languages (mainly from the Algol family) that have more structure in their basic loop constructs, conditionals etc. (Modern dialects of BASIC have adopted some of these structured programming constructs.)

4.6 LISP

Lisp is another very different example of an interpretive language. Lisp first appeared in 1960, a product of research on "Artificial Intelligence" being carried out at Massachusetts Institute of Technology.

A language for symbolic computations

Lisp is not a language for writing programs that do calculations --- it is intended for "symbolic computation". What is "symbolic computation"?

A numerical computation:

Calculate the value of the polynomial when $x = 3.8$

$$5x^4 - 7x^3 + 4x^2 + 2x - 11$$

A symbolic computation:

Differentiate the polynomial

$$5x^4 - 7x^3 + 4x^2 + 2x - 11$$

In the case of numeric computation, the programmer has simply to code up the formula so that the computer can work out the answer:

```
X = 3.8
VAL = (((5*X - 7)*X + 4)*X + 2)*X - 11
```

For a symbolic computation, the programmer must create a much more complicated program that can read in some representation of the polynomial and produce output that represents the polynomial's derivative:

$$20x^3 - 21x^2 + 8x + 2$$

(Most students will get to use a program called Mathematica that does this kind of symbolic calculation. Mathematica is a descendant of a large Lisp program called Mathlab; the current version of Mathematica may have been written in C rather than Lisp.)

Lisp was invented to handle all kinds of non-numeric calculations. As well as "mathematical applications" like symbolic differentiation and integration, Lisp has been used to write programs that:

- translate English to Spanish or to other natural languages,
- "understand" children's stories,
- interpret photos of rooms and pick out desks, chairs, doors etc,
- execute rules that diagnose microbial infections
- check architectural drawings to make certain they comply with local government ordinances
- solve those "analogy" problems you get in IQ tests
- infer rules that capture regularities in set of data

Dynamic data structures

It is very difficult to determine in advance what data elements are going to be needed in the kind of application where Lisp is used. Examples:

- the number of terms needed varies with the polynomial given as input
- the number of chairs, doors, desks etc varies with the picture used as input
- the number of inferences made while determining the cause of a patient's infection depends on both the data that define the patient's clinical history and on the organism causing the infection (some are easy to identify and don't involve many rules, others require lots of rules to discriminate among less common types of organism)

Instead of having all the data variables allocated before the program started, a Lisp program started with a large collection of "free storage". This "free storage" is allocated dynamically --- pieces being used as and when a program needs them. Whenever a Lisp program needs a new data element, it takes pieces of the free space and uses them to build up the data structure required. *Free storage*

Other languages subsequently copied Lisp and provided their own forms of dynamic storage (though most copies are actually less cleverly implemented than the original scheme used in Lisp).

4.7 COBOL

COBOL is yet another language from around 1960. Its name stands for "COmmon Business Oriented Language" and it is intended primarily for applications like record keeping and accounting.

Businesses had been using files of record cards to keep track of customers since the late 1890s. These record cards were processed using electromechanical sorting machines, tabulators (which printed data in tabular form on sheets of paper), and more complex accumulators (that added up numeric values recorded on the punched cards). Computers started to be used for such record keeping applications and accounting from about 1949. The general public probably first heard of computers when a Univac computer appeared on television in 1952 while being used to predict the result of the US presidential election.

Initially, commercial applications programs had to be written in assembly language. Of course, assembly language was just as unsuitable for commercial applications as it was for engineering calculations. So high level languages started to be invented for commercial work.

*DOD requires a
standard data
processing language*

In the late 1950s, each computer manufacturer was developing their own commercial/business data processing language. The US's Department of Defence was buying computers from many manufacturers. It needed "business" programs (e.g. programs to do the army's payroll, programs to keep track of munitions in warehouses, etc). It wanted its programmers to be able to write programs that could run on any of its computers. Consequently, it was not pleased to find that the languages were different! The DOD sponsored a consortium of computer manufacturers to come up with a common language — COBOL.

On the whole, COBOL is not an interesting or attractive language. It is clumsy. It is verbose. It has a poor subroutine structure. But it did do one thing very well: it allowed programmers to define "data records" and provided support for files of records on tapes and disks.

Business data processing and "records"

Typical business processing applications need to keep track of things like "customers", or "employees", or "hospital patients" etc. Each such thing is characterized by many separate data values, e.g. for a hospital patient one might need:

patient's name, patient's given names, age, sex,
insurance number, account number,
date of admission, ward number, bed number,
doctor in charge, reason for admission,
info on surgery required
bill
...

These many different values all are parts of the same thing.

If these data were represented by separate variables in a program, one would lose track of the fact that they belonged together. COBOL (and some of its defunct predecessors) allowed programmers to define "records" (they were inspired by "record cards" that businesses had previously used to keep track of things like customers).

A programmer could declare a record, naming and defining the types of constituent fields. So a "patient record" might be defined as having a 'name' field (with space for 30 characters), a 'given names' field (space for 60 characters), account number (an integer with 6 digits), a ward number (integer with 2 digits), and so forth.

Record data structures

Given a declaration of such a patient record, a COBOL compiler would then allow the programmer to have variables of type "PATIENT". The compiler would arrange for the variable to be represented by a block of memory sufficiently large to hold all the constituent data fields of a patient. The language then allowed the programmer to refer to data fields within a particular PATIENT record (this helped make it clear that related data fields did belong together).

As well as having PATIENT records in memory, a COBOL programmer could specify that the program needed to use a file of such records held on tape or disk. The compiler would provide lots of support for operations like reading a complete record from file into memory and writing updated records back to files.

Record files

The idea of a "record" grouping together related data is very useful in most applications, not just business data processing. Modern languages like Pascal, C, and C++ incorporate the idea of records. You will start to use records (simple things like "patient" records) towards the end of your first programming course and spend considerable time on more elaborate record structures in many of your later courses.

4.8 ALGOL

1960 again! A good year for programming languages, Lisp, COBOL and *Algol-60*.

FORTRAN had just happened. No-one designed it. Its developers had simply sat down to write a "Formula Translator" that would simplify the work of those writing programs to do scientific or engineering calculations. It had no style, no elegance.

A number of mathematicians wanted something better: a language for expressing algorithms, all kinds of algorithms including recursive ones. After an abortive effort in 1958, in 1960 a group of mathematicians came up with a design for an ALGorithmic Language.

Algol really was conceived of mainly as a mathematical device. It was meant to allow mathematicians to express complex algorithms: algorithms for mathematical tasks like "inverting matrices", finding "eigenvalues of matrices", solving linear equations, calculating recursive functions. It wasn't really thought of as a practical programming language, for example, the original version didn't bother to define any mechanism for input of data or output of results. But it was mathematically elegant.

A mathematicians toy?

Algol was too elegant to be used simply as a language appearing in mathematics journals. So, people started to try to implement it as a programming language. A few of these implementations were successful and Algol was used a bit as a programming language in the '60s. Although not wildly successful as a practical language, Algol did serve as a starting point for most subsequent developments of new programming languages.

Stack based function calls and recursion

Algol couldn't use the simple mechanisms that FORTRAN could employ to organize data storage and function calls. Instead, it had to have a more complex mechanism based on a "stack" storage structure. This was because Algol had to handle "recursive" functions; the mathematicians insisted on having these. At the time, this need for a stack all seemed obscure, unnecessarily complex and difficult; but now it is the standard. All modern languages use this approach rather than the older simpler static scheme of FORTRAN.

Recursive functions?
!

You are unlikely to have had to deal with many recursive functions in your earlier studies of maths. (quite possibly, you'll not have encountered any recursive functions). There are relatively few simple examples of recursive functions (and most of these are unrepresentative because they relate to problems where there are alternative iterative solutions). Real recursive functions turn up in obscure areas of mathematics and, somewhat oddly, in the context of tasks like finding a route through a maze or a way around a network of nodes and edges. Many of the more elaborate data structures now used in computing actually represent things like networks, and so you will be meeting some more realistic recursive functions when you start to write programs that use such structures.

A problem using recursion

The following is a slightly contrived example of a problem that can be solved using a recursive function.

You have to print the value of a positive integer that represents a result from your program. Unfortunately you are using a language whose inventor failed to provide any output mechanism other than the function "putchar(<character>)" which prints a character (e.g. the character 'A', or the digit character '3' or whatever else you want). So, you are going to have to find a way of generating the digit characters that represent your number and getting these printed using the provided putchar() routine. For example, if your number is three hundred and twelve, you'll need to call putchar() to print a digit '3', then another call to print digit '1', and finally a call to get digit 2.

How could you get the digits?

Getting the low order digit is not hard. The circuits in the CPU that do division can also give you the remainder value from a division:

```
e.g.      78   divided by ten   leaves remainder 8
          312  divided by ten   leaves remainder 2
```

Programming languages will have "operators" that allow you to ask for this remainder. e.g. in C (and C++) you could have

```
remainder = number % 10;
```

Doing the remainder (%) operation gives you a numeric value in the range 0..9. You still have to get a printable digit character --- '0', '1', '2', ..., '8', '9'. There would be many ways of doing this (and they aren't really important in this example). One way might be to use your remainder value to select the right digit from an array of characters (char is C's abbreviation for character)...

```
char  digits[] = { '0', '1', '2', '3', '4', '5', '6',
                  '7', '8', '9' };
...
remainder = number % 10;
char  correctdigit = digits[remainder];
putchar(correctdigit);
```

So, you know that it is easy to print the last digit of the number.

You could deal with printing a complete number by getting someone else to print all the leading digits, then you could print the last digit.

What would the leading digits be? Well, they will be the digits that represent your number divided by ten .e.g. you get given the value seven thousand, eight hundred and twenty four to print, you can do a remainder and see that the last digit should be a '4' but you'll need someone to print the rest – the digits that represent the number seven hundred and eight two.

That solves it! We can get the problem solved by a employing bureaucracy of people each of whom follows the same data processing rules, see Figure 4.4.

In an Algol-based language you can define recursive routine that simulate the workings of such a bureaucracy (the following is simply an Algol-ish pseudocode, the statements don't exactly match any real language):

```
recursive routine PrintPosNumber(integer Number)
begin
  if(Number<10) then PrintDigit(Number);
  else begin
    integer quotient;
    integer remainder;
    quotient = Number / 10;
    remainder = Number % 10;
    PrintPosNumber(quotient);
    PrintDigit(remainder);
  end;
end;
```

A routine has a name, e.g. PrintPosNumber (for print positive number), and an "argument list" (a specification of the data it needs to be given).

```
recursive routine PrintPosNumber(integer Number)
```

Here, only one integer data value has to be given to the function.

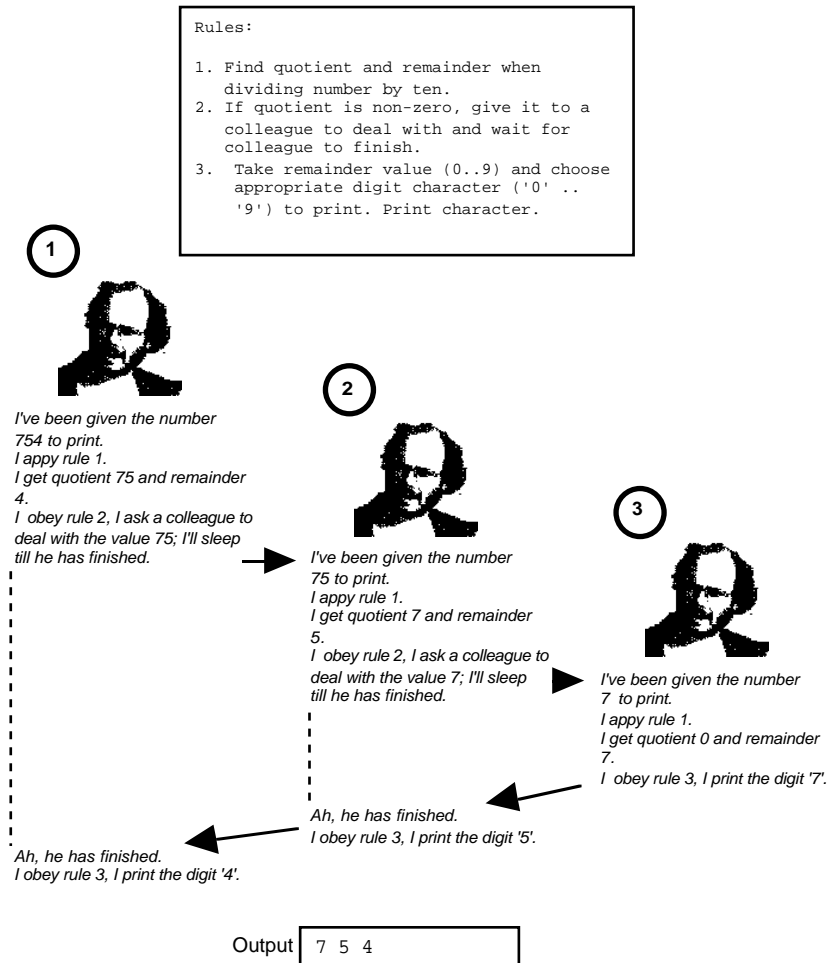


Figure 4.4 A "recursive" problem solving process.

The code checks for the easy case first (always a good idea for recursive routines):

```
begin
  if(Number<10) then PrintDigit(Number);
```

If the number is less than ten, it can be converted directly to a digit character and this character can be output using the given `putchar()` routine. This would be done by an auxiliary routine, "PrintDigit." (Usually, it is better to have lots of little routines rather than one complex routine that does all the work.)

If the number exceeds ten, it has to be broken down to get a quotient and a remainder.

```
integer quotient;
integer remainder;
```



```
quotient = Number / 10;
remainder = Number % 10;
```

The quotient has to be passed in a recursive call for its digits must be printed first.

```
PrintPosNumber(quotient);
```

Once the quotient has been dealt with, this routine can print the digit corresponding to the remainder.

```
PrintDigit(remainder);
end;
```

At the point where the first character is printed, there will be many versions of PrintPosNumber running. In fact, there will be one PrintPosNumber for every digit in the decimal representation of a number. A five digit number, like sixteen thousand one hundred and nine will need five PrintPosNumbers working; an eleven digit number would need eleven versions. Each version has to keep track of its own "Number", "Quotient" and "Remainder" and needs some place in memory to store these values. This is where a "stack" gets involved.

A compiler for Algol (or any similar language) generates code for (recursive) routines that allows them to claim space from a "stack" as they start and then to release this space when they finish. The stack itself is simply a big chunk of a program's main memory that has been reserved for this purpose.

Stack

Routines reserve space for their input arguments ("Number" in the example) and local variables ("Quotient" and "Remainder"). A little more space is needed for 'housekeeping' purposes (like remembering a return address when another routine has to be called). The stack space reserved by a routine is called a "stack frame". Figure 4.5 illustrates the frames on the stack at the moment where the first digit is about to be printed (same example data as Figure 4.4).

Stack frames for arguments and local variables

Although invented for the somewhat atypical case of recursive routines, the stack based scheme for allocating memory proved to be much more useful than expected.

Stacks have advantages over "static" allocation

FORTRAN's static allocation scheme for arranging space for variables tends to waste space. Often a routine will need lots of local variables, but these only really need to have space allocated while that routine is being executed. If space is allocated statically as in a simple FORTRAN system, then its reserved for the entire time that the program is running. If all routines were to get their space allocated automatically on the stack when they started, and they freed the stack space automatically as they finished, then the space need for a routine's local variables would only be reserved while the routine was executing.

Most current languages copy Algol and make use of a stack for organizing the memory needs of routines. The stack is also used for the housekeeping work of keeping track of the sequence of subroutine calls, return addresses etc. This "system's stack" is all handled by runtime support routines and extra bits of code inserted by the compiler. Programmers don't really notice its being there, it is all automatic. (In fact, stack based storage is often called "automatic storage".)

"Automatic" storage

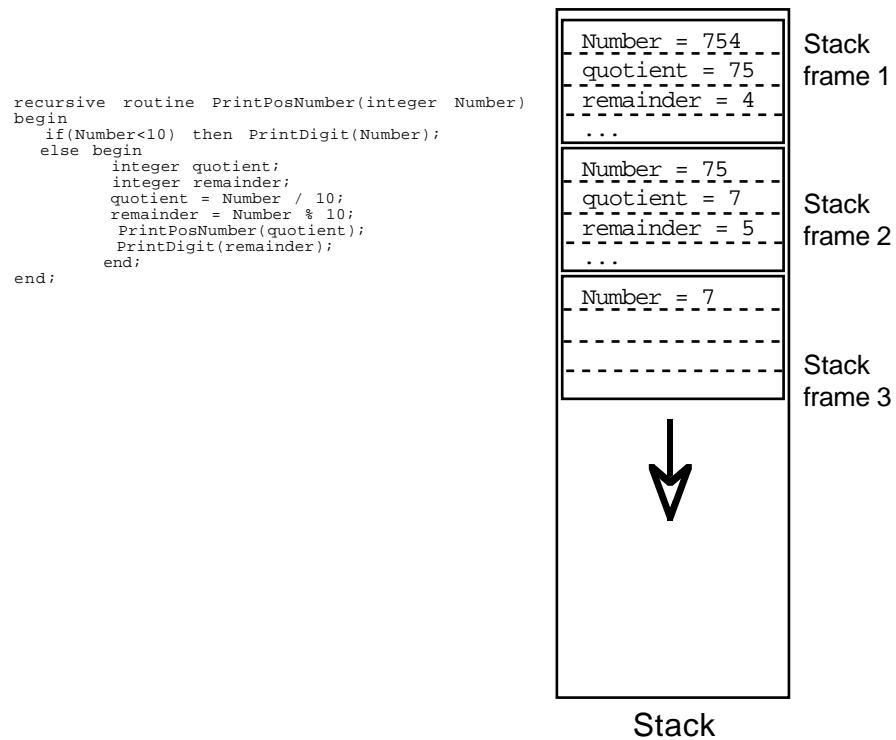


Figure 4.5 Stack during execution of recursive routine.

Nested structure

A FORTRAN program was always thought of as being made up from separately compiled files that got linked together. As noted earlier (section 4.4), this scheme has advantages like making it easy to develop subroutine libraries, but also disadvantages like the compiler being unable to check whether the calls to other routines were correct with respect to arguments etc.

An Algol program was a single composite entity. It contained inside itself a "main" routine and all the other functions and subroutines ("procedures") that it needed. With an Algol program you would have something like the following:

```

program demo;           Program header and
var                    declaration of shared
  integer count;        data variables
...
...
procedure HeatFlow(...);
begin                  Definitions of subroutines
  ...                  and functions
end;

```

```

    procedure PrintResults;
    begin
        ...
    end;

begin
    Main program
    ...
    HeatFlow(...);
    ...
    PrintResults;
end.

```

With only a single program file, the compilation process was simpler than that for FORTRAN. Figure 4.6 illustrates the approach.

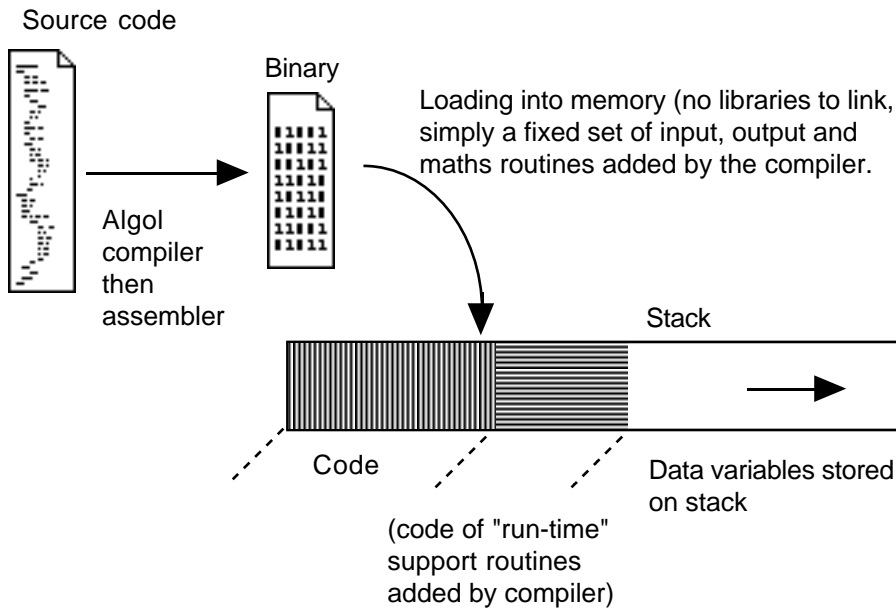


Figure 4.6 Algol style compilation and loading processes.

This organization of a program made it possible for the compiler to do more checking of the code. A procedure declaration or function declaration had to specify details of the data values that that procedure would need to be given to work with (like the integer `Number` in the `PrintPosNumber()` function above). Later in the code, there would be calls to that procedure. The compiler could check that these calls provided appropriate data values.

Compile time checks

Nesting could be taken further. A procedure declared within the main program could have other procedures and functions declared within it. These enclosed functions would be auxiliary routines used only within the procedure wherein they were defined. Usually there were limits on the extent of nesting, you couldn't define a procedure inside a function inside a procedure inside another function

This nesting scheme was meant to encourage and help programmers break down complex processing steps into lots of small functions that could be grouped together. However, it did make things more difficult for compiler writers to handle the source code and some aspects of the scheme forced the generation of less efficient machine code. Although some of the Algol family languages continue using nested declarations, other descendants of Algol (e.g. C and C++) have abandoned nested procedures.

Use of language

Never a great success as a practical programming language, Algol's importance is that it was the starting point from which most of the modern programming languages have evolved.

4.9 THE "ALGOL FAMILY"

4.9.1 AlgolW, Pascal, and Modula2

Niklaus Wirth, a Swiss computer scientist, is responsible for these members of the Algol family.

AlgolW

Along with a number of others at Stanford, Wirth developed AlgolW about 1965. Although it was meant to become a practical programming language for writing things like operating systems, compilers, etc., AlgolW was never taken further than a first limited version. This version was well suited to teaching programming, but was only available for IBM-360 computers.

Pascal compiler and its P-code interpreter

Wirth returned to Switzerland about 1970. In Switzerland, he only had access to a CDC computer, so he couldn't use AlgolW. He revised the language and wrote a compiler that generated partially interpreted code. The interpreter for this "P-code" was easy to implement in assembly language and versions were prepared for many different computers. This meant that the new language, Pascal, was soon widely available. But it was a fairly limited implementation and its main use was still teaching.

Modula2 was Wirth's third revision. It tidies up a few problem areas of Pascal and includes some minor extensions.

Although these languages have many small differences, they are basically the same, so the most commonly used, Pascal, can serve as an example.

Wirth used Algol as the base for these languages, so they have a nested structure, and they utilize stacks for organizing "automatic" memory space. Wirth omitted a number of features originally present in Algol that had proven difficult to use. He also improved the language with:

- more built in data types
- better control structures for iteration and selection
- programmer defined records

- dynamic storage structures

More built in data types

Algol had only had real and integer numbers. It is often useful to have other data types: "booleans" (variables that have the values TRUE or FALSE), characters (and groups of characters, or "strings", that can represent words or sentences), and "sets". All these were provided in AlgolW and its successors.

Boolean, character, string, and set data types

Control structures

Of course, these languages have counting loops e.g.:

```
for i:= 1 to 10 do begin
..
    stuff to be done ten times
...
end;
```

Statements defining iterative (loop) structures

But they have other loop constructs such as "while" loops ...

```
readln(value);           Gets first data value from user
while(value>0) do begin
    ...
                        Process positive data values
    ...
    readln(value);       Reads next value entered by user
end;
```

For selection, there is an "if ... then ..." statement that can be used to choose whether an action is (or is not) performed:

Selection statements

```
if(current>maxfound) then
    maxfound := current; updates 'maxfound' if necessary
```

There is an "if ... then ... else ..." statement for selecting between two alternatives:

```
if(sex = 'F') then females := females+1;
else males := males+1;
```

Such statements can be concatenated to select from among more than two alternatives:

```
if(age < 1) then infants := infants+1;
else
if(age < 16) then children := children+1;
else adults := adults+1;
```

But there is an alternative statement for dealing with multiway selection that is often more convenient e.g.:

```

      case MonthNumber of
1:      writeln("January has 31 days");
2:      writeln("February usually has 28 days");
      ...
12:     writeln("December has 31 days");
      end;

```

Records

Programmer defined "record structures"

AlgolW added record structures (inspired by the records used earlier in languages like COBOL); these records were further refined in Pascal and Modula2. So, if a programmer wanted to represent a "patient" in a program handling hospital records, it was possible to define such a record:

```

type
  patient = record
    name : array [1..30] of char;
    age : integer;
    sex : char;
    ward : integer;
    ...
  end;

```

and then have variables of type patient (and also files of patient records).

Dynamic ("heap based") data structures

Dynamic data structures

The AlgolW language group also adopted ideas from Lisp. In these languages, it is possible to create structures "dynamically" when a program finds that they are necessary (just like in Lisp where lists and other structures are built as needed).

The "heap"

Another area of memory is reserved for these dynamic structures (it is called the *heap*). Space for dynamic structures can be requested in this area. Dynamic structures remain in existence until explicitly freed (when the space they occupied should be released).

Limitations

Pascal never quite made it as a full scale programming language (and Modula2 is of quite minor importance).

"Pascal is inefficient." This was often given as a reason for not using Pascal. The first compilers for Pascal were usually written to run quickly but to generate simple non-optimized code. (The standard compiler that generated interpreted code was used to move Pascal to a new machine. Once it was running on that the new machine, the code generation parts were changed to produce real assembly

language rather than interpreted P-code.) Usually, Pascal was adopted first for teaching, and students typically spend much more time running the compiler than ever running their own programs so efficiency of programs wasn't that important. The nested procedural structure does entail some run-time overheads that reduce efficiency, but this is not that major a problem. It is possible to get Pascal compilers that generate efficient code; it is just that most compilers don't try.

A more serious problem with Pascal related to the issue of separate compilation. The standard Algol structure of a single program with no separately compiled parts is far too restrictive. In fact it had to be abandoned. Practical programming projects need to make use of subroutine libraries and need to have schemes for separate compilation of program parts followed by linking (as in FORTRAN). Most implementations of Pascal allowed for separate compilation, but this involved extensions to the language that were non-standard (the standard has finally been revised to accommodate separate compilation). Compilers from different suppliers tended to provide such extensions in slightly different ways, so limiting the interoperability of Pascal code.

If you have separately compiled parts, then you have the problem of the compiler not being able to check consistency of these parts. This problem was solved in the extended Pascal dialects (and Modula2) using an approach developed earlier for BCPL and C (see section 4.9.3).

Another common complaint about Pascal was that "*Pascal makes you say 'please'.*" The Pascal language has rather strict rules on how you use data and limits on the extent to which you can do things like find the addresses of data elements. These rules and limits are very helpful to beginners who tend to misuse data and to do things with addresses that really shouldn't be done! But in more advanced work, programmers often need to build up complex network structures that involve address data. In Pascal, code becomes a little verbose because everywhere data are used in a slightly non-standard way the programmer has to indicate that this was intentional ("saying 'please' to the compiler"). Many programmers disliked this style.

4.9.2 ADA

ADA is a rather specialized descendant of Pascal.

The ADA language was designed for the US Department of Defence for "embedded systems" (control programs in everything: fighter aircraft, radar units, automated warehouses, ...).

The style of the language is vaguely reminiscent of Pascal, and like Pascal the compiler incorporates lots of analysis and checking to try to detect errors in a source program. But ADA adds much to Pascal: support for separate compilation, features to allow for multiprocessors, communications, mechanisms for handling run-time errors. The extensions are so numerous as to make the ADA language one of the larger, more complex languages now in use.

Defence related projects are often required to use ADA, but the language is otherwise not that popular.

4.9.3 BCPL, C, AND C++

Way back around 1963, an attempt was made to define a Combined Programming Language (CPL) that would somehow combine the "best" features of an algorithmic language like Algol and a data processing language like COBOL. This exercise proved to be too ambitious and was abandoned. (Later in the 1960s, IBM sponsored another attempt along these lines, combining features from FORTRAN, Algol, and COBOL; this project was completed and resulted in the language PL1.)

BCPL – a limited language designed for writing software systems

Martin Richards, a junior in the CPL project, invented BCPL (Basic CPL) a much simpler language with a much more specific aim. BCPL wasn't intended to be good at everything. Its aim was simply to be a good language for writing system's software (things like compilers, editors, components of operating systems etc). BCPL was moderately successful in this role. It did have limitations (e.g. no real numbers, but why would you want real numbers if you are writing something like a "device driver" (code to control a peripheral device)?).

The BCPL compiler was written largely in BCPL (with just a bit of assembly language code that could be easily changed). It translated BCPL statements into instruction sequences using little templates that were defined in terms of the instructions available on a particular kind of computer. If you changed these templates a BCPL compiler on one machine could generate code for a different kind of computer. This made the BCPL language easy to transport and quite a number of organizations had people using BCPL in the late 1960s.

The language "B"

Kernighan and Ritchie at ATT laboratories had a BCPL compiler. They decided to rework the language, removing limitations, adding a few features, while keeping the main idea of a language that would be good for writing system's software. Their first attempt (the language "B") was simply a minor reworking, simplification, and shortening of BCPL. Their second version was "C".

Caution, C, handle with care

Just a little language to call their own. Something suited to a couple of gifted, experienced programmers working on their own high tech projects. But C escaped. The world's first computer virus. C got out from the ATT laboratories and infected machines all over the world before it was ever made safe for the average programmer.

C was designed to be a language that would be suitable for writing system's software, like the core parts of an operating system. The code generated by the compiler had to be very efficient if the language was to be used in this way. Anything a compiler would have difficulties with was dropped. Consequently, in some respects the language is simpler than other Algol family languages that have retained the relatively complex nested program structures.

Down to the hardware level!

If C was to be used for writing things like "device driver code" (the code that actually interacts with the peripheral controllers), then it had to allow the programmer get down the hardware level and manipulate bits in specific registers and in particular memory addresses. This gives the programmer considerable power, and lots of responsibility.

The programmer is always right

The most dangerous design aspect of C was the requirement that the compiler should always assume that the programmer was always right. Even if a compiler could detect a probable error in a program, it should just go ahead and generate the

code. After all, the programmer was going to be one of the ATT team (Thompson, Ritchie, or Kernighan) and they'd know what they were doing. They wouldn't need a compiler to second guess them or try to hold their hand and protect them..

All this makes C a rather dangerous tool for the average programmer. (If the Pascal compiler keeps making you say "please", a C compiler keeps making you say "sorry".) The power and the danger have attracted programmers and C is now one of the three main languages (C for programmers, FORTRAN for engineers, COBOL for accountants).

Actually, the ANSI standard for C put a lot of compile time checks into the language, so taming it a little.

C program structure

C programs are typically built from multiple, separately compilable source files, and functions taken from libraries. The program generation process is illustrated in Figure 4.7.

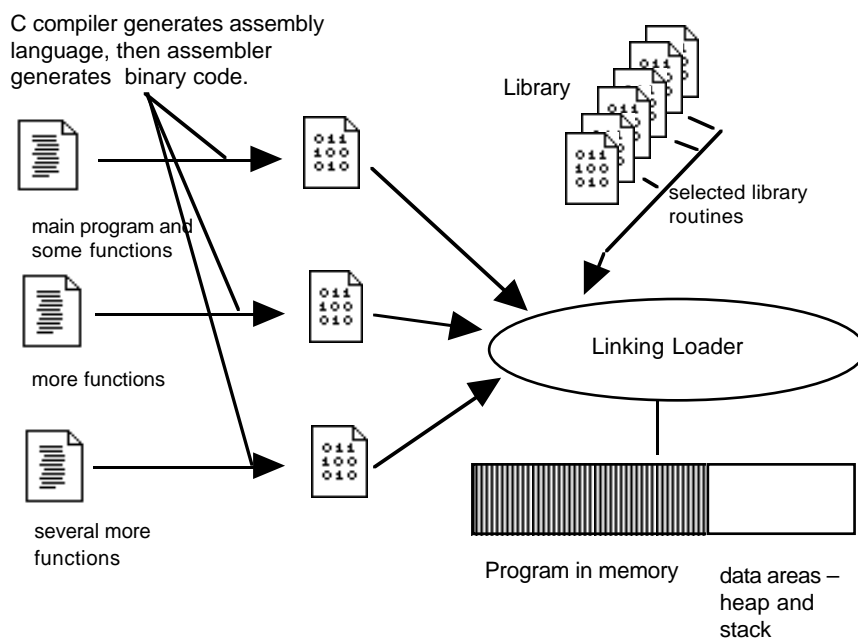


Figure 4.7 Compiling, assembling, and link-loading a C program.

This structure gives C the same advantages as FORTRAN

- it is easy to build up libraries of functions that can be used in many programs;
- a large program can be split into small parts with different programmers working on each part;

- the editing/compilation process tends to be efficient, you only need to recompile the file(s) containing code that has changed.

C provides a mechanism that a programmer can use to get the compiler to check the consistency of separately compiled parts of a program (you aren't forced to use this mechanism all the time, a C compiler will allow you to make as many mistakes as you want).

Header files

The source code for C routines is stored in files, usually these will have names that end in .c; e.g. File1.c, and File2.c. A second file, a "header" file, may be associated with each .c file; e.g. File2.c can have an associated header file File2.h. A header file contains information describing the routines (and any shareable data structures) that are defined in the associated .c file. Libraries, that contain very large numbers of functions, will have associated header files that describe their contents.

Using header information for compile time checks

Suppose the main program in File1.c uses functions from File2.c and you want the compiler to check that it is using them correctly. You can put a line into File1.c that tells the C compiler to read File2.h so that it knows what is defined in File2.c.

#include

This instruction to the compiler would say something like:

```
#include "File2.h"
```

and would appear on a line near the start of File1.c.

When it encounters a #include statement, a C compiler reads the information in the file specified and records details for later reference. If it later finds references to things defined in File2.c, it uses the information it saved to check that routines are being used correctly.

"Standard" header files

You will get used to seeing #include statements at the start of most files of C code. Usually, these will be 'including' header files that describe things in the standard libraries of input/ output routines or the mathematical functions.

A C source file will contain a number of routines ("functions" that compute values, and "procedures" that perform tasks). There may also be some definitions of data variables that can be shared by many routines (as in FORTRAN). One of the files forming a program has to contain a function called "main". This is the main program that contains the calls to all the other routines; the compiler arranges for this routine to be started when the program gets loaded into memory.

```
#include <stdlib.h>      Include standard header files
...

int SharedCounter;      Define any shared data
...

void Sort(int a[], int n)
{
    ...                Define functions
}

int LoadData()
```

```

{
    ...
}

int main()
{
    ...
}

```

Provide a main program

Each individual routine (function or procedure) is built up from sequences of statements: assignment statements for calculations, conditional statements for selecting among choices, iterative statements for loops, and "subroutine call" statements that invoke other routines.

All the modern (post-AlgolW) languages have essentially the same kinds of statement. There are differences in layout and in the keywords used, but the statements are essentially the same. Those in C are typically a little more concise than the equivalent statements in other languages like Pascal.

So, C has "for" loop and "while" loops; it has "if (...) ..." statements, and "if (...) ... else ..." statements; it has a multiway selection statement ("switch() { ... }"). Once you've learnt any one of these Algol family languages, you've really learnt them all!

Statements

The data types are again similar. There are the standard integer and real numbers, and characters. Unlike Pascal, C doesn't define any standard "boolean" or "set" data types (but it is easy to define your own if you need these data types).

Standard data types

C has record structure definitions similar to those in Pascal (maybe slightly more flexible than those in Pascal). Of course a different syntax is used to define record structures.

Record structures

C allows the programmer to choose how storage space is allocated to variables.

One can have static data like FORTRAN. Space for static data is organized by the compiler and linker-loader and is allocated before the program starts and remains allocated the entire time that the program is running.

Static variables

As an Algol-family language, C uses stack-based "automatic" storage for most variables.

Automatic variables

C also has dynamic storage with the program requesting space for structures that are built as the program runs (and freeing the space when the structures are no longer necessary).

Dynamic, heap-based, variables

In your first programming subject, you won't bother that much about the different kinds of storage, you will mainly use automatic storage. It is in your second programming subject that you will need to use all the different kinds of storage.

C++

You will actually be learning C++ rather than standard C.

C dates from around about 1970. C++ started as a dialect of C around about 1980, it has been revised twice since then. But they are still very similar languages (in fact, a correct C program should be acceptable to a C++ compiler).

The C++ language aimed to achieve three things:

- to be a better C
- to support "abstract data types"
- to permit the use of a programming technique known as "object oriented programming" (OOP).

A better C? C++ was designed to permit more compile time checking and also to offer alternatives to various features of C that were known to be common sources of programming errors.

"Abstract data types" – these will be the main topic of your second programming course, they are a kind of elaboration of the idea of a record structure.

OOP –see next section of Simula, Smalltalk, and Eiffel.

4.9.4 SIMULA, SMALLTALK, AND EIFFEL

These are the principal specialized "Object Oriented" languages.

Simula Simula was developed in the mid-1960s as a language for simulations (simulations of anything you wanted: modelling aircraft movements at an airport, modelling the spread of a disease in a population of individuals, modelling the activities in an automated car wash, ...)

Objects Simula was based on Algol-60 but added a variety of constructs that were needed for simulation work. Essentially, it allowed the programmer to create in the computer a set of "objects" (each of which owned some resources and had specified behaviours) that modelled things in the real world. Once the objects had been created, the Simula run-time system could mimic the passage of time and could allow the programmer to track interactions among the objects.

Smalltalk The Smalltalk language was developed by the very innovative research group at Xerox's Palo Alto Research Centre (the same group as invented the prototype for the "Macintosh/Windows" OS and interface). Smalltalk offers a different way of thinking about programming problems.

Reusable component libraries Usually, each problem is treated as if it were totally new. The problem gets analyzed, broken down into subtasks, and then new code is written to handle each of these subtasks. Smalltalk encourages an alternative view; instead of writing new special purpose code, try to find a way of building up a solution to a problem by combining reusable components.

The reusable components are Smalltalk objects. A Smalltalk system provides hundreds of different kinds (classes) of "off the shelf " reusable components. Actually, Smalltalk is an interpretive system (a bit like Lisp) and the language is not strictly in the Algol family.

Eiffel In some respects, Eiffel is the best programming language currently available. It takes advantage of the experience gained with earlier languages like Simula, Pascal, Smalltalk, ADA and others.

It is a compiled language (so Eiffel programs are much more efficient than interpreted Smalltalk programs). The basic idea is the same as Smalltalk, i.e. the best way to construct programs is to build them out of reusable objects.

Although in many respects very good, Eiffel is restrictive. It enforces the use of an Object Oriented (OO) style. You have to learn several styles, not just OO. For this reason, you are learning C++ because it supports conventional procedural style as well as OO.

EXERCISES

- 1 Began programming on the electromechanical computing devices used in the 1940s by the US Navy to generate gunnery tables. Reputedly invented the term "bug" for a programming error after having had to remove a crushed moth that was jamming one of the relays in this computer. Rose to be a high ranking US Navy officer. Prime mover in the standardisation efforts that lead to the development of the COBOL programming language.

Identify this person and write a more complete biography.

- 2 Implemented one of the first (if not the very first) Algol compiler. Created the first operating system that was designed on a layered model with a kernel providing low level services, and surrounding layers that added functionality. Proponent of structured programming techniques and critic of earlier coding styles with their criss-crossing flows of control induced by indiscriminate use of "GOTO" statements. Tried to convince programmers that they needed a little discipline.

Identify this person and write a more complete biography.

5 C++ development environment

5.1 INTEGRATED DEVELOPMENT ENVIRONMENT

Usually it is necessary to learn how to use:

- a "command language" for a computer system, (e.g. Unix shell or DOS commands)
- an editor (specialised word processor)
- a "make" system that organizes the compilation of groups of files
- a compiler
- a linking-loader and its associated libraries.

Fortunately, if you are working on a personal computer (Macintosh or PC) you will usually be able to use an "Integrated Development Environment" (IDE).

An IDE packages all the components noted above and makes them available through some simple to use visual interface.

On a Intel 486-, or Pentium- based PC system, you will be using either Borland's C++ environment or Microsoft's C++. On a Macintosh system, you will probably be using Symantec C++. These systems are fairly similar.

They employ a variety of different windows on the screen to present information about a program (usually termed a "project") that you are developing. At least one of these windows will be an editing window where you can change the text of the source code of the program. Another window will display some kind of summary that specifies which files (and, possibly, libraries) are used to form a program. Figure 5.1 illustrates the arrangement with Symantec 8 for the Power PC. The editing window is on the left, the project window is to the right.

Project

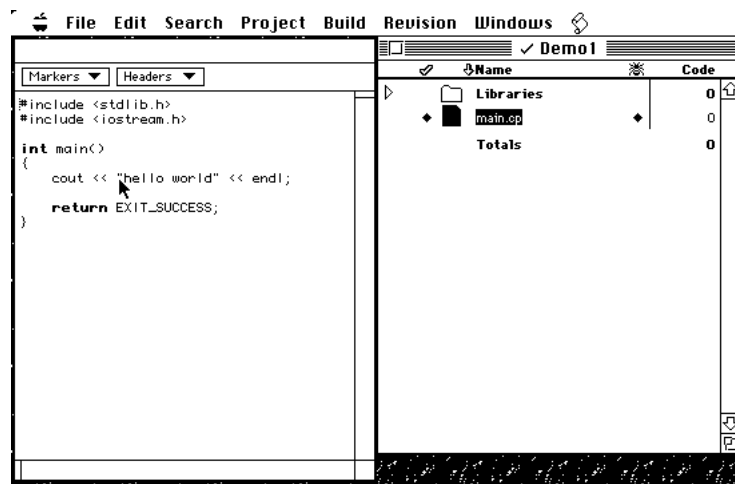


Figure 5.1 Illustration of typical editing and project windows of an example integrated development environment.

The figure illustrates the situation when the Symantec system has been asked to create a new project with the options that it be "ANSI C++ with iostreams". Such options identify things such as which input output library is to be used. The environment generates a tiny fragment of C++ code to get you started; this fragment is shown in the editing window:

```
#include <stdlib.h>
#include <iostream.h>

int main()
{
    cout << "hello world" << endl;

    return EXIT_SUCCESS;
}
```

The two `#include` lines inform the compiler about the standard libraries that this program is to use. The code from "int main" to the final "`}`" is a skeletal main program. (It is a real working program; if you compiled and ran it then you would get the message "hello world" displayed.)

The project window lists the components of the program. Here, there is a folder containing libraries and a single program file "main.cp".

The Borland environment would generate something very similar. One of the few noticeable differences would be that instead of being separate windows on the desktop, the project and editing windows would both be enclosed as subwindows of a single large window.

Menu commands

Like spreadsheets, paint programs, and word processor programs, most of the operations of an IDE are controlled through menu commands. There are a number of separate menus:

- File: things like saving to file, opening another file, printing etc.
- Edit: cut-copy-paste editing commands etc.
- Search: finding words, go to line etc
- Project: things like "Run" (run my program!)
- Source: options like "check" and "compile"

The various environments have similar menu options though these are named and organized slightly differently.

5.2 C++ INPUT AND OUTPUT

Because any interesting program is going to have to have at least some output (and usually some input), you have to learn a little about the input and output facilities before you can do anything.

In some languages (FORTRAN, Pascal, etc), the input and output routines are fixed in the language definition. C and C++ are more flexible. These languages assume only that the operating system can provide some primitive 'read' and 'write' functions that may be used to get bytes of data into or out from a program. More useful sets of i/o routines are then provided by libraries. Routines in these libraries will call the 'read' and 'write' routines but they will do a lot of additional work (e.g. converting sequences of digits into the appropriate bit pattern to represent a number).

Input/output capabilities defined by libraries

Most C programs use an i/o library called `stdio` (for standard i/o). This library can be used in C++; but more often, C++ programs make use of an alternative library called `iostream` (i/o stream library).

stdio and iostream libraries

The i/o stream library (`iostream`) makes use of some of the "object oriented" features of C++. It uses "stream objects" to handle i/o.

Stream objects

Now in general an object is something that "owns a resource and provides services related to that resource". A "stream object" owns a "stream", either an output stream or an input stream. An output stream is something that takes character data and gets them to an output device. An input stream gets data bytes from some input device and routes them to a program. (A good way of thinking of them is as kinds of software analogues of hardware peripheral device controllers.)

An `ostream` object owns an output stream. The "services" an `ostream` object provides include the following:

ostream objects

- an `ostream` object will take data from an integer variable (i.e. a bit pattern!) and convert it to digits (and \pm sign if needed) and send these through its output stream to an output device;
- similarly, an `ostream` object can take data from a "real number" variable and convert to a character sequence with sign character, digits, and decimal point (e.g. 3.142, -999.9, or it may use scientific notation and produce a character sequence like 1.70245E+43);

- an `ostream` object can take a message string (some character data) and copy these characters to the output stream;
- an `ostream` object can accept directions saying how many digits to show for a real number, how much space on a line to use, etc.

istream objects An `istream` object owns an input stream. The "services" an `istream` object provides include the following ...

- an `istream` object can be asked to find a value for an integer variable, it will check the input stream, miss out any blank space, read in a series of digits and work out the value of the number which it will then put into the variable specified;
- similarly, an `istream` object can be asked to get a "real number" variable, in this case it will look for input patterns like 2.5, -7.9, or 0.6E-20 and sort them out to get the value for the number;
- an `istream` object can be asked to read in a single character or a complete multicharacter message.

Error handling with input and output It is unusual for an `ostream` object not to be able to deal with a request from a program. Something serious would have had to have gone wrong (e.g. the output is supposed to be being saved on a floppy disk and the disk is full). It is fairly common for an `istream` object to encounter problems. For example, the program might have asked for an integer value to be input, but when the `istream` object looks in the input stream it might not find any digits, instead finding some silly input entered by the user (e.g. "Hello program").

Response to bad input data An `istream` object can't convert "Hello program" into an integer value! So, it doesn't try. It leaves the bad data waiting in the input stream. It puts 0 (zero) in the variable for which it was asked to find a value. It records the fact that it failed to achieve the last thing it was asked to do. A program can ask a stream object whether it succeeded or failed.

"end of file" condition Another problem that an `istream` object might encounter is "end of file". Your program might be reading data from a file, and have some loop that says something like "istream object get the next integer from the file". But there may not be any more data in the file! In this case, the `istream` object would again put the value 0 in the variable, and record that it had failed because of end-of-file. A program can ask an `istream` object whether it has reached the end of its input file.

Simple use of streams The first few programming exercises that you do will involve the simplest requests to `iostream` objects ... "Hey input stream object, read an integer for me.", "Hey output stream object, print this real number." The other service requests ("Did that i/o operation work?", "Are we at the end of the input file?", "Set the number precision for printing reals to 3 digits", ...) will be introduced as needed.

Standard streams A program can use quite a large number of stream objects (the operating system may set a limit on the number of streams used simultaneously, 10, 16, 200 etc depends on the system). Streams can be attached to files so that you can read from a data file and write to an output file.

The `iostream` library sets up three standard streams:

- `cin` standard input stream (reads data typed at keyboard)

- `cout` prints results in an "output window"
- `cerr` prints error messages in an "output window" (often the same window as used by `cout`); in principle, this allows the programmer to create error reports that are separate from the main output.

You don't have to 'declare' these streams, if your program says it wants to use the `iostream` library then the necessary declarations get included. You can create `iostream` objects that are attached to data files. Initially though, you'll be using just the standard `iostream` objects: `cout` for output and `cin` for input.

Requests for output to `cout` look like the following:

Output via cout

```
int    aCounter;
double aVal;
char Message[] = "The results are: ";
...
cout << Message;
cout << aCounter;
cout << " and ";
cout << aVal;
...
```

The code fragment starts with the declarations of some variables, just so we have something with values to output.:

Explanation of code fragment

```
int    aCounter;
double aVal;
char Message[] = "The results are: ";
```

The declaration `int aCounter;` defines `aCounter` as a variable that will hold an integer value. (The guys who invented C didn't like typing, so they made up abbreviations for everything, `int` is an abbreviation for `integer`.) The declaration `double aVal;` specifies that there is to be a variable called `aVal` that will hold a double precision real number. (C and C++ have two representations for real numbers – `float` and `double`. `double` allows for greater accuracy.) The declaration `"char Message ..."` is a bit more complex; all that is really happening here is that the name `Message` is being associated with the text `The results...`

In this code fragment, and other similar fragments, an ellipsis (...) is used to indicate that some code has been omitted. Just assume that there are some lines of code here that calculate the values that are to go in `aVal` etc.

```
cout << Message;
cout << aCounter;
cout << " and ";
cout << aVal;
```

These are the requests to `cout` asking it to output some values. The general form of a request is:

```
cout << some value
```

(which you read as "cout takes from some value"). The first request is essential "Please cout print the text known by the name Message." Similarly, the second request is saying: "Please cout print the integer value held in variable aCounter." The third request, `cout << " and "` simply requests output of the given text. (Text strings for output don't all have to be defined and given names as was done with Message. Instead, text messages can simply appear in requests to `cout` like this.) The final request is: "Please cout print the double value held in variable aVal."

Abbreviations C/C++ programmers don't like typing much, so naturally they prefer abbreviated forms. Those output statements could have been concatenated together. Instead of

```
cout << Message;
cout << aCounter;
cout << " and ";
cout << aVal;
```

one can have

```
cout << Message << aCounter << " and " << aVal;
```

Basic formatting of output lines

`cout` will keep appending output to the same output line until you tell it to start a new line. How do you tell it to start a new line?

Well, this is a bit system dependent. Basically, you have to include one or more special characters in the output stream; but these characters differ depending on whether your program is running on Unix, or on DOS, or on ... To avoid such problems, the `iostream` library defines a 'thing' that knows what system is being used and which sends the appropriate character sequence (don't bother about what this 'thing' really is, that is something that only the guys who wrote the `iostream` library need to know).

endl This 'thing' is called "endl" and you can include it in your requests to `cout`. So, if you had wanted the message text on one line, and the two numbers on the next line (and nothing else appended to that second line) you could have used the following request to `cout`:

```
cout << Message << endl
    << aCounter << " and " << aVal << endl;
```

(You can split a statement over more than one line; but be careful as it is easy to make mistakes when you do such things. Use indentation to try to make it clear that it is a single statement. Statements end with semicolon characters.)

Input from cin

Input is similar in style to output except that it uses ">>" (the 'get from operator') with `cin` instead of "<<" (the 'put operator') and `cout`.

If you want to read two integers from `cin`, you can write

```
int aNum1, aNum2;
...
```

```
cin >> aNum1;  
cin >> aNum2;
```

or

```
cin >> aNum1 >> aNum2;
```

In general you have

```
cin >> some variable
```

which is read as "*cin gives to some variable*".

5.3 A SIMPLE EXAMPLE PROGRAM IN C++

This example is intended to illustrate the processes involved in entering a program, getting it checked, correcting typing errors, compiling the program and running it. The program is very simple; it reads two integer values and prints the quotient and remainder. The coding will be treated pretty informally, we won't bother with all the rules that specify what names are allowed for variables, where variables get defined, what forms of punctuation are needed in statements. These issues are explored after this fairly informal example.

5.3.1 Design

You never start coding until you have completed a design for your program!

But there is not much to design here.

What data will we need?

Seems like four integer variables, two for data given as input, one for quotient and the other for a remainder.

Remember there are different kinds of integers, short integers and long integers. We had better use long integers so permitting the user to have numbers up to ± 2000 Million.

Program organization?

Main program could print a prompt message, read the data, do the calculations and print the results – a nice simple sequence of instructions.

"Pseudo-code" outline

So the program will be something like:

```
define the four integer variables;
get cout to print a prompt, e.g. "Enter the data values"

get cin to read the two input values

calculate the quotient, using the '/' divide operator
calculate the remainder, using the '%' operator

get cout to print the two results
```

This will be our "main" program. We have to have a routine called `main` and since this is the only routine we've got to write it had better be called `main`.

Here the operations of the routine could be expressed using English sentences. There are more elaborate notations and "pseudo-codes" that may get used to specify how a routine works. English phrases or sentences, like those just used, will suffice for simple programs. You should always complete an outline in some form of pseudo code before you start implementation.

Test data

If you intend to develop a program, you must also create a set of test data. You may have to devise several sets of test data so that you can check all options in a program that has lots of conditional selection statements. In more sophisticated development environments like Unix, there are special software tools that you can use to verify that you have tested all parts of your code. In an example like this, you would need simple test data where you can do the calculations and so check the numerical results of the program. For example, you know that seven divided by three gives a quotient of two and a remainder of one; the data set 7 and 3 would suffice for a simple first test of the program.

5.3.2 Implementation

#including headers for libraries

Our program is going to need to use the `iostream` library. This better be stated in the file that contains our main program (so allowing the compiler to read the "header" file for the `iostream` library and then, subsequently, check that our code using the `iostream` facilities correctly).

The file containing the program had better start with the line

```
#include <iostream.h>
```

This line may already be in the "main.cp" file provided by the IDE.

main() routine

In some environments (e.g. Unix) the `main()` routine of a C/C++ program is expected to be a function that returns an integer value. This is because these environments allow "scripting". "Scripts" are "programs" written in the system's job control language that do things like specifying that one program is to run, then the output files it generated are to be input to a second program with the output of this second program being used as input to a third. These scripts need to know if something has gone wrong (if the first program didn't produce any output there isn't much point starting the second and third programs).

The integer result from a program is therefore usually used as an error indicator, a zero result means no problems, a non-zero result identifies the type of problem that stopped the program.

*Integer value
returned by main()*

So, usually you will see a program's `main()` routine having a definition something like:

```
int main()
{
    ...      definitions of variables
    ...      code statements
    return 0; return statement specifying 'result' for
script
}
```

The code "`int main()`" identifies `main` as being the name of a function (indicated by the `()` parentheses), which will compute an integer value. The `main` function should then end with a "return" statement that specifies the value that has been computed and is to be returned to the caller. (Rather than "`return 0;`", the code may be "`return EXIT_SUCCESS;`". The name `EXIT_SUCCESS` will have been defined in one of the header files, e.g. `stdlib`'s header; its definition will specify its value as 0. This style with a named return value is felt to be slightly clearer than the bare "`return 0;`").

The integrated development environments, like Symantec or Borland., don't use that kind of scripting mechanism to chain programs together. So, there is no need for the `main()` routine to return any result. Instead of having the `main` routine defined as returning an integer value, it may be specified that the result part of the routine is empty or "`void`".

In these IDE environments, the definition for a `main()` routine may be something like:

```
void main()
{
    ...      definitions of variables
    ...      code statements
}
```

The keyword `void` is used in C/C++ to identify a procedure (a routine that, unlike a function, does not return any value or, if you think like a language designer, has a return value that is empty or `void`).

void

Generally, you don't have to write the outline of the `main` routine, it will be provided by the IDE which will put in a few `#include` lines for standard libraries and define either `"int main()"` or `"void main()"`. The code outline for `main()` provided by Symantec 8 was illustrated earlier in section 5.1.

Variable definitions

The four (long) integer values (two inputs and two results) are only needed in the `main` routine, so that is where they should be defined. There is no need to make them "global" (i.e. "shareable") data.

The definitions would be put at the start of the `main` routine:

```
int main()
{
    long aNum1;
    long aNum2;
    long aQuotient;
    long aRemainder;
    ...
}
```

Naturally, because C/C++ programmers, don't like unnecessary typing, there is an abbreviated form for those definitions ...

```
int main()
{
    long aNum1, aNum2, aQuotient, aRemainder;
    ...
}
```

Choice of variable names

The C/C++ languages don't have any particular naming conventions. But, you will find it useful to adopt some conventions. The use of consistent naming conventions won't make much difference until you get to write larger programs in years 2 and 3 of your course, but you'll never be able to sustain a convention unless you start with it.

Some suggestions:

- local variables and arguments for routines should have names like `aNum`, or `theQuotient`.
- shared ("global") variables should have names that start with 'g', e.g. `gZeroPt`.
- functions should have names that summarize what they do, use multiple words (run together or separated by underscore characters `_`)

There will be further suggestions later.

Variable "definitions" and "declarations"

This is a subtle point of terminology, reasonably safe to ignore for now! But, if you want to know, there is a difference between variable "definition" and "declaration". Basically, in C++ a *declaration* states that a variable or function exists, and will be defined somewhere (but not necessarily in the file where the declaration appears). A *definition* of a function is its code; a definition of a variable identifies what storage it will be allocated (and may specify an initial value). Those were definitions because they also specified implicitly the storage

that would be used for the variables (they were to be 'automatic' variables that would be stored in main's stack frame).

Sketch of code

The code consists mainly of i/o operations:

```
int main()
{
    long aNum1, aNum2, aQuotient, aRemainder;
    cout << "Enter two numbers" << endl;
    cin >> aNum1 >> aNum2;

    aQuotient = aNum1 / aNum2;
    aRemainder = aNum1 % aNum2;

    cout << "The quotient of " << aNum1 << " and "
          << aNum2 << " is " << aQuotient << endl;
    cout << "and the remainder is " << aRemainder << endl;
    return EXIT_SUCCESS;
}
```

Explanation of code fragment

```
int main()
{
    ...
}
```

The definition of the `main()` routine starts with `int main` and ends with the final `"}"`. The `{ }` brackets are delimiters for a block of code (they are equivalent to Pascal's begin and end keywords).

```
    long aNum1, aNum2, aQuotient, aRemainder;
```

Here the variables are defined; we want four long integers (long so that we can deal with numbers in the range ± 2000 million).

```
    cout << "Enter two numbers" << endl;
```

This is the first request to the `cout` object, asking it to arrange the display of the prompt message.

```
    cin >> aNum1 >> aNum2;
```

This is a request to the `cin` object, asking it to read values for two long integers. (Note, we don't ask `cin` whether it succeeded, so if we get given bad data we'll just continue regardless.)

```
    aQuotient = aNum1 / aNum2;
    aRemainder = aNum1 % aNum2;
```

These are the statements that do the calculations. In C/C++ the "/" operator calculates the quotient, the "%" operator calculates the remainder for an integer division.

```
cout << "The quotient of " << aNum1 << " and "
      << aNum2 << " is " << aQuotient << endl;
cout << "and the remainder is " << aRemainder << endl;
```

These are the requests to cout asking it to get the results printed.

```
return EXIT_SUCCESS;
```

This returns the success status value to the environment (normally ignored in an IDE).

You should always complete a sketch of the code of an assignment before you come to the laboratory and start working on the computer!

Code entry and checking

IDEs use font styles and colours to distinguish elements in the code

The code can be typed into an editor window of the IDE. The process will be fairly similar to text entry to a word processor. The IDE may change font style and or colour automatically as the text is typed. These colour/style changes are meant to highlight different program components. So, language keywords ("return, int, long, ...") may get displayed in bold; #include statements and quoted strings may appear in a different colour from normal text. (These style changes help you avoid errors. It is quite easy to forget to do something like put a closing " sign after a text string; such an error would usually cause numerous confusing error messages when you attempted to run the compiler. But such mistakes are obvious when the IDE uses visual indicators to distinguish program text, comments, strings etc.)

Save often!

You will learn by bitter experience that your computer will "crash" if you type in lots of code without saving! Use the File/Save menu option to save your work at regular intervals when you are entering text.

Compile and "Syntax check" menu options

When you have entered the code of your function you should select the "Compile" option from the appropriate menu (the compile option will appear in different menus of the different IDEs). The IDE may offer an alternative "Syntax check" option. This check option runs just the first part of the compiler and does not go on to generate binary code. It is slightly quicker to use "Syntax check" when finding errors in code that you have just entered.

#include files add to cost of compilations

You will be surprised when you compile your first twenty line program. The compiler will report that it has processed something between 1000 and 1500 lines of code. All those extra lines are the #included header files!

Compilation errors

The compiler will identify all places where your code violates the "syntax rules" of the language. For example, the program was entered with the calculation steps given as:

```
aQuotient = aNum1 / aNum2
```

```
aRemainder = aNum1 % aNum2;
```

(The code is incorrect. The statement setting a value in `aQuotient` doesn't end in a semicolon. The compiler assumes that the statement is meant to continue on the next line. But then it finds that it is dealing with two variable names, `aNum2` and `aRemainder`, without any operator between them. Such a construct is illegal.) When this code was compiled, the Symantec compiler generated the report

```
File "main.cp"; Line 11
Error:      ';' expected
```

The IDE environments all use the same approach when reporting the compilation errors. Each error report actually consists of commands that the IDE can itself interpret. Thus, the phrase `File "main.cp";` can be interpreted by Symantec's IDE as a command to open an editing window and display the text of the file `main.cp` (if the file is already displayed in a window, then that window is brought in front of all other windows). The second part of the error report, `Line 11`, is a directive that the IDE can interpret to move the editing position to the appropriate line. The rest of the error report will be some form of comment explaining the nature of the error.

Error messages are commands that select source code lines with errors

If you do get compilation errors, you can select each error report in turn (usually, by double clicking the mouse button in the text of the error message). Then you can correct the mistake (quite often the mistake is on a line just before that identified in the report).

Generally, you should try to correct as many errors as possible before recompiling. However, some errors confuse the compiler. An initial error message may be followed by many other meaningless error messages. Fixing up the error identified in the first message of such a group will also fix all the subsequent errors.

If your code compiles successfully you can go ahead and "Build" your project. The build process essentially performs the link-loading step that gets the code of functions from the libraries and adds this code to your own code.

Building a project

During the build process, you may get a "link error". A link error occurs when your code specifies that you want to use some function that the IDE can not find in the libraries that you have listed. Such errors most often result from a typing error where you've given the wrong function name, or from your having forgotten to specify one of the libraries that you need.

Link errors

Running the program

You can simply "Run" your program or you can run it under control from the IDE's debugger. If you just run the program, it will open a new window for input and output. You enter your data and get your results. When the program terminates it will display some message (e.g. via an "Alert" dialog); when you clear ("OK") the dialog you are returned to the main part of the development environment with its editor and project windows.

Running under the control of a debugger takes a little more setting up. Eventually, the IDE will open a window showing the code of your main program with an arrow marker at the first statement. Another "Data" window will also have

Debugger

been opened. You can then control execution of your program, making it proceed one statement at a time or letting it advance through a group of statements before stopping at a "breakpoint" that you set (usually by clicking the mouse button on the line where you want your program to stop).

Stack backtrace

Figure 5.2 illustrates the windows displayed by the Symantec 8 debugger. The left pane of the window summarizes the sequence of function calls;. In this case there is no interesting information, the data simply show that the program is executing the main program and that this has been called by one of the startup routines provided by the IDE environment. (In Borland's IDE, this "stack backtrace" information is displayed in a separate window that is only shown if specifically requested by menu command.)

Code of current procedure

The right pane of the window shows the code of the procedure currently being executed. The arrow marks the next statement to be executed. Statements where the program is to stop and let the debugger regain control are highlighted in some way (Symantec uses diamond shaped check marks in the left margin, Borland uses colour coding of the lines). If you execute the program using the "Step" menu command you only execute one statement, a "Go" command will run to the next breakpoint.

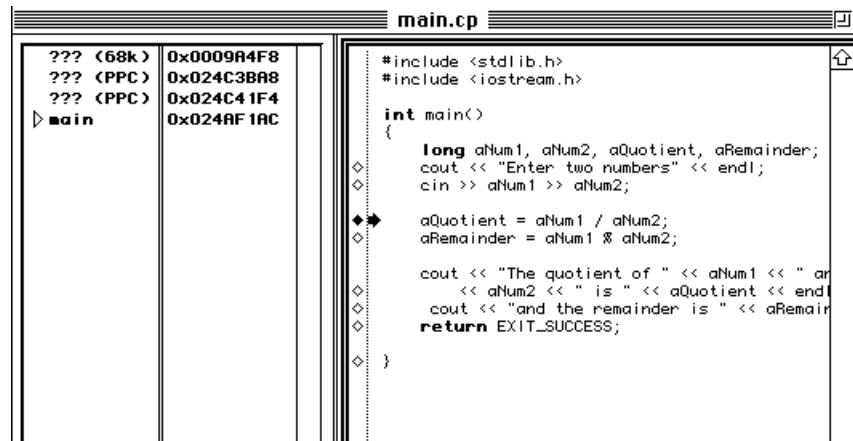


Figure 5.2 Display from a debugger.

The contents of variables can be inspected when the debugger is in control. Typically, you select the variable with the mouse, its value will be displayed in the separate Data window (it may be necessary to use a "Data" or "Inspect" menu command).

You should learn to use the debugger starting with your very first programming exercises. Initially, using the debugger to step through a program one statement at a time helps you understand how programs get executed. Later, you will find the debugger very useful for eliminating programming errors. It is easy to have programming errors such as defining a collection (array) with four data elements and then coding a loop slightly wrongly so that the program ends up checking for a fifth data element. Such code would be syntactically correct and so would be accepted by the compiler, but would either "bomb out" when run or would generate

an incorrect result. It is very hard to find such an error if all you know is that the program "bombs". It is relatively easy to find such errors if you are able to use the debugger to stop the program and check what is going.

6 Sequence

The simplest programs consist just of sequences of statements with no loops, no selections amongst alternative actions, no use of subroutines (other than possibly those provided in the input/output or mathematical libraries).

They aren't very interesting as programs! They do things that you could usually do on a calculator (at least you could probably do with a calculator that had some memory to store partial results). Most often they involve simply a change of scale (e.g. you have an amount in \$US and want it in ¥, or you have a temperature on the Celsius scale and want it in Fahrenheit, or you have an H⁺ ion concentration and want a pH value). In all such programs, you input a value, use some formula ("x units on scale A equals y units on scale B") and print the result.

Such programming problems, which are commonly used in initial examples, are meant to be so simple that they don't require any real thought as to how to solve the problem. Because the problem is simple, one can focus instead on the coding, all those messy little details, such as:

- the correct forms for variable names,
 - the layout of variable declarations and statements,
- and
- the properties of the "operators" that are used to combine data values.

6.1 OVERALL STRUCTURE AND MAIN() FUNCTION

The first few programs that you will write all have the same structure:

```
#include <iostream.h>
#include <math.h>
/*
  This program converts [H+] into
  pH using formula
    pH = - log10 [H+]
```

```

*/

int main()
{
    double Hplus;
    double pH;

    cout << "Enter [H+] value";
    ...
    return 0;
}

```

#includes The program file will start with `#include` statements that tell the compiler to read the "headers" describing the libraries that the program uses. (You will always use `iostream.h`, you may use other headers in a particular program.)

```

#include <iostream.h>
#include <math.h>

```

Introductory comments It is sensible to have a few comments at the start of a program explaining what the program does.

```

/*
    This program converts [H+] into
    pH using formula
    pH = - log10 [H+]
*/

```

Program outline The outline for the program may be automatically generated for you by the development environment. The outline will be in the form `void main() { ... }`, or `int main() { ... return 0; }`.

Own data definitions and code You have to insert the code for your program between the { ("begin") and } ("end") brackets.

The code will start with definitions of constants and variables:

```

double Hplus;
double pH;

```

and then have the sequence of statements that describe how data values are to be combined to get desired results.

Libraries

What libraries do you need to `"#include"`? Someone will have to tell you which libraries are needed for a particular assignment.

Almost standard libraries! The libraries are not perfectly standardized. You find difference between environments (e.g. the header file for the maths library on Unix contains definitions of the values of useful constants like $\pi = 3.14159\dots$, the corresponding header file with Symantec C++ doesn't have these constants).

Libraries whose headers often get included are:

iostream	standard C++ input output library
stdio	alternate input output library (standard for C programs)
math	things like sine, cosine, tan, log, exponential etc.
string	functions for manipulating sequences of characters ("strings") – copying them, searching for occurrences of particular letters, etc.
ctype	functions for testing whether a character is a letter or a digit, is upper case or lower case, etc.
limits	constants defining things like largest integer that can be used on a particular machine
stdlib	assorted "goodies" like a function for generating random numbers.

6.2 COMMENTS

"Comments" are sections of text inserted into a program that are ignored by the compiler; they are only there to be read by the programmers working on a project. Comments are meant to explain how data are used and transformed in particular pieces of code.

Comments are strange things. When you are writing code, comments never seem necessary because it is "obvious" what the code does. When you are reading code (other peoples' code, or code you wrote more than a week earlier) it seems that comments are essential because the code has somehow become incomprehensible.

C++ has two constructs that a programmer can use to insert comments into the text of a program.

"Block comments" are used when you need several lines of text to explain the purpose of a complete program (or of an individual function). These start with the character pair `/*` (no spaces between the `/` and the `*`), and end with `*/` (again, no space) e.g.

"Block comments"

```
/*
   Program for assignment 5; program reads 'customer'
   records from
   ....
*/
```

Be careful to match `/*` (begin comment) and `*/` (end comment) markers! Some very puzzling errors are caused by comments with wrongly matched begin-end markers. Such incorrectly formed comments "eat" code; the program looks as if it has coded statements but the compiler ignores some of these statements because it thinks they are meant to be part of a comment. (You won't get such problems with the better IDEs because they use differently coloured letters for comments making it much less likely that you would do something wrong.)

C++ shares "block comments" with C. C++ also has "line comments". These start with `//` (two `/` characters, no space between them) and continue to the end of a line. They are most commonly used to explain the roles of individual variables:

"Line comments"

```
int count1; // Count of customers placing orders today
int count2; // Count of items to be delivered today
int count3; // Count of invoices dispatched
```

6.3 VARIABLE DEFINITIONS

A variable definition specifies the type of a variable, gives it a name, and sometimes can give it an initial value.

"Built in" types

You will start with variables that are of simple "built in" types. (These types are defined in the compiler.) These are:

<u>type</u>	<u>(short name)</u>	<u>data</u>
short int	short	integer value in range -32768 to +32768
long int	long	integer value in range ±2,000million
float		real number (low accuracy)
double		real number (standard accuracy)
char		character (a letter, digit, or other character in defined character set)

Integer variables are often declared as being of type `int` rather than `short` or `long`. This can cause problems because some systems use `int` as synonymous with `short int` while others take it to mean `long int`. This causes programs using `ints` to behave differently on different systems.

Example definitions:

```
short   aCount;
double  theTotal;
char     aGenderFlag; // M = male, F = Female
```

You can define several variables of the same type in a single definition:

```
long   FailCount, ECount, DCount, CCount, BCount, ACount;
```

Definitions do get a lot more complex than this. Later examples will show definitions of "derived types" (e.g. an array of integers as a type derived from integer) and, when structs and classes have been introduced, examples will have variables of programmer defined types. But the first few programs will use just simple variables of the standard types, so the complexities of elaborate definitions can be left till later.

Since the first few programs will consist solely of a `main()` routine, all variables belong to that routine and will only be used by that routine. Consequently, variables will be defined as "locals" belonging to `main()`. This is

achieved by putting their definitions within `main()`'s `{` (begin) and `}` (end) brackets.

```
int main()
{
    long aCount;
    double theTotal;
    ...
}
```

6.4 STATEMENTS

We begin with programs that are composed of:

"input statements"	these get the "cin object" to read some data into variables,
"assignment statements"	these change values of variables,
"output statements"	these get the "cout object" to translate the data values held in variables into sequences of printable characters that are then sent to an output device.

In fact, most of the early programs are going to be of the form

```
main()
{
    get some data           // Several input statements, with
                           // maybe some outputs for prompts
    calculate with data     // Several assignment statements
    output results         // Several more output statements
}
```

Input

These statements are going to have the form:

```
cin >> Variable;
```

where *Variable* is the name of one of the variables defined in the program.

Examples

```
short    numItems;
double   delta;
char     flag;
...
cin >> numItems;
...
cin >> delta;
...
cin >> flag;
...
```

If several values are to be read, you can concatenate the input statements together.
Example

```
double    xcoord, ycoord, zcoord;
...
cin >> xcoord;
cin >> ycoord;
cin >> zcoord;
...
```

or

```
double    xcoord, ycoord, zcoord;
...
cin >> xcoord >> ycoord >> zcoord;
...
```

***Common error made
by those who have
programmed in other
languages***

The `cin` object has to give data to each variable individually with a `>>` operator. You may have seen other languages where input statements have the variable names separated by commas e.g. `xcoord, ycoord`. Don't try to follow that style! The following is a legal C++ statement:

```
cin >> xcoord, ycoord;
```

***cin's handling of >>
operation***

but it doesn't mean read two data values! Its meaning is actually rather odd. It means: generate the code to get `cin` to read a new value and store it in `xcoord` (OK so far) then generate code that fetches the current value of `ycoord` but does nothing with it (i.e. load its value into a CPU register, then ignores it).

The `cin` object examines the characters in the input stream (usually, this stream comes from the keyboard) trying to find a character sequence that can be converted to the data value required. Any leading "white space" is ignored ("white space" means spaces, newlines, tabs etc). So if you ask `cin` to read a character, it will try to find a printable character. Sometimes, you will want to read the whitespace characters (e.g. you want to count the number of spaces between words). There are mechanisms (explained later when you need them) that allow you to tell the `cin` object that you want to process whitespace characters.

Output

Output statements are going have the form

```
cout << Variable;
```

where *Variable* is the name of one of the variables defined in the program, or

```
cout << Text-String;
```

where *Text-String* is a piece of text enclosed in double quotes. Examples:

```
cout << "The coords are: ";

cout << xcoord;
```

Usually, output operations are concatenated:

```
cout << "The coords are:  x " << xcoord << ", y  "
      << ycoord << ", z " << zcoord;
```

(An output statement like this may be split over more than one line; the statement ends at the semi-colon.)

As previously explained (in the section 5.2), the `iostream` library defines something called "endl" which knows what characters are needed to get the next data output to start on a new line. Most concatenated output statements use endl:

```
cout << "The pH of the solution is " << pHValue << endl;
```

In C++, you aren't allowed to define text strings that extend over more than one line. So the following is not allowed: *Long text strings*

```
cout << "  Year      Month      State      Total-Sales  Tax-
due" << endl;
```

But the compiler allows you to write something like the following ...

```
cout << "  Year      Month      State      "
      "Total-Sales  Tax-due" << endl;
```

It understands that the two text strings on successive lines (with nothing else between them) were meant as part of one longer string.

Sometimes you will want to include special characters in the text strings. For example, if you were producing an output report that was to be sent to a printer you might want to include special control characters that the printer could interpret as meaning "start new page" or "align at 'tab-stop'". These characters can't be typed in and included as part of text strings; instead they have to be coded.

Special control characters

You will often see coded characters that use a simple two character code (you will sometimes see more complex four character codes). The two character code scheme uses the "backslash" character \ and a letter to encode each one of the special control characters.

Commonly used two character codes are:

\t	tab
\n	newline
\a	bell sound
\p	new page?
\\	when you really want a \ in the output!

Output devices won't necessarily interpret these characters. For example a "tab" may have no effect, or may result in output of one space (or 4 spaces, or 8 spaces),

or may cause the next output to align with some "tab-position" defined for the device.

Generally, '\n' causes the next output to appear at the start of new line (same effect as endl is guaranteed to produce). Consequently, you will often see programs with outputs like

```
cout << "Year      Month      Sales\n"
```

rather than

```
cout << "Year      Month      Sales" << endl;
```

The style with '\n' is similar to that used in C. You will see many examples using '\n' rather than endl; the authors of these examples probably have been programming in C for years and haven't switched to using C++'s endl.

Calculations

Along with input statements to get data in, and output statements to get results back, we'd better have some ways of combining data values.

The first programs will only have assignment statements of the form

```
Variable = Arithmetic Expression;
```

where *Variable* is the name of one of the variables defined for the program and *Arithmetic Expression* is some expression that combines the values of variables (and constants), e.g.

```
double s, u, a, t;
cout << "Enter initial velocity, acceleration, and time : ";
cin >> u >> a >> t;

s = u*t + 0.5*a*t*t;

cout << "Distance travelled " << s << endl;
```

The compiler translates a statement like

```
s = u*t + 0.5*a*t*t;
```

into a sequence of instructions that evaluate the arithmetic expression, and a final instruction that stores the calculated result into the variable named on the left of the = (the "assignment operator").

"lvalue"

The term "lvalue" is often used by compilers to describe something that can go on the left of the assignment operator. This term may appear in error messages. For example, if you typed in an erroneous statement like "3 = n - m;" instead of say "e = n - m;" the error message would probably be "lvalue required". The

compiler is trying to say that you can't have something like 3 on the left of an assignment operator, you have to have something like the name of a variable.

As usual, C++ allows abbreviations, if you want several variables to contain the same value you can have a concatenated assignment, e.g.

```
xcoord = ycoord = zcoord = 1.0/dist;
```

The value would be calculated, stored in `zcoord`, copied from `zcoord` to `ycoord`, then to `xcoord`. Again be careful, the statement "`xcoord, ycoord, zcoord = 1.0/dist;`" is legal C++ but means something odd. Such a statement would be translated into instructions that fetch the current values of `xcoord` and `ycoord` and then does nothing with their values, after which there are instructions that calculate a new value for `zcoord`.

Arithmetic expressions use the following "operators"

*Arithmetic
"operators"*

+	addition operator
-	subtraction operator (also "unary minus sign")
*	multiplication operator
/	division operator
%	remainder operator (or "modulo" operator)

Note that multiplication requires an explicit operator. A mathematical formula may get written something like

```
v = u + a t
```

But that isn't a valid expression in C++; rather than `a t` you must write `a*t`.

You use these arithmetic operators with both real and integer data values (% only works for integers).

Arithmetic expressions work (almost) exactly as they did when you learnt about them at school (assuming that you were taught correctly!). So,

```
v = u + a * t
```

means that you multiply the value of `a` by the value of `t` and add the result to the value of `u` (and you don't add the value of `u` to `a` and then multiply by `t`).

The order in which operators are used is determined by their *precedence*. The multiply (*) and divide (/) operators have "higher precedence" than the addition and subtraction operators, which means that you do multiplies and divides before you do addition and subtraction. Later, we will encounter more complex expressions that combine arithmetic and comparison operators to build up logical tests e.g. when coding something like "if sum of income and bonus is greater than tax threshold or the interest income exceeds value in entry fifteen then ...". The operators (e.g. + for addition, > for greater than, || for or etc) all have defined precedence values; these precedence values permit an unambiguous interpretation of a complex expression. You will be given more details of precedence values of operators as they are introduced.

Operator precedence

"Arithmetic expressions work (almost) exactly as they did when you learnt them at school". Note, it does say "almost". There are actually a number of glitches. These arise because:

- In the computer you only have a finite number of bits to represent a number

This leads to two possible problems

your integer value is too big to represent

your real number is only represented approximately

- You have to be careful if you want to combine integer and real values in an expression.

You may run into problems if you use short integers. Even if the correct result of a calculation would be in the range for short integers (-32768 to +32767) an intermediate stage in a calculation may involve a number outside this range. If an out of range number is generated at any stage, the final result is likely to be wrong. (Some C++ compilers protect you from such problems by using long integers for intermediate results in all calculations.)

Integer division is also sometimes a source of problems. The "/" divide operator used with integers throws away any fraction part, so 10/4 is 2. Note that the following program says "result = 2" ...

```
void main()
{
    int i, j;
    double result;
    i = 10;
    j = 4;
    result = i / j;
    cout << "result = " << result << endl;
}
```

Because the arithmetic is done using integers, and the integer result 2 is converted to a real number 2.0.

6.5 EXAMPLES

6.5.1 "Exchange rates"

Specification:

1. The program is to convert an amount of money in Australian dollars into the corresponding amount given in US dollars.
2. The program will get the user to input two values, first the current exchange rate then the money amount. The exchange rate should be given as the

amount of US currency equivalent to \$A1, e.g. if \$A1 is 76¢ US then the value given as input should be 0.76.

3. The output should include details of the exchange rate used as well as the final amount in US money.

Program design

1. What data?

Seems like three variables:

exchange rate
aus dollars
us dollars

If these are "real" numbers (i.e. C++ floats or doubles) we can use the fraction parts to represent cents.

2. Where should variables be defined?

Data values are only needed in a single `main()` routine, so define them as local to that routine.

3. Pseudo-code outline
 - a) prompt for and then input exchange rate;
 - b) prompt for and then input Australian dollar amount;
 - c) calculate US dollar amount;
 - d) print details of exchange rate and amount exchanged;
 - e) print details of US dollars obtained.

4. Test data

If the exchange rate has $\$A1.00 = \$US0.75$ then \$A400 should buy \$US300. These values can be used for initial testing.

Implementation

Use the menu options in your integrated development environment to create a new project with options set to specify standard C++ and use of the `iostream` library. The IDE should create a "project" file with a main program file (probably named "main.cp" but the name may vary slightly according to the environment that you are using).

The IDE will create a skeletal outline for `main()`, e.g. Symantec 8 gives:

```
#include <stdlib.h>
```

```
#include <iostream.h>

int main()
{
    cout << "hello world" << endl;

    return EXIT_SUCCESS;
}
```

(Symantec has the name `EXIT_SUCCESS` defined in its `stdlib`; it just equates to zero, but it makes the return statement a little clearer.) Replace any junk filler code with your variable declarations and code:

```
int main()
{
    double ausdollar;
    double usdollar;
    double exchangerate;

    cout << "Enter the exchange rate : ";
    cin >> exchangerate;
    cout << "How many Australian dollars do you want "
          "to exchange? ";
    cin >> ausdollar;

    usdollar = ausdollar*exchangerate;
    cout << "You will get about $" << usdollar <<
          " less bank charges etc." << endl;

    return EXIT_SUCCESS;
}
```

Insert some comments explaining the task performed by the program. These introductory comments should go either at the start of the file or after the `#include` statements.

```
#include <stdlib.h>
#include <iostream.h>
/*
    Program to do simple exchange rate calculations
*/

int main()
{
    ...
}
```

Try running your program under control from the debugger. Find how to execute statements one by one and how to get data values displayed. Select the data variables `usdollar` etc for display before starting program execution; each time a variable is changed the value displayed should be updated in the debugger's data window. The values initially displayed will be arbitrary (I got values like `1.5e-303`); "automatic" variables contain random bit patterns until an assignment statement is executed.

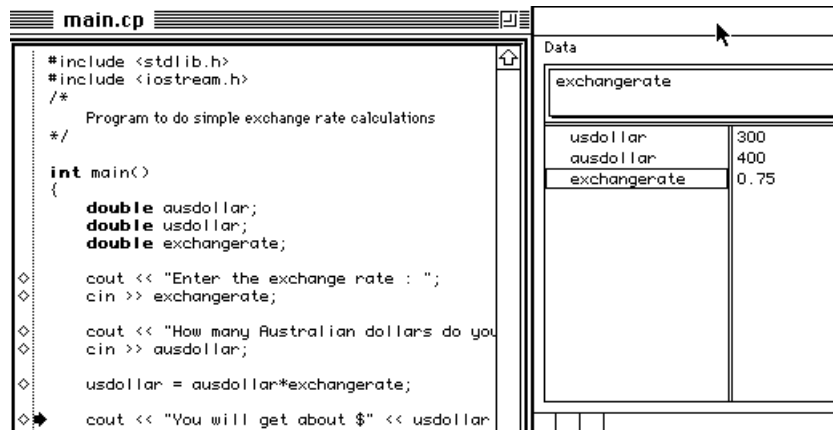


Figure 6.1 Screen dump of program run through the debugger.

Figure 6. 1 is a screen dump showing the program being executed in the Symantec 8 environment.

6.5.2 pH

Specification:

1. The program is to convert a "pH" value into a H⁺ ion concentration ([H⁺]) using (in reverse) the definition formula for pH:

$$\text{pH} = -\log_{10} [\text{H}^+]$$

If you have forgotten you high school chemistry, the [H⁺] concentration defines the acidity of an aqueous solution; its units are in "moles per litre" and values vary from about 10 (very acidic) down to 0.000000000000001 (highly alkaline). The pH scale, from about -1 to about 14, is just a more convenient numeric scale for talking about these acidity values.

2. The user is to enter the pH value (should be in range -1..14 but there is no need to check this); the program prints both pH value and [H⁺].

Program design

1. Data:
Just the two values, pH and HConc, both would be "doubles", both would be defined as local to the main program.
2. The formula has to be sorted out:

$$\text{pH} = -\log_{10} [\text{H}^+]$$

so

$$[H^+] = 10^{-pH}$$

(meaning 10 to power -pH).

How to calculate that?

This can be calculated using a function provided in the maths support library `math.h`; a detailed explanation is given at the start of "Implementation" below.

3. Program organization:

Need to `#include` the maths library in addition to `iostream.h`.

4. Pseudo code outline

- a) prompt for and then input pH;
- b) calculate HConc;
- d) print details of pH and corresponding $[H^+]$ value.

Implementation

Finding out about library functions

Many assignments will need a little "research" to find out about standard library functions that are required to implement a program.

Here, we need to check what the `math.h` library offers. Your IDE will let you open any of the header files defining standard libraries. Usually, all you need do is select the name of a header file from among those listed at the top of your program file and use a menu command such as "Open selection". The header file will be displayed in a new editor window. Don't change it! Just read through the list of functions that it defines.

The library `math.h` has a function

```
double pow(double d1, double d2)
```

which computes $d1$ to the power $d2$. We can use this function.

Getting details about library functions

The Borland IDE has an integrated help system. Once you have a list of functions displayed you can select any function name and invoke "Help". The help system will display information about the chosen function. (Symantec's IDE is slightly clumsier; you have to have a second "Reference" program running along with the IDE. But you can do more or less the same name lookup as is done directly in the Borland system; `pow` is in Symantec's Reference database called SLR – standard library routines.)

Using simple mathematical functions

The use of functions is examined in more detail in Chapters 11 and 12. But the standard mathematical functions represent a particularly simple case. The maths functions include:

Function prototypes of some maths functions

<code>double cos(double);</code>	<i>cosine function</i>
<code>double sin(double);</code>	<i>sine function</i>
<code>double sqrt(double);</code>	<i>square-root</i>
<code>double tan(double);</code>	<i>tangent</i>
<code>double log10(double);</code>	<i>logarithm to base 10</i>

```
double pow(double, double);    power
```

These are "function prototypes" – they specify the name of the function, the type of value computed (in all these cases the result is a double real number), and identify the data that the functions need. Apart from `pow()` which requires two data items to work with, all these functions require just one double precision data value. The functions `sin()`, `cos()` and `tan()` require the value of an angle (in radians); `sqrt()` and `log10()` must be given a number whose root or logarithm is to be calculated.

You can understand how to work with these functions in a program by analogy with the similar function keys on a pocket calculator. If you need the square root of a number, you key it in on the numeric keys and get the number shown in the display register. You then invoke the `sqrt` function by activating its key. The `sqrt` function is implemented in code in the memory of the microprocessor in the calculator. The code takes the value in the display register; computes with it to get the square root; finally the result is put back into the display register.

It is very much the same in a program. The code for the *calling* program works out the value for the data that is to *be passed* to the function. This value is placed on the stack (see section 4.8), which here serves the same role as the display in the calculator. Data values passed to functions are referred to as "arguments of the function". A subroutine call instruction is then made (section 2.4) to invoke the code of the function. The code for the function takes the value from the stack, performs the calculation, and puts the result back on the stack. The function returns to the calling program ("return from subroutine" instruction). The calling program collects the result from the stack.

"Passing arguments to a function"

The following code fragment illustrates a call to the sine function passing a data value as an argument:

Using a mathematical function

```
double angle;
double sine;
cout << "Enter angle : ";
cin >> angle;
sine = sin(angle*3.141592/180.0);
cout << "Sine(" << angle << ") = " << sine << endl;
```

The implementation of the pH calculation program is straightforward. The IDE's skeletal outline for `main()` has to be modified to include the `math.h` library, and the two variables must be declared:

```
#include <stdlib.h>
#include <iostream.h>
#include <math.h>

int main()
{
    double pH;
    double HConc;
```

The body of the main function must then be filled out with the code prompting for and reading the pH value and then doing the calculation:

```

cout << "Enter pH : ";
cin >> pH;

HConc = pow(10.0, -pH);

cout << "If the pH is " << pH << " then the [H+] is " <<
      HConc << endl;

```

6.6 NAMING RULES FOR VARIABLES

C++ has rules that limit the forms of names used for variables.

- Variables have names that start with a letter (uppercase or lowercase), or an underscore character (`_`).
- The names contain only letters (abcdefghijklmnopqrstuvwxyzABC...YZ), digits (0123456789) and underscore characters `_`.
- You can't use C++ "reserved words".

In addition, you should remember that the libraries define some variables (e.g. the variable `cin` is defined in the `iostream` library). Use of a name that already has been defined in a library that you have `#included` will result in an error message.

Avoid names that start with the underscore character `"_"`. Normally, such names are meant to be used for extra variables that a compiler may need to invent.

C++'s "reserved words"

C++'s "reserved words" are:

<code>asm</code>	<code>auto</code>	<code>bool</code>	<code>break</code>	<code>case</code>
<code>catch</code>	<code>char</code>	<code>class</code>	<code>const</code>	<code>const_cast</code>
<code>continue</code>	<code>default</code>	<code>delete</code>	<code>do</code>	<code>double</code>
<code>dynamic_cast</code>	<code>else</code>	<code>enum</code>	<code>extern</code>	<code>false</code>
<code>float</code>	<code>for</code>	<code>friend</code>	<code>goto</code>	<code>if</code>
<code>inline</code>	<code>int</code>	<code>long</code>	<code>mutable</code>	<code>namespace</code>
<code>new</code>	<code>operator</code>	<code>private</code>	<code>protected</code>	<code>public</code>
<code>register</code>	<code>reinterpret_cast</code>	<code>return</code>	<code>short</code>	<code>switch</code>
<code>signed</code>	<code>sizeof</code>	<code>static</code>	<code>struct</code>	<code>typedef</code>
<code>template</code>	<code>this</code>	<code>throw</code>	<code>try</code>	<code>using</code>
<code>typeid</code>	<code>typename</code>	<code>union</code>	<code>unsigned</code>	<code>while</code>
<code>virtual</code>	<code>void</code>	<code>volatile</code>	<code>wchar_t</code>	

The following words (classed as "alternative representations for operators") are also reserved:

<code>bitand</code>	<code>and</code>	<code>bitor</code>	<code>xor</code>	<code>compl</code>
<code>and_eq</code>	<code>or_eq</code>	<code>xor_eq</code>	<code>not</code>	<code>not_eq</code>

You can't use any of these words as names of variables. (C++ is still undergoing standardization. The set of reserved words is not quite finalized. Some words

given in the list above are relatively new additions and some compilers won't regard them as reserved.)

Unlike some other languages, C++ (and C) care about the case of letters. So, for example, the names `largest`, `Large`, `LARGE` would be names of *different* variables. (You shouldn't use this feature. Any coworkers who have to read your program will inevitably get confused by such similar names being used for different data items.)

6.7 CONSTANTS

You often need constants in your program:

```
gravitational constant (acceleration by earth's gravity,
                        approximately 32 feet per second per second)
speed of light
 $\pi$ 
marks used to define grades (45, 50, 65, 75, 85 etc)
size of "window" to be displayed on screen (width = ..., height = ...)
...
```

It is a *bad* idea to type in the value of the constant at each place you use it:

Don't embed "magic numbers" in code!

```
/* cannonball is accelerated to earth by gravity ... */
Vert_velocity = -32.0*t*t;
...
```

If you need to change things (e.g. you want to calculate in metres rather than feet) you have to go right through your program changing things (the gravitational constant is 9.8 not 32.0 when you change to metres per second per second).

It is quite easy to make mistakes when doing such editing. Further, the program just isn't clear when these strange numbers turn up in odd places (one keeps having to refer back to separate documentation to check what the numbers might represent).

C++ allows you to define constants – named entities that have an fixed values set before the code is run. (The compiler does its best to prevent you from treating these entities as variables!) Constants may be defined local to a particular routine, but often they are defined at the start of a file so that they can be used in any routine defined later in that file.

A constant definition has a form like

Definition of a constant

```
const   Type  Name = Value;
```

Type is the type of the constant, short, long, double etc. *Name* is its name. *Value* is the value (it has to be something that the compiler can work out and is appropriate to the type specified).

Examples

"const"

```
const double g = 32.0; /* Gravitational constant in fpsps */
```

```

const double Neutral_pH = 7.0; /* pH of pure water */
const double Pool_pH = 7.4;
      /* Swimming pool is slightly alkaline */
const int PCMARK = 45;
const int PMARK = 50;
const int CREDITMARK = 65;

const char TAB = '\t';
const char QUERY = '?';

const double PI = 3.1415926535898;
const double SOMENUM = 1.235E75;

const double DEGREES_TO_RADIANS = PI / 180.0;

```

As shown in last example, the value for a constant isn't limited to being a simple number. It is anything that the compiler can work out.

Character constants, e.g. `Query`, have the character enclosed in single quote marks (if the character is one of those like `TAB` that is represented by a special two or four character code then this character code is bracketed by single quotes). Double quote marks are used for multi-character text strings like those already shown in output statements of example programs.

Integer constants have the sign (optional if positive) followed by a digit sequence. Real number constants can be defined as `float` or `double`; fixed and scientific formats are both allowed: `0.00123`, or `1.23E-3`.

You will sometimes see constant definitions with value like `49L`. The `L` after the digits informs the compiler that this number is to be represented as a long integer. You may also see numbers given in hexadecimal, e.g. `0xA3FE`, (and even octal and binary forms); these usually appear when a programmer is trying to define a specific bit pattern. Examples will be given later when bitwise operations are illustrated.

#define

The keyword `const` was introduced in C++ and later was retrofitted to the older C language. Before `const`, C used a different way of defining constants, and you will certainly see the old style in programs written by those who learnt C a long time ago. C used "*#define macros*" instead of `const` definitions:

```

#define g 32.0
#define CREDITMARK 65

```

(There are actually subtle differences in meaning; for example, the `#define` macros don't specify the type of the constant so a compiler can't thoroughly check that a constant is used only in correct contexts.)

6.7.1 Example program with some `const` definitions.

Specification

The program it print the area and the circumference of a circle with a radius given as an input value.

Program design

1. **Data:**
The three values, radius, area, and circumference, all defined as local to the main program.
2. The value π has to be defined as a const. It can be defined inside main routine as there are no other routines that would want to use it.

```
int main()
{
    const double PI = 3.1415926535898;

    double radius;
    double area;
    double circumference;

    cout << "Enter radius : ";
    cin >> radius;

    area = PI*radius*radius;
    circumference = 2.0*PI*radius;

    cout << "Area of circle of radius " << radius << " is "
         << area << endl;
    cout << "\tand its circumference is " << circumference
         << endl;
}
```

6.8 INITIALIZATION OF VARIABLES

When you plan a program, you think about the variables you need, you decide how these variables should be initialized, and you determine the subsequent processing steps. Initially, you tend to think about these aspects separately. Consequently, in your first programs you have code like:

```
int main()
{
    double x, y, z;
    int n;
    ...
    // Initialize
    x = y = z = 1.0;
    n = 0;
    ...
}
```

The initialization steps are done by explicit assignments after the definitions.

This is not necessary. You can specify the initial values of variables as you define them.

```
int main()
```

```
{
    double x = 1.0, y = 1.0, z = 1.0;
    int n = 0;
```

Of course, if a variable is immediately reset by an input statement there isn't much point in giving it an initial value. So, in the example programs above, things like the radius `r` and the `pH` value were defined but not initialized.

***Beware of
uninitialized
variables***

However, you should be careful about initializations. It has been noticed that uninitialized variables are a common source of errors in programs. If you forget to initialize an "automatic variable" (one of the local variables defined in a function) then its initial value is undefined. It usually gets to have some arbitrary numeric value based on whatever bit pattern happened to be found in the memory location that was set aside for it in the stack (this bit pattern will be something left over from a previously executing program). You are a bit safer with globals – variables defined outside of any function – because these should be initialized to zero by the loader.

It is easy to miss out an initialization statement that should come after the definition of a local variable. It is a little safer if you combine definition with initialization.

6.9 CONVERTING DATA VALUES

What should a compiler do with code like the following?

```
int main()
{
    int    n;
    double d;
    ...
    cin >> n;
    ...
    d = n;
```

The code says assign the value of the integer `n` to the double precision real number `d`.

As discussed in Chapter 1, integers and real numbers are represented using quite different organization of bit pattern data and they actually require different numbers of bytes of memory. Consequently, the compiler cannot simply code up some instructions to copy a bit pattern from one location to another. The data value would have to be converted into the appropriate form for the lvalue.

The compilers for some languages (e.g. Pascal) are set to regard such an assignment as a kind of mistake; code equivalent to that shown above would result in a warning or a compilation error.

Such assignments are permitted in C++. A C++ compiler is able to deal with them because it has an internal table with rules detailing what it has to do if the type of the lvalue does not match the type of the expression. The rule for converting integers to reals are simple (call a "float" routine that converts from integer to real). This rule would be used when assigning an integer to a double as

in the assignment shown above, and also if the compiler noticed that a function that needed to work on a double data value (e.g. `sqrt()`) was being passed an integer. Because the compiler checks that functions are being passed the correct types of data, the following code works in C++:

```
double d;
...
d = sqrt(2);
```

Since the `math.h` header file specifies that `sqrt()` takes a double, a C++ compiler carefully puts in extra code to change the integer 2 value into a double 2.0.

Conversions of doubles to integers necessarily involve loss of fractional parts; so the code:

```
...
int    m, n;
double x, y;
...
x = 17.7; y = - 15.6;
m = x; n = y;
cout << m << endl;
cout << n << endl;
```

will result in output of 17 and -15 (note these aren't the integers that would be closest in value to the given real numbers).

The math library contains two functions that perform related conversions from doubles to integers. The `ceil()` function can be used to get the next integer greater than a given double:

```
x = 17.7; y = - 15.6;
m = ceil(x); n = ceil(y);
cout << m << endl;
cout << n << endl;
```

will print the values 18 and -15.

Similarly there is a `floor()` function; it returns the largest integer smaller than a given double value. If you want to get the closest integer to real number `x`, you can try `floor(x + 0.5)` (this should work for both positive and negative `x`).

The code fragments above all printed data values one per line. What should you get from code like this:

```
int    n, m;
...
n = 55;
m = 17;
...
cout << n << m << endl
```

You get a single line of output, with apparently just one number; the number 5517.

The default setting for output to `cout` prints numbers in the minimum amount of space. So there are no spaces in front of the number, the sign is only printed if the

*Minor note regarding
default output
formats*

number is negative, and there are no trailing spaces printed after the last digit. The value fifty five can be printed using two 5 digits, the digits 1 and 7 for the value seventeen follow immediately.

Naturally, this can confuse people reading the output. ("Shouldn't there be two values on that line?" "Yes. The values are five and five hundred and seventeen, I just didn't print them tidily.")

If you are printing several numeric values on one line, put spaces (or informative text labels) between them. At least you should use something like:

```
cout << n << ", " << m << endl;
```

so as to get output like "55, 17".

EXERCISES

Programs that only use sequences of statements are severely limited and all pretty much the same. They are still worth trying – simply to gain experience with the development environment that is to be used for more serious examples.

1. Write a program that given the resistance of a circuit element and the applied voltage will calculate the current flowing (Ohm's law). (Yes you do remember it – $V = IR$.)
2. Write a program that uses Newton's laws of motion to calculate the speed and the distance travelled by an object undergoing uniform acceleration. The program is to take as input the mass, and initial velocity of an object, and the constant force applied and time period to be considered. The program is to calculate the acceleration and then use this value to calculate the distance travelled and final velocity at the end of the specified time period.
3. Write a program that takes as input an investment amount, a percentage rate of compound interest, and a time period in years and uses these data to calculate the final value of the investment (use the `pow()` function).

7 Iteration

7.1 WHILE LOOPS

Loop constructs permit slightly more interesting programs. Like most Algol-family languages, C++ has three basic forms of loop construct:

- while-loops (*while condition X is true keeping doing the following ...*)
- for-loops (mainly a "counting loop" construct – *do the following ten times* – but can serve in more general roles)
- repeat-loops (*do the following ... while condition Z still true*)

Repeat-loops are the least common; for-loops can become a bit complex; so, it is usually best to start with "while" loops.

These have the basic form:

While loops

```
while ( Expression )
    body of loop;
```

Where *Expression* is something that evaluates to "True" or "False" and body of loop defines the work that is to be repeated. (The way the values True and False are represented in C/C++ is discussed more later.)

When it encounters a while loop construct, the compiler will use a standard coding pattern that lets it generate:

- instructions that evaluate the expression to get a true or false result,
 - a "jump if false" conditional test that would set the program counter to the start of the code following the while construct,
 - instructions that correspond to the statement(s) in the body of the loop,
- and
- a jump instruction that sets the program counter back to the start of the loop.

Sometimes, the body of the loop will consist of just a single statement (which would then typically be a call to a function). More often, the body of a loop will involve several computational steps, i.e. several statements.

Compound statements

Of course, there has to be a way of showing which statements belong to the body of the loop. They have to be clearly separated from the other statements that simply follow in sequence after the loop construct. This grouping of statements is done using "compound statements".

C and C++ use { ("begin bracket") and } ("end bracket") to group the set of statements that form the body of a loop.

```
while ( Expression ) {
    statement-1;
    statement-2;
    ...
    statement-n;
}
/* statements following while loop, e.g. some output */
cout << "The result is ";
```

(These are the same begin-end brackets as are used to group all the statements together to form a main-routine or another routine. Now we have outer begin-end brackets for the routine; and inner bracket pairs for each loop.)

Code layout

How should "compound statements" be arranged on a page? The code should be arranged to make it clear which group of statements form the body of the while loop. This is achieved using indentation, but there are different indenting styles (and a few people get obsessed as to which of these is the "correct" style). Two of the most common are:

<pre>while (Expression) { statement-1; statement-2; ... statement-n; }</pre>	<pre>while (Expression) { statement-1; statement-2; ... statement-n; }</pre>
--	--

Sometimes, the editor in the integrated development environment will choose for you and impose a particular style. If the editor doesn't choose a style, you should. Be consistent in laying things out as this helps make the program readable.

Comparison expressions

Usually, the expression that controls execution of the while loop will involve a test that compares two values. Such comparison tests can be defined using C++'s comparison operators:

<pre>== != > >= < <=</pre>	<pre>equals operator (note, two = signs; the single equals sign is the assignment operator that changes a variable) not equals operator greater than operator greater or equal operator less than operator less than or equal operator</pre>
--	--

7.2 EXAMPLES

7.2.1 Modelling the decay of CFC gases

Problem:

Those CFC compounds that are depleting the ozone level eventually decay away. It is a typical decay process with the amount of material declining exponentially. The rate of disappearance depends on the compound, and is usually defined in terms of the compound's "half life". For example, if a compound has a half life of ten years, then half has gone in ten years, only a quarter of the original remains after 20 years etc.

Write a program that will read in a half life in years and which will calculate the time taken (to the nearest half life period) for the amount of CFC compound to decay to 1% of its initial value.

Specification:

1. The program is to read the half life in years of the CFC compound of interest (the half life will be given as a real value).
2. The program is to calculate the time for the amount of CFC to decay to less than 1% of its initial value by using a simple while loop.
3. The output should give the number of years that are needed (accurate to the nearest half life, not the nearest year).

Program design

1. What data?
Some are obvious:
 a count of the number of half-life periods
 the half life

Another data item will be the "amount of CFC". This can be set to 1.0 initially, and halved each iteration of the loop until it is less than 0.01. (We don't need to know the actual amount of CFC in tons, or Kilos we can work with fractions of whatever the amount is.)

These are all "real" numbers (because really counting in half-lives, which may be fractional, rather than years).

2. Where should data be defined?
Data values are only needed in single main routine, so define them as local to that routine.

3. Processing:
 - a) prompt for and then input half life;
 - b) initialize year count to zero and amount to 1.0;
 - c) while loop:
 - terminates when "amount < 0.01"
 - body of loop involves
 - halving amount and
 - adding half life value to total years;
 - d) print years required.

Implementation

The variable declarations, and a constant declaration, are placed at the start of the main routine. The constant represents 1% as a decimal fraction; it is best to define this as a named constant because that makes it easier to change the program if the calculations are to be done for some other limit value.

```
int main()
{
    double      numyears;
    double      amount;
    double      half_life;

    const double limit = 0.01; // amount we want
```

The code would start with the prompt for, and input of the half-life value. Then, the other variables could be initialized; the amount is 1.0 (all the CFC remains), the year counter is 0.

```
cout << "Enter half life of compound " ;
cin >> half_life;

amount = 1.0; // amount we have now
numyears = 0.0; // Number of years we have waited
```

As always, the while loop starts with the test. Here, the test is whether the amount left exceeds the limit value; if this condition is true, then another cycle through the body of the loop is required.

```
while(amount > limit) {
    numyears = numyears + half_life;
    amount = amount*0.5;
}
```

In the body of the loop, the count of years is increased by another half life period and the amount of material remaining is halved.

The program would end with the output statements that print the result.

```
cout << "You had to wait " << numyears <<
```



```
" years" << endl;
```

7.2.2 Newton's method for finding a square root

Problem:

The math library, that implements the functions declared in the `math.h` header file, is coded using the best ways of calculating logarithms, square roots, sines and so forth. Many of these routines use power series polynomial expansions to evaluate their functions; sometimes, the implementation may use a form of interpolation in a table that lists function values for particular arguments.

Before the math library was standardized, one sometimes had to write one's own mathematical functions. This is still occasionally necessary; for example, you might need a routine to work out the roots of a polynomial, or something to find the "focus of an ellipse" or any of those other joyous exercises of senior high school maths.

There is a method, an algorithm, due to Newton for finding roots of polynomials. It works by successive approximation – you guess, see whether you are close, guess again to get a better value based on how close you last guess was, and continue until you are satisfied that you are close enough to a root. We can adapt this method to find the square root of a number. (OK, there is a perfectly good square root function in the maths library, but all we really want is an excuse for a program using a single loop).

*Successive
approximation
algorithm*

If you are trying to find the square root of a number x , you make successive guesses that are related according to the following formula

$$\text{new_guess} = 0.5 (x / \text{guess} + \text{guess})$$

This new guess should be closer to the actual root than the original guess. Then you try again, substituting the new guess value into the formula instead of the original guess.

Of course, you have to have a starting guess. Just taking $x/2$ as the starting guess is adequate though not always the best starting guess.

You can see that this approach should work. Consider finding the square root of 9 (which, you may remember, is 3). Your first guess would be 4.5; your second guess would be $0.5(9/4.5 + 4.5)$ or 3.25; the next two guesses would be 3.009615 and 3.000015.

You need to know when to stop guessing. That aspect is considered in the design section below.

Now, if you restrict calculations to the real numbers of mathematics, the square root function is only defined for positive numbers. Negative numbers require the complex number system. We haven't covered enough C++ to write any decent checks on input, the best we could do would be have a loop that kept asking the user to enter a data value until we were given something non-negative. Just to simplify things, we'll make it that the behaviour of the program is undefined for negative inputs.

Specification:

1. The program is to read the value whose root is required. Input is unchecked. The behaviour of the program for invalid data, such as negative numbers is not defined.
2. The program is to use half of the input value as the initial guess for its square root.
3. The program is to use a while loop to implement the method of successive approximations. The loop is to terminate when the iterative process converges (the way to test for convergence is discussed in the design section below).
4. The program is to print the current estimate for the root at each cycle of the iterative process.
5. The program is to terminate after printing again the number and the converged estimate of its square root.

Program design

1. What data?
 - the number whose root is required
 - the current estimate of the root
 - maybe some extra variables needed when checking for convergence

These should all be double precision numbers defined in the main program.

2. Processing:
 - a) prompt for and then input the number;
 - b) initialize root (to half of given number)
 - c) while loop:
 - test on convergence*
 - body of loop involves
 - output of current estimate of root
 - calculation of new root
 - d) print details of number and root

The only difficult part of the processing is the test controlling iteration. The following might *seem* suitable:

```
x  the number whose root is required
r  current guess for its square root

while(x != r*r)
```

This test will keep the iterative process going while x is not equal to r^2 (after all, that is the definition of r being the square root of x). Surprisingly, this doesn't work in practice, at least not for all values of x .

The reason it doesn't work is "roundoff error". That test would be correct if we were really working with mathematical real numbers, but in the computer we only have floating point numbers. When the iteration has progressed as far as it can, the value of root will be approximated by the floating point number that is closest to its actual real value – but the square of this floating point number may not be exactly equal the floating point value representing x .

The test could be rephrased. It could be made that we should keep iterating until the difference between x and r^2 is very small. The difference $x - r^2$ would be positive if the estimate r was a little too small and would be negative if r was a little too large. We just want the size of the difference to be small irrespective of whether the value difference is positive or negative. So, we have to test the absolute value of the difference. The maths library has a function `fabs()` that gets the absolute value of a double number. Using this function, we could try the following:

```
const double VERYSMALL = 1.0E-8;
...
while(fabs(x-r*r) > VERYSMALL)
```

This version would be satisfactory for a wide range of numbers but it still won't work for all. If you tried a very large value for x , e.g. $1.9E+71$, the iteration wouldn't converge. At that end of the number range, the "nearest floating point numbers" are several integer values apart. It simply isn't possible to get the fixed accuracy specified.

The test can't use an absolute value like $1.0E-8$. The limit has to be expressed relative to the numbers for which the calculation is being done.

The following test should work:

```
const double SMALLFRACTION = 1.0E-8;
...
while(fabs(x - r*r) > SMALLFRACTION*x) {
```

Implementation

The implementation is straightforward. The code would be:

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    const double SMALLFRACTION = 1.0E-8;
    double x;
    double r;
    cout << "Enter number : ";
    cin >> x;
```

```

    r = x / 2.0;
    while(fabs(x - r*r) > SMALLFRACTION*x) {
        cout << r << endl;
        r = 0.5 *(x / r + r);
    }

    cout << "Number was : " << x << ", root is "
          << r << endl;
    return EXIT_SUCCESS;
}

```

Test data are easy:

```

Enter number : 81
40.5
21.25
12.530882
9.497456
9.013028
9.000009
Number was : 81, root is 9

```

7.2.3 Tabulating function values

Problem

If you are to compare different ways of solving problems, you need to define some kind of performance scale. As you will learn more later, there are various ways of measuring the performance of programs.

Often, you can find different ways of writing a program that has to process sets of data elements. The time the program takes will increase with the number of data elements that must be processed. Different ways of handling the data may have run times that increase in different ways as the number of data elements increase. If the rate of increase is:

linear	time for processing 20 data elements will be twice that needed for 10 elements
quadratic	time for processing 20 data elements will be 4x that needed for 10 elements
cubic	time for processing 20 data elements will be 8x that needed for 10 elements
exponential	time for processing 20 data elements will be about one thousand times that needed for 10 elements

Usually, different ways of handling data are described in terms of how their run times increase as the number of data elements (N) increases. The kinds of functions encountered include

log(N)	<u>logarithmic</u> (very slow growth)
---------------	---------------------------------------

N	<u>linear</u>
$N * \log(N)$	<u>log linear</u> (slightly faster than linear growth)
N^2	<u>quadratic</u>
2^N	<u>exponential</u> (unpleasantly fast growth)
$N!$	<u>factorial</u> (horribly fast growth)
N^N	disgustingly fast growth

When you study algorithms later, you will learn which are linear, which are quadratic and so on. But you need some understanding of the differences in these functions, which you can get by looking at their numeric values.

The program should print values for these functions for values $N=1, 2, 3, \dots$. The program is to stop when the values get too large (the function N^N , "N to the power N", grows fastest so the value of this function should be used in the check for termination).

Specification:

1. The program is to print values of $N, N^2, N^3, \log(N), N*\log(N), 2^N$ and N^N for values of $N = 1, 2, 3, 4, \dots$
2. The loop that evaluates these functions for successive values of N is to terminate when the value of N^N exceeds some large constant.

Program design

1. What data?
Value N , also variables to hold values of $N^2, N^3, \log(N)$ etc

Some could be integers, but the ones involving logarithms will be real numbers. Just for simplicity, all the variables can be made real numbers and represented as "doubles".
2. Where should data be defined?
Data values are only needed in single main routine, so define them as local to that routine.
3. Processing ...
a) initialize value of N , also variable that will hold N^N , both get initialized to 1
b) while loop
terminates when value in variable for N^N is "too large"
body of loop involves
calculating and printing out all those functions of N ;

incrementing N ready for next iteration;

Implementation

The implementation is again straightforward. The code would be:

```
#include <iostream.h>
#include <math.h>
/*
    Program to print lists of values ...

N    log(N)    N*log(N)    N^2    N^3    2^N    N^N

for N = 1, 2, 3, ...
values to be printed while N^N is not "too large".

*/

int main()
{
    const char TAB = '\t';
    // Used to help tidy output a little
    const double TOO_LARGE = 1000000000000000.0;
    // Arbitrary large value
    double N, LGN, N_LGN, NSQ, NCUBE, TWO_POW_N, N_POW_N;
    N = 1.0;
    N_POW_N = 1.0;

    while (N_POW_N < TOO_LARGE) {
        NSQ = N*N; NCUBE = N*NSQ;
        LGN = log(N); N_LGN = N*LGN;
        TWO_POW_N = pow(2.0,N);
        N_POW_N = pow(N,N);
        cout << N << TAB << LGN << TAB << N_LGN << TAB <<
            NSQ << TAB << NCUBE <<
            TAB << TWO_POW_N << TAB <<
            N_POW_N << endl;
        N = N + 1.0;
    }
}
```

Note the lines with multiple statements, e.g. `NSQ = N*N; NCUBE = N*NSQ;`. This is perfectly legal. Statements are terminated by semicolons, not newlines. So you can have statements that extend over several lines (like the monstrous output statement), or you can have several statements to a line.

The program produced the following output:

1	0	0	1	1	2	1	
2	0.693147		1.386294		4	8	4
3	1.098612		3.295837		9	27	8
4	1.386294		5.545177		16	64	16
5	1.609438		8.04719		25	125	32
6	1.791759		10.750557		36	216	64

7	1.94591	13.621371	49	343	128	823543
...						
...						
13	2.564949	33.344342	169	2197	8192	3.0287e+14
14	2.639057	36.946803	196	2744	16384	1.1112e+16

By the time $N=14$, N^N is already a ridiculously large number!

Algorithms (i.e. prescriptions of how to solve problems) whose run times increase at rates like 2^N , or $N!$, or N^N are really not much use. Such "exponential" algorithms can only be used with very small sets of data. There are some problems for which the only known exact algorithms are exponential. Such problems generally have to be solved by ad hoc approximate methods that usually work but don't always work.

Even with the use of tabs to try to get output lined up, the printed results were untidy! You will later learn about additional capabilities of the `cout` object that can be used to tidy the output (Chapter 9.7).

7.3 BLOCKS

In the last example program, the variables `N` and `N_pow_N` were initialized outside the while loop; all the others were only used inside the body of the while loop and weren't referenced anywhere else.

If you have a "compound statement" (i.e. a { begin bracket, several statements, and a closing } end bracket) where you need to use some variables not needed elsewhere, you can define those variables within the compound statement. Instead of having the program like this (with all variables declared at start of the main function):

```
int main()
{
    const char TAB = '\t';
    const double TOO_LARGE = 1000000000000000.0;
    double N, LGN, N_LGN, NSQ, NCUBE, TWO_POW_N, N_POW_N;
    N = 1.0;
    ...
    while (N_POW_N < TOO_LARGE) {
        NSQ = N*N; NCUBE = N*NSQ;
        ...
    }
}
```

You can make it like this, with only `N` and `N_pow_N` defined at start of program, variables `NSQ` etc defined within the body of the while loop:

```
void main()
{
    const char TAB = '\t';
    const double TOO_LARGE = 1000000000000000.0;
    double N, N_POW_N;
    N = 1.0;
```

```

...
while (N_POW_N < TOO_LARGE) {
    double LGN, N_LGN, NSQ, NCUBE, TWO_POW_N;
    NSQ = N*N; NCUBE = N*NSQ;
    ...
}
}

```

It is usually sensible to define variables within the compound statement where they are used; the definitions and use are closer together and it is easier to see what is going on in a program.

Defining a variable within a compound statement means that you can only use it between that compound statement's { begin bracket and } end bracket. You can't use the variable anywhere else. Because they had to be initialized and used in the test controlling the iterative loop, the variables `N` and `N_pow_N` had to be defined at the start of the program; they couldn't be defined within the compound statement.

Compound statements that contain definitions of variables are called "blocks". The body of a function forms a block. Other blocks can be defined inside as just illustrated.

***Where should
variables be defined***

In all the examples so far the variables have been defined at the start of the block before any executable statements. This is required in C (and in languages like Pascal and Modula2). C++ relaxes the rules a little and allows variables to be defined in the body of a block after executable statements. Later examples will illustrate the more relaxed C++ style.

Scope

The part of the text of the program source text where a variable can be referenced is called the "scope" of that variable. Variables defined in a block can only be referenced in statements following the definition up to the end of block bracket.

7.4 "BOOLEAN" VARIABLES AND EXPRESSIONS

7.4.1 True and False

In C/C++, the convention is that

```

the (numeric) value 0 (zero) means false,
any non-zero numeric value means true.

```

This is the reason why you will see "funny looking" while loops:

```

int n;
// read some +ve integer specify number of items to process
cin >> n;
while(n) {
    ...
    ...
    n = n - 1;
}

```


The `while(n)` test doesn't involve a comparison operator. What it says instead is "keep doing the loop while the value of `n` is non-zero".

Representing the true/false state of some condition by a 1 or a 0 is not always clear. For example, if you see code like:

```
int married;
...
...
married = 1;
```

It is not necessarily obvious whether it is updating a person's marital status or initiating a count of the number of times that that person has been married. Other languages, like Pascal, Modula2 and modern versions of FORTRAN have "Boolean" (or "logical") variables for storing true/false values and have constants `true` and `false` defined.

Code like

```
bool married;
...
...
married = true;
```

is considerably clearer than the version with integers.

C never defined a standard "Boolean" data type. So, every programmer working with C added their own version. The trouble was there were several different ways that the boolean type (and the constants `true` and `false`) could be added to C. Programmers used different mechanisms, and these different mechanisms got into library code.

*C "hacks" for
boolean types*

One way used by some programmers was to use `#define` macro statements. These macro statements simply instruct the compiler to replace one bit of text by another. So you could have

```
#define Boolean int
#define True 1
#define False 0
```

The compiler would then substitute the word `int` for all occurrences of `Boolean` etc. So you could write:

```
Boolean married;
...
married = True;
```

and the compiler would switch this to

```
int married;
...
married = 1;
```

before it tried to generate code.

Another programmer working on some other code for the same project might have used a different approach. This programmer could have used a "typedef" statement to tell the compiler that `boolean` was the name of a data type, or they might have used an "enumeration" to specify the terms { `false`, `true` }. (Typedefs are explained later in Chapter 11; enumerated types are introduced in Chapter 16). This programmer would have something code like:

```
enum boolean { false, true };
...
married = true;
```

Of course, it all gets very messy. There are *Booleans*, and *booleans*, and *True* and *true*.

*bool type in new C++
standard*

The standards committee responsible for the definition of C++ has decided that it is time to get rid of this mess. The proposed standard for C++ makes `bool` a built in type that can take either of the constant values `true` or `false`. There are conversions defined that convert 0 values (integer 0, "null-pointers") to `false`, and non-zero values to `true`.

A compiler that implements the proposed standard will accept code like:

```
bool married;
...
...
married = true;
```

Most compilers haven't got this implemented yet.

The older C hacks for `#define Boolean int`, or `typedef int boolean`, or `enum boolean` etc are everywhere. For years to come, older code and libraries will still contain these anachronistic forms.

7.4.2 Expressions using "AND"s and "OR"s

The simple boolean expressions that can be used for things like controlling while loops are:

1. a test on whether a variable is 0, false, or non zero, true; e.g.

```
while(n) ...
```

2. a test involving a comparison operator and either a variable or a constant; e.g

```
while(sum < 1000) ...
```

```
while(j >= k) ...
```

Sometimes you want to express more complex conditions. (The conditions that control `while` loops should be simple; the more complex boolean expressions are more likely to be used with the `if` selection statements covered in Chapter 8.)

For example, you might have some calculation that should be done if a double value `x` lies between 0.0 and 1.0. You can express such a condition by combining two simple boolean expressions:

```
(x > 0.0) && (x < 1.0)
```

The requirement for `x` being in the range is expressed as "x is greater than 0.0 AND x is less than 1.0". The AND operation is done using the `&&` operator (note, in C++ a single `&` can appear as an operator in other contexts – with quite different meanings).

The "&&" AND operator

There is a similar OR operator, represented by two vertical lines `||`, that can combine two simpler boolean expressions. The expression:

The "||" OR operator

```
(error_estimate > 0.0001) || (iterations < 10)
```

is true if either the value of `error_estimate` exceeds its 0.0001 OR the number of iterations is less than 10. (Again, be careful; a single vertical line `|` is also a valid C++ operator but it has slightly different meaning from `||`.)

Along with an AND operator, and an OR operator, C++ provides a NOT operator. Sometimes it is easier to express a test in a sense opposite that required; for example, in the following code the loop is executed while an input character is *not* one of the two values permitted:

The "!" NOT operator

```
char ch;
...
cout << "Enter patient's gender (M for Male or"
      " F for Female)" << endl;
cin >> ch;
while(!((ch == 'M') || (ch == 'F'))) {
    cout << "An entry of M or F is required here. "
          "Please re-enter gender code" << endl;
    cin >> ch;
}
```

The expression `((ch == 'M') || (ch == 'F'))` returns true if the input character is either of the two allowed values. The NOT operator, `!`, reverses the truth value – so giving rise to the correct condition to control the input loop.

Expressions involving AND and OR operators are evaluated by working out the values of the simple subexpressions one after the other reading the line from left to right until the result is known (note, as explained below, AND takes precedence over OR so the order of evaluation is not strictly left to right). The code generated for an expression like:

Evaluation of boolean expressions

```
A || B || C
```

(where A, B, and C are simple boolean expressions) would evaluate A and then have a conditional jump to the controlled code if the result was true. If A was not true, some code to evaluate B would be executed, again the result would be tested. The code to evaluate C would only be executed if both A and B evaluated to false.

Parentheses If you have complex boolean expressions, make them clear by using parentheses. Even complex expressions like

```
(A || B) && (C || D) && !E
```

are reasonably easy to understand if properly parenthesised. You can read something like

```
X || Y && Z
```

It does mean the same as

```
X || (Y && Z)
```

but your thinking time when you read the expression without parentheses is much longer. You have to remember the "precedence" of operators when interpreting expressions without parentheses.

Expressions without parentheses do all have meanings. The compiler is quite happy with something like:

```
i < limit && n != 0
```

It "knows" that you don't mean to test whether `i` is less than `limit && n`; you mean to test whether `i` is less than `limit` and if that is true you want also to test that `n` isn't equal to zero.

The meanings of the expressions are defined by operator precedence. It is exactly the same idea as how you learnt to interpret arithmetic expressions in your last years of primary school. Then you learnt that multiply and divide operators had precedence over addition and subtraction operators.

The precedence table has just been extended. Part of the operator precedence table for C++ is:

Operator precedence	<code>++, --</code>	<i>increment and decrement operators</i>
	<code>!</code>	<i>NOT operator</i>
	<code>*, /, %</code>	<i>multiply, divide, and modulo</i>
	<code>+, -</code>	<i>addition and subtraction</i>
	<code><, <=, >, >=</code>	<i>greater/less comparisons</i>
	<code>==, !=</code>	<i>equality, inequality tests</i>
	<code>&&</code>	<i>AND operator</i>
	<code> </code>	<i>OR operator</i>
	<code>=</code>	<i>assignment operator</i>

You can think of these precedence relations as meaning that the operators higher in the table are "more important" and so get first go at manipulating the data. (The `++` increment and `--` operators are explained in section 7.5.)

Because of operator precedence, a statement like the following can be unambiguously interpreted by a compiler:

```
res = n == m > 7*++k;
```

(It means that `res` gets a 1 (true) or 0 (false) value; it is true if the value of `n` (which should itself be either true or false) is the same as the true or false result obtained when checking whether the value of `m` exceeds 7 times the value of `k` after `k` has first been incremented!)

Legally, you can write such code in C++. Don't. Such code is "write only code". No one else can read it and understand it; everyone reading the code has to stop and carefully disentangle it to find out what you meant.

If you really had to work out something like that, you should do it in steps:

```
++k;
int temp = m > 7*k; // bool temp if your compiler has bool
res = n == temp;
```

Readers should have no problems understanding that version.

There are many traps for the unwary with complex boolean expressions. For example, you might be tempted to test whether `x` was in the range 0.0 to 1.0 inclusive by the following: **Caution**

```
0.0 <= x <= 1.0
```

This is a perfectly legal C++ test that will be accepted by the compiler which will generate code that will execute just fine. Curiously, the code generated for this test returns true for `x = -17.5`, `x = 0.4`, `x = 109.7`, and in fact for any value of `x`!

The compiler saw that expression as meaning the following:

```
compare x with 0.0
  getting result 1 if x >= 0.0, or 0 if x < 0.0
compare the 0 or 1 result from last step with 1.0
  0 <= 1 gives result 1, 1 <= 1 also gets result 1
```

So, the code, that the compiler generated, says "true" whatever the value of `x`.

Another intuitive (but **WRONG**) boolean expression is the following (which a student once invented to test whether a character that had been read was a vowel)

```
ch == 'a' || 'e' || 'i' || 'o' || 'u'
```

Once again, this code is perfectly legal C++. Once again, its meaning is something quite different from what it might appear to mean. Code compiled for this expression will return true whatever the value in the character variable `ch`.

The `==` operator has higher precedence than the `||`. So this code means

```
(ch == 'a') || 'e' || ...
```

Now this is true when the character read was 'a'. If the character read wasn't 'a', the generated code tests the next expression, i.e. the 'e'. Is 'e' zero (false) or non-zero (true)? The value of 'e' is the numeric value of the ASCII code that represents this character (101); value 101 isn't zero, so it must mean true.

The student wasn't much better off when she tried again by coding the test as:

```
ch == ('a' || 'e' || 'i' || 'o' || 'u')
```

This version said that no character was a vowel. (Actually, there is one character that gets reported as a vowel, but it is the control character "start of header" that you can't type on a keyboard).

The student could have expressed the test as follows:

```
((ch == 'a') || (ch == 'e') || (ch == 'i') || (ch == 'o'))
 || (ch == 'u')
```

It may not be quite so intuitive, but at least it is correct code.

7.5 SHORT FORMS: C/C++ ABBREVIATIONS

As noted in other contexts, the C and C++ languages have lots of abbreviated forms that save typing. There are abbreviations for some commonly performed arithmetic operations. For example, one often has to add one to a counter:

```
count_customers = count_customers + 1;
```

The += operator In C and C++, this could be abbreviated to

```
count_customers += 1;
```

The "++" increment operator

or even to

```
count_customers++;
```

Actually, these short forms weren't introduced solely to save the programmer from typing a few characters. They were meant as a way that the programmer could give hints that helped the early C compilers generate more efficient code.

An early compiler might have translated code like

```
temp = temp + 5;
```

into a "load instruction", an "add instruction", and a "store instruction". But the computer might have had an "add to memory instruction" which could be used instead of the three other instructions. The compiler might have had an alternative coding template that recognized the += operator and used this add to memory instruction when generating code for things like:

```
temp += 5;
```

Similarly, there might have been an "increment memory" instruction (i.e. add 1 to contents of memory location) that could be used in a compiler template that generated code for a statement like

```
temp++;
```

Initially, these short forms existed only for addition and subtraction, so one had the four cases

<code>++</code>	increment
<code>+= <i>Expression</i></code>	add value of expression
<code>--</code>	decrement (i.e. reduce by 1)
<code>-= <i>Expression</i></code>	subtract value of expression

Later, variants of `+=` and `-=` were invented for things like `*` (the multiplication operator), `/` (the division operator), and the operators used to manipulate bit-patterns ("BIT-OR" operator, "BIT-AND" operator, "BIT-XOR" operator etc --- these will be looked at later, Chapter 18). So one can have:

```
x *= 4;    // Multiply x by 4, short form for x = x * 4
y /= 17;   // Short form for y = y/17;
```

(Fortunately, the inventors of the C language didn't add things like `a**` or `b//` - because they couldn't agree what these formulae would mean.)

These short forms are OK when used as simple statements:

```
N++;
```

is OK instead of (maybe even better than)

```
N = N + 1;

Years += Half_life;
```

is OK instead of (maybe even better than)

```
Years = Years + Half_life;
```

The short forms can be used as parts of more complex expressions:

```
xval = temp + 7 * n++;
```

this changes both `xval` and `n`. While legal in both C and C++, such usage should be avoided (statements with embedded increment, `++`, or decrement, `--`, operators are difficult to understand and tend to be sources of error).

Expressions with embedded `++` and `--` operators are made even more complex and confusing by the fact that there are two versions of each of these operators.

The `++` (or `--`) operator can come before the variable it changes, `++x`, or after the variable, `x++`. Now, if you simply have a statement like

```
x++;           or           ++x;
```

*Pre-op and post-op
versions of ++ and --*

it doesn't matter which way it is written (though `x++` is more common). Both forms produce identical results (`x` is incremented).

But, if you have these operators buried in more complex expressions, then you will get different results depending on which you use. The prefix form (`++x`) means "*increment the value of `x` and use the updated value*". The postfix form (`x++`) means "*use the current value of `x` in evaluating the expression and also increment `x` afterwards*". So:

```
n = 15;
temp = 7 * n++;
cout << "temp  " << temp << endl;
cout << "n  " << n << endl;
```

prints temp 105 and n 16 while

```
n = 15;
temp = 7 * ++n;
cout << "temp  " << temp << endl;
cout << "n  " << n << endl;
```

prints temp 112 and n 16.

General advice

Only use short form operators `++`, `--`, `+=`, `-=` etc in simple statements with a single operator. The following are OK

```
i++;          years += half_life;
```

anything more complex should be avoided until you are much more confident in your use of C++.

Why C++?

Now you understand why language is called C++:

take the C language and add a tiny bit to get one better.

7.6 DO ...WHILE

The standard while loop construct has a variation that is useful if you know that a loop must always be executed at least once.

This variation has the form

```
do
    statement
while (expression);
```

Usually, the body of the loop would be a compound statement rather than a simple statement; so the typical "do" loop will look something like:

```
do {
    ...
    ...
} while (expression);
```


The expression can be any boolean expression that tests values that get updated somewhere in the loop.

A contrived example –

```
// Loop getting numbers entered by user until either five
// numbers have been entered or the sum of the numbers
// entered so far exceeds limit
const int kLIMIT = 500;
int entry_count = 0;
int sum = 0;
do {
    cout << "Give me another number ";
    int temp;
    cin >> temp;
    entry_count++;
    sum += temp;
} while ((entry_count < 5) && (sum < kLIMIT));
```

7.7 FOR LOOP

Often, programs need loops that process each data item from some known fixed size collection. This can be handled quite adequately using a standard while loop –

```
// Initialize counter
int loop_count = 0;
// loop until limit (collection size) reached
while(loop_count < kCOLLECTION_SIZE) {
    // Do processing of next element in collection
    ...
    // Update count of items processed
    loop_count++;
}
```

The structure of this counting loop is: initialize counter, loop while counter less than limit, increment counter at the end of the body of the loop.

It is such a common code pattern that it deserves its own specialized statement – the `for` statement.

```
int loop_count;
for(loop_count=0;
    loop_count < kCOLLECTION_SIZE;
    loop_count++) {
    // Do processing of next element in collection
    ...
}
```

The `for` statement has the following structure:

```
for( First part, do things like initialize counts ;
     Second part, the termination test ;
     Third part, do things like updating
```

```

                                counters (done after body of loop
                                has been executed)
                                )
                                Statement of what is to be done ;

```

There are three parts to the parenthesised construct following the keyword `for`. The first part will be a simple statement (compound statements with `begin {` and `}` end brackets aren't allowed); normally, this statement will initialize the counter variable that controls the number of times the loop is to execute. The second part will be a boolean expression; the loop is to continue to execute as long as this expression is true. Normally, this second part involves a comparison of the current value of the loop control variable with some limit value. Just as in a `while` loop, this loop termination test is performed before entering the body of the loop; so if the condition for termination is immediately satisfied the body is never entered (e.g. `for(i=5; i<4; i++) { ... }`). The third part of the `for(...;...;...)` contains expressions that are evaluated after the body has been executed. Normally, the third part will have code to increment the loop control variable.

A `for` loop "controls" execution of a statement; usually this will be a compound statement but sometimes it will be a simple statement (generally a call to a function). The structures are:

```

for(...; ...; ...)          for(i=0;i<10;i++)
    statement;              sum += i;

```

(note the semicolon *after* the controlled statement.) and

```

for(...; ...; ...) {        for(i=0;i<10;i++) {
    statement;              cout << i << " ";
    statement;              ...
    ...                     ...
}                           }

```

You can have loops that have the loop control variable decreasing –

```

for(j = 100; j > 0; j--) {
    // work back down through a collection of data
    // elements starting from top
    ...
}

```

***Don't change the
loop control variable
in the body of the
code***

If you are using a `for` as a counting loop, then you should not alter the value of the control variable in the code forming the body of the loop. It is legal to do so; but it is very confusing for anyone who has to read your code. It is so confusing that you will probably confuse yourself and end up with a loop that has some bug associated with it so that under some circumstances it doesn't get executed the correct number of times.

***Defining a loop
control variable in
the for statement***

As noted earlier, section 7.3, variables don't have to be defined at the start of a block. If you know you are going to need a counting loop, and you want to follow convention and use a variable such as `i` as the counter, you can arrange your code like this where `i` is defined at the start of the block:

```
int main()
{
    int i, n;
    cout << "Enter number of times loop is to execute";
    cin >> n;
    for(i=0; i < n; i++) {
        // loop body with code to do something
        // interesting
        ...
    }
    cout << "End of loop, i is " << i << endl;
    ...
}
```

or you can have the following code where the loop control variable `i` is defined just before the loop:

```
int main()
{
    int n;
    cout << "Enter number of times loop is to execute";
    cin >> n;
    int i;
    for(i=0; i < n; i++) {
        ...
    }
    cout << "End of loop, i is " << i << endl;
    ...
}
```

or you can do it like this:

```
int main()
{
    int n;
    cout << "Enter number of times loop is to execute";
    cin >> n;
    for(int i=0; i < n; i++) {
        ...
    }
    cout << "End of loop, i is " << i << endl;
    ...
}
```

A loop control variable can be defined in that first part of the `for(...;...;...)` statement. Defining the control variable inside the `for` is no different from defining it just before the `for`. The variable is then defined to the end of the enclosing block. (There are other languages in the Algol family where loop control variables "belong" to their loops and can only be referenced within the body of their loop. The new standard for C++ suggests changing the mechanism so that the loop control variable does in fact belong with the loop; most compilers don't yet implement this.)

Note area of possible language change

You are not restricted in the complexity of the code that you put in the three parts of the `for(...;...;...)`. The initialization done in the first part can for example involve a function call: `for(int i = ReadStartingValue(); ...; ...)`. The termination test can be something really elaborate with a fearsome boolean expression with lots of `&&` and `||` operators.

You aren't allowed to try to fit multiple statements into the three parts of the `for` construct. Statements end in semicolons. If you tried to fit multiple statements into the `for` you would get something like

```
// Erroneous code in initialization part of for
for( sum = 0; i= 1;
    ...; // termination test
        // Erroneous code in update part of for
        i++; sum+= i*val; ...)
    ...
```

The semicolons after the extra statements would break up the required pattern `for(... ; ... ; ...)`.

The comma operator

The C language, and consequently the C++ language, has an alternative way of sequencing operations. Normally, code is written as `statement; statement; statement;` etc. But you can define a sequence of expressions that have to be evaluated one after another; these sequences are defined using the comma `,` operator to separate the individual expressions:

```
expression , expression, expression
```

Another oddity in C (and hence C++) is that the "assignment statement" is actually an expression. So the following are examples of single statements:

```
i = 0, max = SHRT_MAX, min = SHRT_MIN, sum = 0;
i++, sum += i*val;
```

The first statement involves four assignment expressions that initialize `i` and `sum` to 0 and `max` and `min` to the values of the constants `SHRT_MAX` etc. The second statement has two expressions updating the values of `i` and `sum`. (It was the comma operator that caused the problems noted in section 6.4 with the erroneous input statement `cin >> xcoord, ycoord;`. This statement actually consists of two comma separated expressions – an input expression and a "test the value of" expression.)

Although you aren't allowed to try to fit multiple statements into the parts of a `for`, you are permitted to use these comma separated sequences of expressions. So you will frequently see code like:

```
for(i=0, j=1;                               // initialization part
    i<LIMIT;                                // termination test
    i++, j = ++j % 5) // update part
{
    ...
}
```

(It isn't totally perverse code. Someone might want to have a loop that runs through a set of data elements doing calculations and printing results for each; after every 5th output there is to be a newline. So, the variable `i` counts through the elements in the collection; the variable `j` keeps track of how many output values have been printed on the current line.)

Because you can use the comma separated expressions in the various parts of a `for`, you can code things up so that all the work is done in the `for` statement itself and the body is empty. For example, if you wanted to calculate the sum and product of the first 15 integers you could write the code the simple way:

```
int i, sum = 0, product = 1;
for(i = 1; i<= 15; i++) {
    sum += i;
    product *= i;
}
cout << "Sum " << sum << ", product " << product << endl;
```

or you can be "clever":

```
int i, sum, product ;
for(sum = 0, product = 1, i = 1;
    i<= 15;
    sum += i, product *= i, i++) ;
cout << "Sum " << sum << ", product " << product << endl;
```

A form like:

```
for(...; ...; ...);
```

(with the semicolon immediately after the closing parenthesis) is legal. It means that you want a loop where nothing is done in the body because all the work is embedded in the `for(...;...;...)`. (You *may* get a "warning" from your compiler if you give it such code; some compiler writer's try to spot errors like an extraneous semicolon getting in and separating the `for(...;...;...)` from the body of the loop it is supposed to control.)

While you often see programs with elaborate and complex operations performed inside the `for(...;...;...)` (and, possibly, no body to the loop), you shouldn't write code like this. Remember the "KISS principle" (Keep It Simple Stupid). Code that is "clever" requires cleverness to read, and (cleverness)² to write correctly.

A `while` loop can do anything that a `for` loop can do.

Conversely, a `for` loop can do anything a `while` loop can do.

Instead of the code given earlier (in 7.2.2):

```
r = x / 2.0;
while(fabs(x - r*r) > SMALLFRACTION*x) {
    cout << r << endl;
    r = 0.5 *(x / r + r);
}

cout << "Number was : " << x << ", root is "
    << r << endl;
```

You could make the code as follows:

```

r = x / 2.0;
for( ;           // an empty initialization part
    fabs(x - r*r) > SMALLFRACTION*x; // Terminated?
    ) {          // Empty update part
    cout << r << endl;
    r = 0.5 *(x / r + r);
}
cout << "Number was : " << x << ", root is "
      << r << endl;

```

or, if you really want to be perverse, you could have:

```

for(r = x / 2.0;
    fabs(x - r*r) > SMALLFRACTION*x;
    r = 0.5 *(x / r + r))
    cout << r << endl;
cout << "Number was : " << x << ", root is "
      << r << endl;

```

(It is legal to have empty parts in a `for(...;...;...)`, as illustrated in the second version.)

Most readers prefer the first version of this code with the simple `while` loop. Write your code to please the majority of those that must read it.

7.8 BREAK AND CONTINUE STATEMENTS

There are two other control statements used in association with loop constructs. The `break` and `continue` statements can appear in the body of a loop. They appear as the action parts of `if` statement (section 8.3). The conditional test in the `if` will have identified some condition that requires special processing.

break A `break` statement terminates a loop. Control is transferred to the first statement following the loop construct.

continue A `continue` statement causes the rest of the body of the loop to be omitted, at least for this iteration; control is transferred to the code that evaluates the normal test for loop termination.

Examples of these control statements appear in later programs where the code is of sufficient complexity as to involve conditions requiring special processing (e.g. use of `continue` in function in 10.9.1).

EXERCISES

1. Implement versions of the "square root" program using the "incorrect" tests for controlling the iterative loop. Try finding values for x for which the loop fails to terminate. (Your IDE system will have a mechanism for stopping a program that isn't terminating properly. One of the command keys will send a stop signal to the program.)

Check the IDE documentation to find how to force termination before starting the program.)

If your compiler supports "long doubles" change the code of your program to use the long double data type and rerun using the same data values for which the original program failed. The modified program may work. (The long double type uses more bits to represent the mantissa of a floating point number and so is not as easily affected by round off errors.)

2. Write a program that prints the values of a polynomial function of x at several values of x . The program should prompt for the starting x value, the increment, and the final x value.

A polynomial function like $6x^3 - 8x^2 - 3x + 4.5$ can be evaluated at a particular x value using the formula:

```
val = 6.0 * pow(x, 3.0) - 8.0*pow(x,2.0) -3.0*x + 4.5;
```

or

```
val = 6.0 * x * x * x - 8.0 * x *x -3.0 *x + 4.5;
```

or

```
val = ((6.0*x - 8.0)*x - 3.0)*x + 4.5;
```

The first is the most costly to evaluate because of the `pow()` function calls. The last is the most efficient (it involves fewer multiplication operations than the second version).

(For this exercise, use a fixed polynomial function. List its values at intervals of 0.5 for x in range -5.0 to +5.0.)

3. Generalize the program from exercise 2 to work with any polynomial of a given maximum degree 4, i.e. a function of the form $c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0$ for arbitrary values of the coefficients c_4, c_3, c_2, c_1 , and c_0 .

The formula for evaluating the polynomial at a given value of x is

```
val = (((c4 * x + c3)*x + c2)*x + c1)*x + c0
```

The program is to prompt for and read the values of the coefficients and then, as in exercise 2 it should get the range of x values for which the polynomial is to be evaluated.

4. An object dropped from a height is accelerated to earth by gravity. Its motion is determined by Newton's laws. The formulas for its velocity and distance travelled are:

```
v = 32.0 * t; // velocity in feet per second
s = 16.0 * t * t; // distance fallen in feet
```

Write a program that reads the height from which the object is dropped and which prints a table showing the object's velocity and height at one second intervals. The loop

printing details should terminate when the distance fallen is greater than or equal to the specified height.

5. Write a program that "balances transactions on a bank account".

The program is to prompt for and read in the initial value for the funds in the account. It is then to loop prompting for and reading transactions; deposits are to be entered as positive values, withdrawals as negative values. Entry of the value 0 (zero) is to terminate the loop. When the loop has finished, the program is to print the final value for the funds in the account.

8 Selection

8.1 MAKING CHOICES

Loop constructs permit the coding of a few simple numerical calculations; but most programs require more flexibility. Different input data usually have to be interpreted in different ways. As a very simple example, imagine a program producing some summary statistics from lots of data records defining people. The statistics required might include separate counts of males and females, counts of citizens and resident-alien and visitors, counts of people in each of a number of age ranges. Although similar processing is done for each data record, the particular calculation steps involved will depend on the input data. Programming languages have to have constructs for *selecting* the processing steps appropriate for particular data.

The modern Algol family languages have two kinds of selection statement:

"if" statements

and

"switch" statements (or "case" statements)

There are typically two or three variations on the "if" statement. Usually, they differ in only minor ways; this is possibly why beginners frequently make mistakes with "if"s. Beginners tend to mix up the different variants. Since "if"s seem a little bit error prone, the "switch" selection statement will be introduced first.

switch statement

The C/C++ switch statement is for selecting one processing option from among a choice of several. One can have an arbitrary number of choices in a switch statement.

Each choice has to be given an identifying "name" (which the compiler has to be able to convert into an integer constant). In simple situations, the different

choices are not given explicit names, the integer numbers will suffice. Usually, the choice is made by testing the value of a variable (but it can be an expression).

Consider a simple example, the task of converting a date from numeric form <day> <month> <year>, e.g. 25 12 1999, to text December 25th 1999. The month would be an integer data value entered by the user:

```
int day, month, year;
cout << "Enter date as day, month, and year" << endl;
cin >> day >> month >> year;
...
```

The name of the month could be printed in a switch statement that used the value of "month" to select the appropriate name. The switch statement needed to select the month could be coded as:

```
...
switch(month) {
case 1:
    cout << "January ";
    break;
case 2:
    cout << "February ";
    break;
...
...
case 12:
    cout << "December ";
    break;
}
```

Parts of a switch statement

A switch statement is made up from the following parts:

switch

The keyword switch.

(month)

A parenthesised expression that yields the integer value that is used to make the choice. Often, as in this example, the expression simply tests the value of a variable.

{

The { 'begin block' bracketing character.

case

The keyword case. This marks the start of the code for one of the choices.

1

The integer value, as a simple integer or a named constant, associated with this choice.

```
:
```

A colon, ':', punctuation marker.

```
cout << "January ";
```

The code that does the special processing for this choice.

```
break;
```

The keyword `break` that marks the end of the code section for this choice (or case).

```
case 2:
```

The code for the next case.

```
cout << "February ";
break;
```

Similar code sections for each of the other choices.

```
}
```

Finally, the `}` 'end block' bracket to match the `{` at the start of the set of choices.

Be careful when typing the code of a switch; in particular, make certain that you pair your `case ... break` keywords.

You see, the following is legal code, the compiler won't complain about it:

```
case 1:
    cout << "January ";
case 2:
    cout << "February ";
    break;
```

But if the month is 1, the program will execute the code for both case 1 and case 2 and so print "January February".

C/C++ allows "case fall through" (where the program continues with the code of a second case) because it is sometimes useful. For example, you might have a program that uses a `switch` statement to select the processing for a command entered by a user; two of the commands might need almost the same processing with one simply requiring an extra step : *"case fall through"*

```
/* Command 'save and quit' --- save data and close up */
case 101:
    SaveData();
    /* CASE FALLTHROUGH REQUIRED */
```

```

/* Command 'quit' --- close up */
case 100:
    CloseWindow();
    DisconnectModem();
    break;

```

(If you intend to get "case fall through", make this clear in a comment.)

Another example where case fall through might be useful is a program to print the words of a well known yuletide song:

```

cout << "On the " << day_of_christmas
    << " day of Christmas, " << endl;
cout << "My true love gave to me " << endl;

switch(day_of_christmas) {
case 12:
    cout << "Twelve lords a leaping";
case 11:
    ...
    ...
case 3:
    cout << "Three French hens" << endl;
case 2:
    cout << "Two turtle doves, " << endl;
    cout << "and " << endl;
case 1:
    cout << "A partridge in a pear tree";
};

```

The "case labels" (the integers identifying the choices) don't have to be in sequence, so the following is just as acceptable to the compiler (though maybe confusing to someone reading your code):

```

switch(month) {
case 4:
    cout << "April ";
    break;
case 10:
    cout << "October ";
    break;
    ...
case 2:
    cout << "February";
    break;
}

```

In this simple example, it wouldn't be necessary to invent named constants to characterize the choices – the month numbers are pretty intuitive. But in more complex programs, it does help to use named constants for case labels. You would have something like:

<i>Defining named constants for case labels</i>	<pre> #include <iostream.h> const int JAN = 1; </pre>
---	--

```

const int FEB = 2;
...
const int DEC = 12;

void main()
{
    int day, month, year;
    ...
    switch(month) {
case JAN:
    cout << "January ";
    break;
case FEB:
    cout << "February ";
    break;
...
case DEC:
    cout << "December ";
    break;
    }
    ...

```

As well as printing the name of the month, the dates program would have to print the day as 1st, 2nd, 3rd, 4th, ..., 31st. The day number is easily printed --- *Another switch statement*

```

cout << day;

```

but what about the suffixes 'st', 'nd', 'rd', and 'th'?

Obviously, you could get these printed using some enormous case statement

```

switch(day) {
case 1: cout << "st "; break;
case 2: cout << "nd "; break;
case 3: cout << "rd ", break;
case 4: cout << "th "; break;
case 5: cout << "th "; break;
case 6: cout << "th "; break;
case 7: cout << "th "; break;
...
case 21: cout << "st "; break;
case 22: cout << "nd "; break;
...
case 30: cout << "th "; break;
case 31: cout << "st "; break;
}

```

Fortunately, this can be simplified. There really aren't 31 different cases that have to be considered. There are three special cases: 1, 21, and 31 need 'st'; 2 and 22 need 'nd'; and 3 and 23 need 'rd'. Everything else uses 'th'. *Combining similar cases*

The case statement can be simplified to:

```

switch (day) {
case 1:
case 21:

```

```

case 31:
    cout << "st ";
    break;
case 2:
case 22:
    cout << "nd ";
    break;
case 3:
case 23:
    cout << "rd ";
    break;
default:
    cout << "th ";
    break;
}

```

Where several cases share the same code, that code can be labelled with all those case numbers:

```

case 1:
case 21:
case 31:
    cout << "st ";
    break;

```

(You can view this as an extreme example of 'case fall through'. Case 1 --- do nothing then do everything you do for case 21. Case 21 --- do nothing, then do everything you do for case 31. Case 31, print "st".)

***Default section of
switch statement***

Most of the days simply need "th" printed. This is handled in the "default" part of the switch statement.

```

switch (day) {
case 1:
case 21:
case 31:
    cout << "st ";
    break;
...
...
default:
    cout << "th ";
    break;
}

```

The compiler will generate code that checks the value used in the switch test against the values associated with each of the explicit cases, if none match it arranges for the "default" code to be executed.

A default clause in a switch statement is often useful. But it is not necessary to have an explicit default; after all, the first example with the months did not have a default clause. If a switch does not have a default clause, then the compiler generated code will be arranged so as to continue with the next statement following after the end of the switch if the value tested by the switch does not match any of the explicit cases.

Usually a switch statement tests the value of an integer variable:

```
switch (month) {  
    ...  
}
```

and

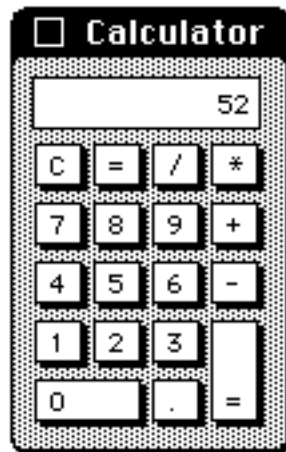
```
switch (day) {  
    ...  
}
```

But it can test the value of any integer expression.

A program can have something like `switch(n+7) { ... }` if this makes sense.

8.2 A REALISTIC PROGRAM: DESK CALCULATOR

Loop and selection constructs are sufficient to construct some quite interesting programs. This example is still fairly simple. The program is to simulate the workings of a basic four function calculator



Specification:

The calculator program is to:

- 1 Display a "current value".
- 2 Support the operations of addition, subtraction, multiplication, and division, operations are to be invoked by entering the character '+', '-', '*', or '/' in response to a prompt from the program.
- 3 When an operator is invoked, the program is to request entry of a second number that is to be combined with the current value; when given a number,

the program is to perform the specified calculation and then display the new current value.

- 4 The program is to accept the character 'C' (entered instead of an operator) as a command to clear (set to zero) the "current value".
- 5 The program is to terminate when the character 'Q' is entered instead of an operator.

Example operation of required calculator program:

```

Program initially displaying          0
prompt for operator                  +
                                     '+' character entered
prompt for number                    1.23
                                     addition operation 0+1.23
current value displayed              1.23
prompt for operator                  *
                                     '*' character entered
prompt for number                    6.4
                                     multiplication operation performed
current value displayed              7.872
prompt for operator                  C
                                     clear operation
current value displayed              0
prompt for operator                  Q
                                     program terminates

```

Design and Implementation

Preliminary design: Programs are designed in an iterative fashion. You keep re-examining the problem at increasing levels of detail.

First iteration through the design process

The first level is simple. The program will consist of

- some initialization steps
- a loop that terminates when the 'Q' (Quit) command is entered.

Second iteration through the design process

The second iteration through the design process would elaborate the "setup" and "loop" code:

- some initialization steps:
 - set current value to 0.0
 - display current value
 - prompt user, asking for a command character
 - read the command character
- loop terminating when "Quit" command entered
 - while command character is not Q for Quit do the following
 - (process command --- if input is needed ask for number then do
 - (operation
 - (display result
 - (prompt the use, asking for a command character
 - (read the command character

The part about prompting for a number and performing the required operation still needs to be made more specific. The design would be extended to something like the following:

*Third iteration
through the design
process*

```

set current value to 0.0
display current value
prompt user, asking for a command character
read the command character

while command character is not Q for Quit do
    examine command character
    'C'          set current number to zero
                finished
    '+'          ask for a number,
                read number,
                do addition
                finished
    '-'          ask for a number,
                read number,
                do subtraction
                finished
    '*'          ask for a number,
                read number,
                do multiplication,
                finished
    '/'          ask for a number,
                read number,
                do division,
                finished

    display result
    prompt for another command,
    read command character

```

The preliminary design phase for a program like this is finished when you've developed a model of the various processing steps that must be performed.

Detailed design

You must then consider what data you need and how the data variables are to be stored. Later, you have to go through the processing steps again, making your descriptions even more detailed.

Only three different items of data are needed:

Data

- "the current displayed value" – this will be a double number;
 - "the command character" – a character,
- and
- another double number used when entering data.

These belong to the main program:

```

int main()
{
    double    displayed_value;
    double    new_entry;

```

```
char        command_character;
```

The while loop We can temporarily ignore the switch statement that processes the commands and concentrate solely on the while loop structure. The loop is to terminate when a 'Q' command has been entered. So, we can code up the loop control:

```
while (command_character != 'Q') {
    /* switch stuff to go here */
    ...
    /* get next command entered. */
    cout << "Value : " << displayed_value << endl;
    cout << "command>";
    cin >> command_character;
}
```

The loop continues while the command character is *NOT* 'Q'. Obviously we had better read another command character each time we go round the loop. We have to start the loop with some command character already specified.

Part of the setup code prompts for this initial command:

```
#include <iostream.h>
#include <stdlib.h>

int main()
{
    double        displayed_value;
    double        new_entry;
    char          command_character;

    displayed_value = 0.0;

    cout << "Calculator demo program" << endl
         << "----" << endl;

    cout << "Enter a command character at the '>' prompt"
         << endl;
    ...

    cout << "Value : " << displayed_value << endl;
    cout << "command>";
    cin >> command_character;
    while (command_character != 'Q') {
        /* Switch statement to go here */
        ...
        cout << "Value : " << displayed_value << endl;
        cout << "command>";
        cin >> command_character;
    }
    return 0;
}
```

The code outlined works even if the user immediately enters the "Quit" command!

**Pattern for a loop
processing input data**

The pattern

- setup code reads initial data value
- while loop set to terminate when specific data value entered
- next input value is read at the end of the body of the loop

is very common. You will find similar while loops in many programs.

Selection of the appropriate processing for each of the different commands can obviously be done with a switch statement: *Switch statement*

```
switch (command_character) {
    ...
}
```

Each of the processing options is independent. We don't want to share any code between them. So, each of the options will be in the form

```
case ... :
    statements
    break;
```

The case of a 'Q' (quit command) is handled by the loop control statement. The switch only has to deal with:

- 'C' clear command, set current value to zero;
- '+' addition, ask for input and do the addition;
- '-' subtraction, ask for input and do the subtraction;
- '*' multiplication, ask for input and do the multiplication;
- '/' division, ask for input and do the division.

Each of these cases has to be distinguished by an integer constant and the switch statement itself must test an integer value.

Fortunately, the C and C++ languages consider characters like 'C', '+' etc to be integers (because they are internally represented as small integer values). So the characters can be used directly in the `switch() { }` statement and for case labels.

```
switch(command_character) {
case 'C':
    displayed_value = 0.0;
    break;
case '+':
    cout << "number>";
    cin >> new_entry;
    displayed_value += new_entry;
    break;
case '-':
    cout << "number>";
```

```

        cin >> new_entry;
        displayed_value -= new_entry;
        break;
    case '*':
        cout << "number>";
        cin >> new_entry;
        displayed_value *= new_entry;
        break;
    case '/':
        cout << "number>";
        cin >> new_entry;
        displayed_value /= new_entry;
        break;
}

```

The code for the individual cases is straightforward. The clear case simply sets the displayed value to zero. Each of the arithmetic operators is coded similarly – a prompt for a number, input of the number, data combination.

Suppose the user enters something inappropriate – e.g. an 'M' command (some calculators have memories and 'M', memorize, and 'R' recall keys and a user might assume similar functionality). Such incorrect inputs should produce some form of error message response. This can be handled by a "default" clause in the switch statement.

```

default:
    cout << "Didn't understand input!";

```

If the user has done something wrong, it is quite likely that there will be other characters that have been typed ahead (e.g. the user might have typed "Hello", in which case the 'H' is read and recognized as invalid input – but the 'e', 'l', 'l' and the 'o' all remain to be read).

Removing invalid characters from the input

When something has gone wrong it is often useful to be able to clear out any unread input. The `cin` object can be told to do this. The usual way is to tell `cin` to ignore all characters until it finds something obvious like a newline character. This is done using the request `cin.ignore(...)`. The request has to specify the sensible marker character (e.g. `'\n'` for newline) and, also, a maximum number of characters to ignore; e.g.

```

cin.ignore(100, '\n');

```

This request gets "cin" to ignore up to 100 characters while it searches for a newline; that should be enough to skip over any invalid input.

With the default statement to report any errors (and clean up the input), the complete switch statement becomes:

```

switch(command_character) {
case 'C':
    displayed_value = 0.0;
    break;
case '+':
    cout << "number>";

```

```

        cin >> new_entry;
        displayed_value += new_entry;
        break;
    case '-':
        cout << "number>";
        cin >> new_entry;
        displayed_value -= new_entry;
        break;
    case '*':
        cout << "number>";
        cin >> new_entry;
        displayed_value *= new_entry;
        break;
    case '/':
        cout << "number>";
        cin >> new_entry;
        displayed_value /= new_entry;
        break;
    default:
        cout << "Didn't understand input!";
        cin.ignore(100, '\n');
}

```

Since there aren't any more cases after the `default` statement, it isn't necessary to pair the `default` with a `break` (it can't "fall through" to next case if there is no next case) But, you probably should put a `break` in anyway. You might come along later and add another case at the bottom of the switch and forget to put the `break` in.

8.3 IF

Switch statements can handle most selection tasks. But, they are not always convenient.

You can use a switch to select whether a data value should be processed:

```

// Update count of students who's assignment marks
// and examination marks both exceed 25
switch((exam_mark > 25) && (assignment_mark > 25)) {
case 1:
    good_student++;
    break;
case 0:
    // Yuk, ignore them
    break;
};

```

The code works OK, but it *feels* clumsy.

What about counting the numbers of students who get different grades? We would need a loop that read the total course marks obtained by each individual student and used this value to update the correctly selected counter. There would

be counters for students who had failed (mark < 50), got Ds (mark < 65), Cs (<75), Bs (<85) and As. The main program structure would be something like:

```
int main()
{
    int A_Count = 0, B_Count = 0, C_Count = 0,
        D_Count = 0, F_Count = 0;
    int mark;
    cin >> mark;
    while(mark >= 0) {
        code to select and update appropriate counter
        cin >> mark; // read next, data terminated by -1
    }
    // print counts
    ...
    ...
}
```

How could the selection be made?

Well, you *could* use a `switch` statement. But it would be pretty hideous:

```
switch(mark) {
case 0:
case 1:
case 2:
...
case 49:
    F_Count++;
    break;
case 50:
case 51:
...
case 64:
    D_Count++;
    break;
...
...
    break;
case 85:
case 86:
case 100:
    A_Count++;
    break;
}
```

Obviously, there has to be a better way.

if The alternative selection constructs use the `if ...` and `if ... else ...` statements. A simple `if` statement has the following form:

```
if(boolean expression)
    statement;
```

Example:

```
if((exam_mark > 25) && (assignment_mark > 25))
    good_student++;
```

Often, an `if` will control execution of a compound statement rather than a simple statement. Sometimes, even if you only require a simple statement it is better to put in the `{ begin bracket and }` end bracket:

```
if((exam_mark > 25) && (assignment_mark > 25)) {
    good_student++;
}
```

This makes it easier if later you need to add extra statements –

```
if((exam_mark > 25) && (assignment_mark > 25)) {
    cout << " ";
    good_student++;
}
```

(also, some debuggers won't let you stop on a simple statement controlled by an `if` but will let you stop in a compound statement). Note, you don't put a semicolon after the `}` end bracket of the compound statement; the `if` clause ends with the semicolon after a simple statement or with the `}` end bracket of a compound statement.

The `if ... else ...` control statement allows you to select between two *if ... else* alternative processing actions:

```
if(boolean expression)
    statement;
else
    statement;
```

Example:

```
if(gender_tag == 'F')
    females++;
else
    males++;
```

You can concatenate `if ... else` control statements when you need to make a *if ... else if ... else if ... else ...* selection between more than two alternatives:

```
if(mark<50) {
    F_Count++;
    cout << "another one bites the dust" << endl;
}
else
if(mark<65)
    D_Count++;
else
if(mark<75)
    C_Count++;
else
```

```

if(mark<85)
    B_Count++;
else {
    A_Count++;
    cout << "*****" << endl;
}

```

You have to end with an `else` clause. The statements controlled by different `if` clauses can be either simple statements or compound statements as required.

Nested ifs and related problems

You need to be careful when you want to perform some complex test that depends on more than one condition. The following code is OK:

```

if(mark<50) {
    F_Count++;
    if((exam_mark == 0) && (assignment_count < 2))
        cout << "Check for cancelled enrollment" << endl;
}
else
if(mark<65)
    D_Count++;
else
...

```

Here, the `{` and `}` brackets around the compound statement make it clear that the check for cancelled enrollments is something that is done only when considering marks less than 50.

The `{ }` brackets in that code fragment were necessary because the `mark<50` condition controlled several statements. Suppose there was no need to keep a count of students who scored less than half marks, the code would then be:

```

if(mark<50) {
    if((exam_mark == 0) && (assignment_count < 2))
        cout << "Check for cancelled enrollment" << endl;
}
else
if(mark<65)
...

```

Some beginners might be tempted to remove the `{ }` brackets from the `mark<50` condition; the argument might be that since there is only one statement "you don't need a compound statement construction". The code would then become:

Buggy code!

```

if(mark<50)
    if((exam_mark == 0) && (assignment_count < 2))
        cout << "Check for cancelled enrollment" << endl;
else
if(mark<65)

```

Now, you have a bug.

An `if` statement can exist on its own, without any `else` clause. So, when a compiler has finished processing an `if` statement, it gets ready to deal with any kind of subsequent statement assignment, function call, loop, switch, If the

compiler encounters the keyword `else` it has to go back through the code that it thought that it had dealt with to see whether the preceding statement was an `if` that it could use to hang this `else` onto. An `else` clause gets attached to an immediately preceding `if` clause. So the compiler's reading of the buggy code is:

```
if(mark<50)
    if((exam_mark == 0) && (assignment_count < 2))
        cout << "Check for cancelled enrollment" << endl;
    else
        if(mark<65)
            D_Count++;
        else
            ...
```

This code doesn't perform the required data processing steps. What happens now is that students whose marks are greater than or equal to 50 don't get processed – the first conditional test eliminates them. A warning message is printed for students who didn't sit the exam and did at most one assignment. The count `D_Count` is incremented for all other students who had scored less than 50, after all they did score less than 65 which is what the next test checks.

It is possible for the editor in a development environment to rearrange the layout of code so that the indentation correctly reflects the logic of any `ifs` and `elses`. Such editors reduce your chance of making the kind of error just illustrated. Most editors don't provide this level of support; it is expected that you have some idea of what you are doing.

Another legal thing that you shouldn't do is use the comma sequencing operator in an `if`:

```
// If book is overdue, disable borrowing and
// increase fines by $3.50
if(Overdue(book_return_date))
    can_borrow = 0, fine += 3.50;
```

Sure this is legal, it just confuses about 75% of those who read it.

Abbreviations again: the conditional expression

You might expect this by now – the C and C++ languages have lots of abbreviated forms that save typing, one of these abbreviated forms exist for the `if` statement.

Often, you want code like:

```
// If Self Employed then deduction is 10% of entry in box 15
// else deduction is $450.
if(Employ_Status == 'S')
    deduction = 0.1*b15;
else
    deduction = 450.0
```

or

```
// If new temperature exceeds maximum
// recorded, update the maximum
if(temperature > maximum)
    maximum = temperature;
```

or

```
// print label "male" or "female" as appropriate
if(gender_tag == 'f')
    cout << "Female : ";
else
    cout << "Male   : ";
```

All can be rewritten using a conditional expression.

This has the form

```
(boolean expression) ? result_if_true : result_if_false
```

The conditional expression has three parts. The first part defines the condition that is to be tested. A question mark separates this from the second part that specifies the result of the whole conditional expression for the case when the condition evaluates to true. A colon separates this from the third part where you get the result for cases where the condition is false.

Using this conditional expression, the code examples shown above may be written as follows:

```
// If Self Employed then deduction is 10% of entry in box 15
// else deduction is $450.
deduction = (Employ_Status == 'S') ?
             0.1*b15      :
             450.0;

maximum = (temperature > maximum) ?
          temperature  :
          maximum;

cout << ((gender_tag == 'f') ? "Female : " : "Male   : ");
```

**Caution – watch out
for typos related to
== operator**

Note: remember to use == as the equality test operator and not the = assignment operator! The following is legal code:

```
cout << ((gender_tag = 'f') ? "Female : " : "Male   : ");
```

but it changes the value of gender_tag, then confirms that the new value is non zero and prints "Female : ".

8.4 TERMINATING A PROGRAM

Sometimes, you just have to stop. The input is junk; your program can do no more. It just has to end.

The easy way out is to use a function defined in `stdlib`. This function, `exit()`, *exit() function* takes an integer argument. When called, the `exit()` function terminates the program. It tries to clean up, if you were using files these should get closed. The integer given to `exit()` gets passed back as the return status from the program (like the 0 or `EXIT_SUCCESS` value returned by the main program). A typical IDE will simply discard this return value. It is used in scripting environments, like Unix, where the return result of a program can be used to determine what other processing steps should follow. The normal error exit value is 1; other values can be used to distinguish among different kinds of error.

Several of the later examples use calls to `exit()` as an easy way to terminate a program if the data given are found to be invalid. For example, the square root program (7.2.2) could check the input data as follows:

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    const double SMALLFRACTION = 1.0E-8;
    double x;
    double r;
    cout << "Enter number : ";
    cin >> x;

    if(x <= 0.0) {
        cout << "We only serve Positive numbers here."
              << endl;
        exit(1);
    }
    ...
}
```

8.5 EXAMPLE PROGRAMS

8.5.1 Calculating some simple statistics

Problem:

You have a collection of data items; each data item consists of a (real) number and a character (m or f). These data represent the heights in centimetres of some school children. Example data:

140.5	f
148	m
137.5	m
133	f

```

129.5      m
156        f
...

```

You have to calculate the average height, and standard deviation in height, for the entire set of children and, also, calculate these statistics separately for the boys and the girls.

Specification:

1. The program is to read height and gender values from `cin`. The data can be assumed to be correct; there is no need to check for erroneous data (such as gender tags other than 'f' or 'm' or a height outside of a 1 to 2 metre range). The data set will include records for at least three boys and at least three girls; averages and standard deviations will be defined (i.e. no need to check for zero counts).
2. The input is to be terminated by a "sentinel" data record with a height 0 and an arbitrary m or f gender value.
3. When the terminating sentinel data record is read, the program should print the following details: total number of children, average height, standard deviation in height, number of girls, average height of girls, standard deviation of height of girls, number of boys, average height of boys, standard deviation of height of boys.

Watch out for the fine print

Note that this specification excludes some potential problems.

Suppose the specification did not mention that there would be a minimum number of boys and girls in the data set?

It would become *your* responsibility to realize that there then was a potential difficulty. If all the children in the sample were boys, you would get to the end of program with:

```

number_girls      0
sum_girls_heights 0.0

```

and would need to work out

```
average_girl_height = sum_girls_heights / number_girls;
```

Execution of this statement when `number_girls` was zero would cause your program to be terminated with arithmetic overflow. You would have to plan for more elaborate processing with `if` statements "guarding" the various output sections so that these were only executed when the data were appropriate.

Remember, program specifications are often drawn up by teams including lawyers. Whenever you are dealing with lawyers, you should watch out for nasty details hidden in the fine print.

"sentinel" data

The specification states that input will be terminated by a "sentinel" data record (dictionary: *sentinel* – 1) soldier etc posted to keep guard, 2) Indian-Ocean crab

with long eye-stalks; this use of sentinel derives from meaning 1). A sentinel data record is one whose value is easily recognized as special, not a normal valid data value; the sentinel record "guards" the end of input stopping you from falling over trying to read data that aren't there. Use of a sentinel data record is one common way of identifying when input is to stop; some alternatives are discussed in Chapter 9 where we use files of data.

Program design

First, it is necessary to check the formulae for calculating averages (means) and standard deviations! *Preliminary design*

Means are easy. You simply add up all the data values as you read them in and then divide by the number of data values. In this case we will get something like:

```
average_height = sum_heights / number_children;
```

The standard deviation is a little trickier. Most people know (or at least once knew but may have long forgotten) the following formula for the standard deviation for a set of values x_i .

$$S = \sqrt{\frac{\sum_{i=1}^N (x_i - \bar{x})^2}{N - 1}}$$

S standard deviation
N number of samples
 x_i sample i
 \bar{x} average value of samples

i.e. to work out the standard deviation you sum the squares of the differences between sample values and the average, divide this sum by N-1 and take the square root.

The trouble with that formula is that you need to know the average. It suggests that you have to work out the value in a two step process. First, you read and store all the data values, accumulating their total as you read them in, and working out their average when all are read. Then in a second pass you use your stored data values to calculate individual differences, summing the squares of these differences etc.

Fortunately, there is an alternative version of the formula for the standard deviation that lets you calculate its value without having to store all the individual data elements.

$$S = \sqrt{\frac{\sum_{i=1}^N x_i^2 - N\bar{x}^2}{N - 1}}$$

This formula requires that as well as accumulating a sum of the data values (for calculating the average), we need to accumulate a sum of the squares of the data values. Once all the data values have been read, we can first work out the average and then use this second formula to get the standard deviation.

**First iteration
through design
process**

If we are to use this second formula, the program structure is roughly:

```
initialize count to zero, sum of values to zero
and sum of squares to zero
read in the first data element (height and gender flag)
while height ≠ 0
    increment count
    add height to sum
    add square of height to sum of squares
    read next data element

calculate average by dividing sum by number
use second formula to calculate standard deviation
```

Of course, we are going to need three counts, three sums, three sums of squares because we need to keep totals and individual gender specific values. We also need to elaborate the code inside the loop so that in addition to updating overall counts we selectively update the individual gender specific counts.

**Second iteration
through design
Data**

We can now begin to identify the variables that we will need:

three integers	children, boys, girls
three doubles	cSum, bSum, gSum
three doubles	cSumSq, bSumSq, gSumSq
double	height
char	gender_tag
double	average
double	standard_dev

The averages and standard deviations can be calculated and immediately printed out; so we can reuse the same variables for each of the required outputs.

Loop code

The code of the loop will include the selective updates:

```
while height ≠ 0
    increment count
    add height to sum
    add square of height to sum of squares
    if(female)
        update girls count, girls sum, girls sumSq
    else
        update boys count, boys sum, boys sumSq
    read next data element
```

and the output section needs elaboration:

Output code

```
calculate overall average by dividing cSum by children
use second formula to calculate standard deviation
using values for total data set
```

Output details for all children

calculate girls' average by dividing gSum by girls
use second formula to calculate standard deviation
using values for girls data set

Output details for girls

... similarly for boys

Some simple data would be needed to test the program. The averages will be easy to check; most spreadsheet packages include "standard deviation" among their standard functions, so use of a spreadsheet would be one way to check the other results if hand calculations proved too tiresome.

Implementation

Given a fairly detailed design, implementation should again be straightforward. The program needs the standard `sqrt()` function so the math library should be included.

```
#include <iostream.h>
#include <math.h>

int main()
{
    int            children, boys, girls;
    double         cSum, bSum, gSum;
    double         cSumSq, bSumSq, gSumSq;

    double         height;
    char           gender_tag;

    children = boys = girls = 0;
    cSum = bSum = gSum = 0.0;
    cSumSq = bSumSq = gSumSq = 0.0;

    cin >> height >> gender_tag;

    while(height != 0.0) {
        children++;
        cSum += height;
        cSumSq += height*height;

        if(gender_tag == 'f') {
            girls++;
            gSum += height;
            gSumSq += height*height;
        }
        else {
            boys++;
            bSum += height;
        }
    }
}
```

*Conditional updates
of separate counters*

Variables declared in the body of the block

```

        bSumSq += height*height;
    }

    cin >> height >> gender_tag;
}

double        average;
double        standard_dev;

cout << "The sample included " << children
    << " children" << endl;
average = cSum / children;
cout << "The average height of the children is "
    << average;

standard_dev = sqrt(
    (cSumSq - children*average*average) /
    (children-1));

cout << "cm, with a standard deviation of " <<
    standard_dev << endl;

cout << "The sample included " << girls
    << " girls" << endl;

average = gSum / girls;
cout << "The average height of the girls is "
    << average;

standard_dev = sqrt(
    (gSumSq - girls*average*average) /
    (girls-1));

...
...

return 0;
}

```

Note, in this case the variables `average` and `standard_dev` were not defined at the start of `main()`'s block; instead their definitions come after the loop at the point where these variables are first used. This is typical of most C++ programs. However, you should note that some people feel quite strongly that all variable declarations should occur at the beginning of a block. You will need to adjust your coding style to meet the circumstances.

Expression passed as argument for function call

Function `sqrt()` has to be given a double value for the number whose root is needed. The call to `sqrt()` can involve any expression that yields a double as a result. The calls in the example use the expression corresponding to the formula given above to define the standard deviation.

Test run

When run on the following input data:

```

135      f
140      m
139      f

```



```

151.5    f
133      m
148      m
144      f
146      f
142      m
144      m
0        m

```

The program produced the following output (verified using a common spreadsheet program):

```

The sample included 10 children
The average height of the children is 142.25cm, with a
standard deviation of 5.702095
The sample included 5 girls
The average height of the girls is 143.1cm, with a standard
deviation of 6.367888
The sample included 5 boys
The average height of the boys is 141.4cm, with a standard
deviation of 5.549775

```

The standard deviations are printed with just a few too many digits. Heights measured to the nearest half centimetre, standard deviations quoted down to about Angstrom units! The number of digits used was determined simply by the default settings defined in the `iostream` library. Since we know that these digits are spurious, we really should suppress them.

Dubious trailing digits!

There are a variety of ways of controlling the layout, or "format", of output if the defaults are inappropriate. They work, but they aren't wildly convenient. Most modern programs display their results using graphics displays; their output operations usually involve first generating a sequence of characters (held in some temporary storage space in memory) then displaying them starting at some fixed point on the screen. Formatting controls for sending nice tabular output to line printers etc are just a bit *passe*.

Formatting

Here we want to fix the precision used to print the numbers. The easiest way is to make use of an extension to the standard `iostream` library. The library file `iomanip` contains a number of extensions to standard `iostream`. Here, we can use `setprecision()`. If you want to specify the number of digits after the decimal point you can include `setprecision()` in your output statements:

iomanip

```
cout << setprecision(2) << value;
```

This would limit the output to two fraction digits so you would get numbers like 12.25 rather than 12.249852. If your number was too large to fit, e.g. 35214.27, it will get printed in "scientific" format i.e. as 3.52e4.

Once you've told `cout` what precision to use, that is what it will use from that point on. This can be inconvenient. While we might want only one fraction digit for the standard deviations, we need more for the averages (if we set `setprecision(1)`, the an average like 152.643 may get printed as 1.5e2). Once you've started specifying precisions, you are committed. You are going to have to do it everywhere. You will need code like:

precision is a "sticky" format setting

```

cout << "The average height of the children is "
      << setprecision(3) << average;

standard_dev = sqrt(
    (cSumSq - children*average*average) /
    (children-1));

cout << "cm, with a standard deviation of " <<
      setprecision(1) << standard_dev << endl;

cout << "The sample included " << girls <<
      " girls" << endl; // integer, no precision

```

(You must `#include <iomanip.h>` if you want to use `setprecision()`.) On the whole, it is best not to bother about these fiddly formats unless you really must.

8.5.2 Newton's method for roots of polynomials

Problem:

You have to find the root of a polynomial; that is given some function of x , like $13.5x^4 - 59x^3 - 28x^2 + 16.5x + 30$, you must find the values of x for which this is 0.

The method is similar to that used earlier to find square roots. You keep guessing, in a systematic controlled way, until you are happy that you are close enough to a root. Figure 8.1 illustrates the basis for a method of guessing systematically. You have to be given two values of x that bracket (lie on either side of) a root; for one ("posX") the polynomial has positive value, the value is negative for the other ("negX"). Your next guess for x should be mid-way between the two existing guesses.

If the value of the polynomial is positive for the new x value, you use this x value to replace the previous posX, otherwise it replaces the previous negX. You then repeat the process, again guessing halfway between the updated posX, negX pair. If the given starting values did bracket a single root, then each successive guess should bring you a little closer.

Specification:

1. The program is to find a root of the polynomial $13.5x^4 - 59x^3 - 28x^2 + 16.5x + 30$.
2. The program is to take as input guesses for two values of x that supposedly bracket a root.

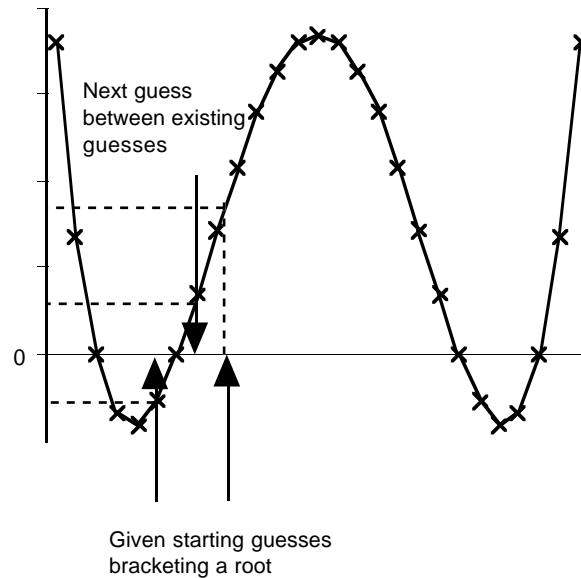


Figure 8.1 Principles of root finding!

3. The program is to verify that the polynomial does have values of opposite sign for the given x values. If the polynomial has the same sign at the two given values (no root, or a pair of roots, existing between these values) the program is to print a warning message and then stop.
4. The program is to use the approach of guessing a new value mid way between the two bracketing values, using the guessed value to replace one of the previous values so that it always has a pair of guesses on either side of the root.
5. The program is to print the current estimate for the root at each cycle of the iterative process.
6. The loop is to terminate when the guess is sufficiently close to a root; this should be taken as meaning the value for the polynomial at the guessed x is less than 0.00001.

Program design

A first iteration through the design process gives a structure something like the *Preliminary design* following:

```

prompt for and read in guesses for the two bracketing x
values
calculate the values of the function at these two xs
if signs of function are same
    print warning message
    exit
initialize negX with x value for which polynomial was -ve

```

```

initialize posX with x value for which polynomial was +ve

pick an x midway between posX and negX
evaluate polynomial at x

while value of polynomial exceeds limit
    print current guess and polynomial value
    if value of polynomial is negative
        replace negX by latest guess
    else replace posX by latest guess
    set x midway between updated posX, negX pair
    evaluate polynomial at x

print final guess

```

Detailed design The data? Rather a large number of simple variables this time. From the preliminary design outline, we can identify the following:

posX, negX	the bracketing values
g1, g2	the initial guesses for bracketing values
x	the current guess for the root
fun1, fun2	the values of the polynomial at g1 and g2
fun3	the value of the polynomial at x

all of these would be doubles and belong to the main routine.

How to evaluate the polynomial?

Really, we want a function to do this! After all, the polynomial is "a function of x". But since we can't yet define our own functions we will have to have the code written out at each point that we need it. Exercise 2 in Chapter 7 presented some alternative ways of evaluating polynomial functions. Here, we will use the crudest method! The value of the polynomial for a specific value of x is given by the expression:

$$13.5 * \text{pow}(x, 4) - 59 * \text{pow}(x, 3) - 28 * \text{pow}(x, 2) + 16.5 * x + 30$$

What is the simplest way of checking whether fun1 and fun2 have the same sign?

You don't need a complex boolean expression like:

```

(((fun1 < 0.0) && (fun2 < 0.0)) ||
 ((fun1 > 0.0) && (fun2 > 0.)))

```

All you have to do is multiply the two values; their product will be positive if both were positive or both were negative.

Implementation

The implementation is again straightforward. The code would be:

```

#include <stdlib.h>
#include <iostream.h>
#include <math.h>

int main()
{
    double posX, negX;

    double fun1, fun2, fun3;

    cout << "Enter guesses that bracket a root" << endl;
    double g1, g2;
    cin >> g1;
    cin >> g2;
    double x = g1;
    fun1 = 13.5*pow(x,4) - 59*pow(x,3)- 28*pow(x,2)
          + 16.5*x + 30;
    x = g2;
    fun2 = 13.5*pow(x,4) - 59*pow(x,3)- 28*pow(x,2)
          + 16.5*x + 30;

    if((fun1*fun2) > 0.0) {
        cout << "Those values don't appear to bracket a"
              " root" << endl;
        exit(1);
    }
    negX = (fun1 < 0.0) ? g1 : g2;
    posX = (fun1 > 0.0) ? g1 : g2;

    x = 0.5*(negX + posX);
    fun3 = 13.5*pow(x,4) - 59*pow(x,3)- 28*pow(x,2)
          + 16.5*x + 30;
    while(fabs(fun3) > 0.00001) {
        cout << x << ", " << fun3 << endl;
        if(fun3 < 0.0) negX = x;
        else posX = x;
        x = 0.5*(negX + posX);
        fun3 = 13.5*pow(x,4) - 59*pow(x,3)- 28*pow(x,2)
              + 16.5*x + 30;
    }

    cout << "My final guess for root : " << endl;
    cout << x << endl;

    return EXIT_SUCCESS;
}

```

Convince yourself that the code does initialize `posX` and `negX` correctly from the guesses `g1` and `g2` with its statements:

```

negX = (fun1 < 0.0) ? g1 : g2;
posX = (fun1 > 0.0) ? g1 : g2;

```

8.6 WHAT IS THE LARGEST? WHAT IS THE SMALLEST?

In example 8.5.1, the statistics that had to be calculated were just mean values and standard deviations. Most such problems also require identification of minimum and maximum values.

Suppose we needed to find the heights of the smallest and tallest of the children. The program doesn't require much extension to provide these statistics. We have to define a couple more variables, and add some code in the loop to update their values:

```
int main()
{
    int          children, boys, girls;
    double       cSum, bSum, gSum;
    double       cSumSq, bSumSq, gSumSq;

    double       height;
    char         gender_tag;

    double       smallest, tallest;

    children = boys = girls = 0;
    ...

    cin >> height >> gender_tag;

    while(height != 0.0) {
        children++;
        cSum += height;
        cSumSq += height*height;

        smallest = (height < smallest) ?
                    height :
                    smallest;

        tallest = (height > tallest) ?
                  height :
                  tallest;
    }
```

Of course, we need `smallest` and `tallest` to have some initial values. The initial value of `tallest` would have to be less than (or equal) to the (unknown) height of the tallest child, that way it will get changed to the correct value when the data record with the maximum height is read. Similarly, `smallest` should initially be greater than or equal to the height of the smallest child.

How should you chose the initial values?

Initialize minima and maxima with actual data

Generally when searching for minimum and maximum values in some set of data, the best approach is to use the first data value to initialize both the minimum and maximum values:

```
cin >> height >> gender_tag;
```

```

smallest = tallest = height;

while(height != 0.0) {
    children++;
    ...

```

This initialization guarantees that initial values are in appropriate ranges.

It would have been a little harder if we had needed the heights of the smallest boy and of the smallest girl. These can't both be initialized from the first data value. The first value might represent a small boy, smaller than any of the girls but not the smallest of the boys; if his height was used to initialize the minimum height for girls then this value would never be corrected.

If you can't just use the first data value, then use a constant that you "know" to be appropriate. The problem specification indicated that the children were in the height range from 1 metre to 2 metres; so we have suitable constants:

*Initialize from
constants
representing extreme
values*

```

// Children should be taller than 100 cm
// and less than 200 cm
const double LowLimit = 100;
const double HiLimit = 200;
...
...
smallest = HiLimit;
tallest = LowLimit;

cin >> height >> gender_tag;
while(height != 0.0) {
    ...

```

Note: `smallest` gets initialized with `HiLimit` (it is to start *greater* than or equal to smallest value and `HiLimit` should be safely greater than the minimum); similarly `tallest` gets initialized with `LowLimit`.

You don't have to define the named constants; you could simply have:

```

smallest = 100;
tallest = 200;

```

but, as noted previously, such "magic numbers" in the code are confusing and often cause problems when you need to change their values. (You might discover that some of the children were aspiring basket ball players; your 200 limit might then be insufficient).

In this example, the specification for the problem provided appropriate values that could be used to initialize the minimum and maximum variables. Usually, this is not the case. The specification will simply say something like "read in the integer data values and find the maximum from among those that ...". What data values? What range? What is a plausible maximum?

*Don't go guessing
limits*

Very often beginners guess.

They guess badly. One keeps seeing code like the following:

```

min_val = 10000;
max_val = 0;

```

```

cin >> val >> code_char;
while(code_char != 'q') {
    min_val = (val < min_val) ? val : min_val;
    ...
}

```

Who said that 10000 was large? Who said that the numbers were positive? Don't guess. If you don't know the range of values in your data, initialize the "maximum" with the least (most negative) value that your computer can use; similarly, initialize the "minimum" with the greatest (largest positive) value that your computer can use.

Limits.h The system's values for the maximum and minimum numbers are defined in the header file limits.h. This can be #included at the start of your program. It will contain definitions (as #defines or const data definitions) for a number of limit values. The following table is part of the contents of a limits.h file. SHRT_MAX is the largest "short integer" value, LONG_MAX is the largest (long) integer and LONG_MIN (despite appearances) is the most negative possible number (-2147483648). If you don't have any better limit values, use these for initialization.

```

#define SHRT_MIN      (~32767)
#define SHRT_MAX      32767
#define USHRT_MAX     0xFFFF

#define LONG_MIN      (~2147483647L)
#define LONG_MAX      2147483647L
#define ULONG_MAX     0xFFFFFFFFL

```

Once again, things aren't quite standardized. On many systems but not all, the limits header file also defines limiting values for doubles (the largest, the smallest non-zero value etc).

Unsigned values The list given above included limit values for "unsigned shorts" and "unsigned longs". Normally, one-bit of the data field used by an integer is used in effect for the \pm sign; so, 16-bits can hold at most a 15-bit value (numbers in range -32768 to +32767). If you know that you only need positive integers, then you can reclaim the "sign bit" and use all 16 bits for the value (giving you a range 0...65535).

Unsigned integers aren't often used for arithmetic. They are more often used when the bit pattern isn't representing a number – it is a bit pattern where specific bits are used to encode the true/false state of separate data values. Use of bit patterns in this way economises on storage (you can fit the values of 16 different boolean variables in a single unsigned short integer). Operations on such bit patterns are illustrated in Chapter X.

The definitions for the "MIN" values are actually using bit operations. That character in front of the digits isn't a minus sign; it is ~ ("tilde") – the bitwise not operator. Why the odd bit operations? All will be explained in due time!

EXERCISES

Loops and selection statements provide considerable computational power. However, the range of programs that we can write is still limited. The limitation is now due mainly to a

lack of adequate data structures. We have no mechanism yet for storing data. All we can do is have a loop that reads in successive simple data values, like the children's heights, and processes each data element individually combining values to produce statistics etc. Such programs tend to be most useful when you have large numbers of simple data values from which you want statistics (e.g. heights for all year 7 children in your state); but large amounts of data require file input. There are just a few exercises here; one or two at the end of Chapter 9 are rather similar but they use file input.

1. Implement a working version of the Calculator program and test its operation.
2. Extend the calculator to include a separate memory. Command 'm' copies the contents of the display into the memory, 'M' adds the contents of the display to memory; similar command 'r' and 'R' recall the memory value into the display (or add it to the display).
3. Write a program that "balances transactions on a bank account" and produces a financial summary.

The program is to prompt for and read in the initial value for the funds in the account. It is then to loop prompting for and reading transactions.

Transactions are entered as a number and a character, e.g.

```
128.35    u
 79.50    f
 10.60    t
 66.67    r
  9.80    e
213.50    j
 84.30    f
 66.67    r
...
```

The character codes are:

debits (expenditures):

u	utilities (electricity, gas, phone)
f	food
r	rent
c	clothing
e	entertainment
t	transport
w	work related
m	miscellaneous

credits (income)

j	job
p	parents
l	loan

Credit entries should increase available funds, debit entries decrease the funds.

Entry of the value 0 (zero) is to terminate the loop (the transaction code with a 0 entry will be arbitrary).

When the loop has finished, the program is to print the final value for the funds in the account. It is also to print details giving the total amounts spent in each of the possible expenditure categories and also giving these values percentages of the total spending. The program should check that the transaction amounts are all greater than zero and that the transaction codes are from the set given. Data that don't comply with these requirements are to be discarded after a warning message has been printed; the program is to continue prompting for and processing subsequent data entries.

4. Extend the childrens' heights program to find the heights of the smallest boy and the smallest girl in the data set.
5. Modify the heights program so that it can deal correctly with data sets that are empty (no children) or have children who all are of the same gender. The modified program should print appropriate outputs in all cases (e.g. "There were no girls in the data given").
6. Write a program to find a root(for polynomial of a given maximum degree 4, i.e. a function of the form $c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0$ for arbitrary values of the coefficients c_4 , c_3 , c_2 , c_1 , and c_0 .

The formula for evaluating the polynomial at a given value of x is

$$\text{val} = (((c_4 * x + c_3) * x + c_2) * x + c_1) * x + c_0$$

The program is to prompt for and read the values of the coefficients.

The program should then prompt for a starting x value, an increment, and a final x value and should tabulate the value the polynomial at each successive x value in the range. This should give the user an idea of where the roots might be located.

Finally, the program should prompt for two x values that will bracket a root and is to find the root (if any) between these given values.

9 Simple use of files

9.1 DEALING WITH MORE DATA

Realistic examples of programs with loops and selection generally have to work with largish amounts of data. Programs need different data inputs to test the code for all their different selections. It is tiresome to have to type in large amounts of data every time a program is tested. Also, if data always have to be keyed in, input errors become more likely. Rather than have the majority of the data entered each time the program is run, the input data can be keyed into a text file once. Then, each time the program is subsequently run on the same data, the input can be taken from this file.

While some aspects of file use have to be left to later, it is worth introducing simple ways of using files. Many of the later example programs will be organized as shown in Figure 9.1.

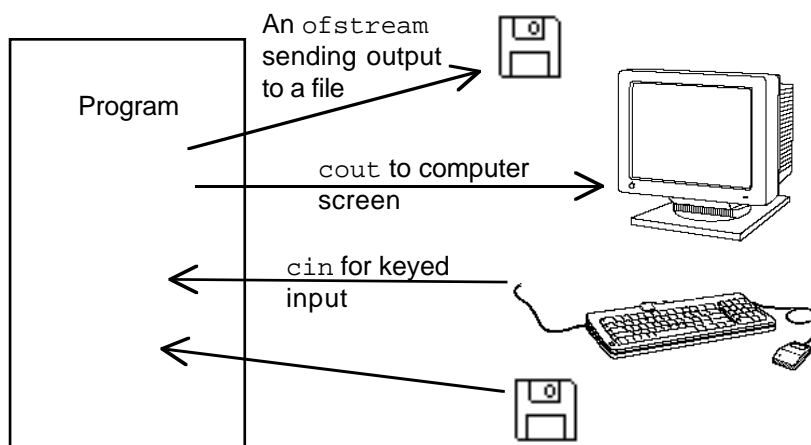


Figure 9.1 Programs using file i/o to supplement standard input and output.

These programs will typically send most of their output to the screen, though some may also send results to output files on disks. Inputs will be taken from file

or, in some cases, from both file and keyboard. (Input files can be created with the editor part of the integrated development environment used for programming, or can be created using any word processor that permits files to be saved as "text only".)

Text files

Input files (and any output files) will be simply text files. Their names should be chosen so that it is obvious that they contain data and are not "header" (".h") or "code" (".cp") files. Some environments may specify naming conventions for such data files. If there are no prescribed naming schemes, then adopt a scheme where data files have names that end with ".dat" or ".txt".

Binary files

Data files based on text are easy to work with; you can always read them and change them with word processors etc. Much later, you will work with files that hold data in the internal binary form used in programs. Such files are preferred in advanced applications because, obviously, there is no translation work (conversion from internal to text form) required in their use. But such files can not be read by humans.

"Redirection of input and output".

Some programming environments, e.g. the Unix environment, permit inputs and outputs to be "redirected" to files. This means that:

- 1) you can write a program that uses `cin` for input, and `cout` for output, and test run it normally with the input taken from keyboard and have results sent to your screen,

then

- 2) you can subsequently tell the operating system to reorganize things so that when the program needs input it will read data from a file instead of trying to read from the keyboard (and, similarly, output can be routed to a file instead of being sent to the screen).

Such "redirection" is possible (though a little inconvenient) with both the Borland Symantec systems. But, instead of using `cin` and `cout` redirected to files, all the examples will use explicitly declared "filestream" objects defined in the program.

The program will attach these filestream objects to named files and then use them for input and output operations.

9.2 DEFINING FILESTREAM OBJECTS

If a program needs to make explicit connections to files then it must declare some "filestream" objects.

fstream.h header file

The header file `fstream.h` contains the definitions of these objects. There are basically three kinds:

- `ifstream` objects --- these can be used to read data from a named file;
- `ofstream` objects --- these can be used to write data to a named file;

and

- `fstream` objects --- these are used with files that get written and read (at different times) by a program.

The examples will mainly use `ifstream` objects for input. Some examples may have `ofstream` objects.

The `fstream` objects, used when files are both written and read, will not be encountered until much later examples (those dealing with files of record structures).

A program that needs to use filestreams must include both the `iostream.h` and `fstream.h` header files:

```
#include <iostream.h>
#include <fstream.h>

int main()
{
    ...;
}
```

`ifstream` objects are simply specialized kinds of input stream objects. They can perform all the same kinds of operations as done by the special `cin` input stream, i.e. they can "give" values to integers, characters, doubles, etc. But, in addition, they have extra capabilities like being able to "open" and "close" files. *ifstream objects*

Similarly, `ofstream` objects can be asked to do all the same things as `cout` – print the values integers, doubles etc – but again they can also open and close output files. *ofstream objects*

The declarations in the file `fstream.h` make `ifstream`, `ofstream`, and `fstream` "types". Once the `fstream.h` header file has been included, the program can contain definitions of variables of these types.

Often, filestream variables will be globals because they will be shared by many different routines in a program; but, at least in the first few examples, they will be defined as local variables of the main program.

There are two ways that such variables can be defined.

```
#include <iostream.h>
#include <fstream.h>

int main()
{
    ifstream    input("theinput.txt", ios::in);
    ...
}
```

*Defining a filestream
attached to a file with
a known fixed name*

Here variable `input` is defined as an input filestream attached to the file called `theinput.txt` (the token `ios::in` specifies how the file will be used, there will be more examples of different tokens later). This style is appropriate when the name of the input file is fixed.

The alternative form of definition is:

*Defining a filestream
that will subsequently
be attached to a
selected file*

```
#include <iostream.h>
#include <fstream.h>

int main()
{
    ifstream    in1;
    ...
}
```

This simply names `in1` as something that will be an input filestream; `in1` is not attached to an open file and can not be used for reading data until an "open" operation is performed naming the file. This style is appropriate when the name of the input file will vary on different runs of the program and the actual file name to be used is to be read from the keyboard.

The open request uses a filename (and the `ios::in` token). The filename will be a character string; usually a variable but it can be a constant (though then one might as well have used the first form of definition). With a constant filename, the code is something like:

Call to open()

```
#include <iostream.h>
#include <fstream.h>

void main()
{
    ifstream    in1;
    ...
    in1.open("file1.txt", ios::in);
    // can now use in1 for input ...
    ...
}
```

(Realistic use of `open()` with a variable character string has to be left until arrays and strings have been covered.)

9.3 USING INPUT AND OUTPUT FILESTREAMS

Once an input filestream variable has been defined (and its associated file opened either implicitly or explicitly), it can be used for input. Its use is just like `cin`:

```
#include <iostream.h>
#include <fstream.h>

void main()
{
    ifstream    input("theinput.txt", ios::in);
    long l1, l2; double d3; char ch;
    ...
    input >> d3; // read a double from file
    ...
    input >> ch; // read a character
    ...
    input >> l1 >> l2; // read two long integer
}
```

Similarly, `ofstream` output files can be used like `cout`:

```
#include <iostream.h>
#include <fstream.h>

void main()
{
    // create an output file
    ofstream out1("results.txt", ios::out);
    int i; double d;
    ...
    // send header to file
    out1 << "The results are :" << endl;      ...
    for(i=0; i<100; i++) {
        ...
        // send data values to file
        out1 << "i : " << i << ",    di " << d << endl;
        ...
    }
    out1.close(); //finished, close the file.
```

This code illustrates "close", another of the extra things that a filestream can do that a simple stream cannot:

```
out1.close();
```

Programs should arrange to "close" any output files before finishing, though the operating system will usually close any input files or output files that are still open when a program finishes.

9.4 STREAM' STATES

All stream objects can be asked about their state: "Was the last operation successful?", "Are there any more input data available?", etc. Checks on the states of streams are more important with the filestreams than with the simple `cin` and `cout` streams (where you can usually see if anything is wrong).

It is easy to get a program to go completely wrong if it tries to take more input data from a filestream after some transfer operations have already failed. The program may "crash", or may get stuck forever in a loop waiting for data that can't arrive, or may continue but generate incorrect results.

Operations on filestreams should be checked by using the normal stream functions `good()`, `bad()`, `fail()`, and `eof()`. These stream functions can be applied to any stream, but just consider the example of an `ifstream`:

```
ifstream in1("indata.txt", ios::in);
```

The state of stream `in1` can be checked:

```
in1.good()      returns "True" (1) if in1 is OK to use
in1.bad()       returns "True" (1) if last operation on
```

	<i>in1 failed and there is no way to recover from failure</i>
<code>in1.fail()</code>	<i>returns "True" (1) if last operation on in1 failed but recovery may be possible</i>
<code>in1.eof()</code>	<i>returns "True" (1) if there are no more data to be read.</i>

If you wanted to give up if the next data elements in the file aren't two integers, you could use code like the following:

```
ifstream in1("indata.txt", ios::in);
long      val1, val2;
in1 >> val1 >> val2;

if(in1.fail()) {
    cout << "Sorry, can't read that file" << endl;
    exit(1);
}
```

(There are actually two status bits associated with the stream – the badbit and the failbit. Function `bad()` returns true if the badbit is set, `fail()` returns true if either is set.)

Naturally, this being C++ there are abbreviations. Instead of phrasing a test like:

```
if(in1.good())
    ...
```

you can write:

```
if(in1)
    ...
```

and similarly you may substitute

```
if(!in1)
```

for

```
if(in1.bad())
```

Most people find these particular abbreviated forms to be somewhat confusing so, even though they are legal, it seems best not to use them. Further, although these behaviours are specified in the `iostream` header, not all implementations comply!

One check should always be made when using `ifstream` files for input. *Was the file opened?*

There isn't much point continuing with the program if the specified data file isn't there.

The state of an input file should be checked immediately after it is opened (either implicitly or explicitly). If the file is not "good" then there is no point in continuing.

A possible way of coding the check for an OK input file should be as follows:

```
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>

int main()
{
    ifstream in1("mydata.txt", ios::in);
    switch(in1.good()) {
case 1:
    cout << "The file is OK, the program continues" << endl;
    break;
case 0:
    cout << "The file mydata.txt is missing, program"
         << " gives up" << endl;
    exit(1);
    }
    ...
}
```

It is a pity that this doesn't work in all environments.

Implementations of C++ aren't totally standardized. Most implementations interpret any attempt to open an input file that isn't there as being an error. In these implementations the test `in1.good()` will fail if the file is non-existent and the program stops.

Unfortunately, some implementations interpret the situation slightly differently. If you try to open an input file that isn't there, an empty file is created. If you check the file status, you are told it's OK. As soon as you attempt to read data, the operation will fail.

The following style should however work in all environments:

```
int main()
{
    ifstream in1("mydata.txt", ios::in | ios::nocreate);
    switch(in1.good()) {
    ...
    }
```

The token combination `ios::in | ios::nocreate` specifies that an input file is required and it is *not* to be created if it doesn't already exist.

9.5 OPTIONS WHEN OPENING FILESTREAMS

You specify the options that you want for a file by using the following tokens either individually or combination:

<code>ios::in</code>	<i>Open for reading.</i>
<code>ios::out</code>	<i>Open for writing.</i>
<code>ios::ate</code>	<i>Position to the end-of-file.</i>
<code>ios::app</code>	<i>Open the file in append mode.</i>
<code>ios::trunc</code>	<i>Truncate the file on open.</i>
<code>ios::nocreate</code>	<i>Do not attempt to create the file if it</i>

```

                                does not exist.
ios::noreplace    Cause the open to fail if the file exists.
ios::translate    Convert CR/LF to newline on input and
                   vice versa on output.

```

(The translate option may not be in your selection; you may have extras, e.g. `ios::binary`.) Obviously, these have to make sense, there is not much point trying to open an ifstream while specifying `ios::out`!

Typical combinations are:

```

ios::in | ios::nocreate    open if file exists,
                           fail otherwise
ios::out | ios::noreplace  open new file for output,
                           fail if filename
                           already exists
ios::out | ios::ate        (re)open an existing output
                           file, arranged so
                           that new data added
                           at the end after
                           existing data
ios::out | ios::noreplace | ios::translate
                           open new file, fail if name
                           exists, do newline
                           translations
ios::out | ios::nocreate | ios::trunc
                           open an existing file for
                           output, fail if file
                           doesn't exist, throw
                           away existing
                           contents and start
                           anew

```

"Appending" data to an output file, `ios::app`, might seem to mean the same as adding data at the end of the file, `ios::ate`. Actually, `ios::app` has a specialized meaning – writes always occur at the end of the file irrespective of any subsequent positioning commands that might say "write here rather than the end". The `ios::app` mode is really intended for special circumstances on Unix systems etc where several programs might be trying to add data to the same file. If you simply want to write some extra data at the end of a file use `ios::ate`.

"Bitmasks"

The tokens `ios::in` etc are actually constants that have a single bit set in bit map. The groupings like `ios::open | ios::ate` build up a "bitmask" by bit-oring together the separate bit patterns. The code of `fstream`'s `open()` routines checks the individual bit settings in the resulting bit mask, using these to select processing options. If you want to combine several bit patterns to get a result, you use an bit-or operation, operator `|`.

Don't go making the following mistakes!

```

ofstream    out1("results.dat", ios::out || ios::noreplace);
ifstream    in1("mydata.dat", ios::in & ios::nocreate);

```

The first example is wrong because the boolean or operator, `||`, has been used instead of the required bit-or operator `|`. What the code says is "If either the

constant `ios::out` or `ios::noreplace` is non zero, encode a one bit here". Now both these constants are non zero, so the first statement really says `outl("results.dat", 1)`. It may be legal C++, but it sure confuses the run-time system. Fortuitously, code 1 means the same as `ios::in`. So, at run-time, the system discovers that you are opening an output file for reading. This will probably result in your program being terminated.

The second error is a conceptual one. The programmer was thinking "I want to specify that it is an input file and it is not to be created". This lead to the code `ios::in & ios::nocreate`. But as explained above, the token combinations are being used to build up a bitmask that will be checked by the `open()` function. The bit-and operator does the wrong combination. It is going to leave bits set in the bitmask that were present in both inputs. Since `ios::in` and `ios::nocreate` each have only one bit, a different bit, set the result of the `&` operation is 0. The code is actually saying `inl("mydata.dat", 0)`. Now option 0 is undefined for `open()` so this isn't going to be too useful.

9.6 WHEN TO STOP READING DATA?

Programs typically have loops that read data. How should such loops terminate?

The sentinel data approach was described in section 8.5.1. A particular data value (something that cannot occur in a valid data element) is identified (e.g. a 0 height for a child). The input loop stops when this sentinel value is read. This approach is probably the best for most simple programs. However, there can be problems when there are no distinguished data values that can't be legal input (the input requirement might be simply "give me a number, any number").

Sentinel data

The next alternative is to have, as the first data value, a count specifying how many data elements are to be processed. Input is then handled using a `for` loop as follows:

Count

```
int      num;
ifstream inl("indata.txt", ios::in);
...
// read number of entries to process
inl >> num;
for(int i = 0; i < num; i++) {
    // read height and gender of child
    char gender_tag;
    double height;
    cin >> height >> gender_tag;
    ...
}
```

The main disadvantage of this approach is that it means that someone has to count the number of data elements!

The third method uses the `eof()` function for the stream to check for "end of file". This is a sort of semi-hardware version of a sentinel data value. There is conceptually a mark at the end of a file saying "this is the end". Rather than check for a particular data value, your code checks for this end of file mark.

eof()

This mechanism works well for "record structured files", see Figure 9.2A. Such files are explained more completely in section 17.3. The basic idea is obvious. You have a file of records, e.g. "Customer records"; each record has some number of characters allocated to hold a name, a double for the amount of money owed, and related information. These various data elements are grouped together in a fixed size structure, which would typically be a few hundred bytes in size (make it 400 for this example). These blocks of bytes would be read and written in single transfers. A record would consist of several successive records.

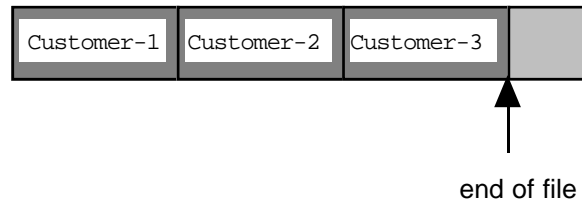
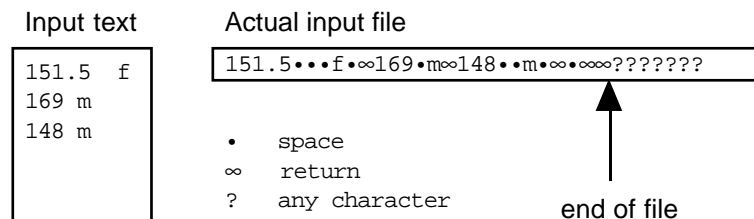
A**"Record structured" file:****B****Text file:**

Figure 9.2 Ends of files.

Three "customer records" would take up 1200 bytes; bytes 0...399 for Customer-1, 400...799 for Customer-2 and 800...1199 for Customer-3. Now, as explained in Chapter 1, file space is allocated in blocks. If the blocksize was 512 bytes, the file would be 1536 bytes long. Directory information on the file would specify that the "end of file mark" was at byte 1200.

You could write code like the following to deal with all the customer records in the file:

```
ifstream sales("Customers.dat", ios::in | ios::nocreate);
if(!sales.good()) {
    cout << "File not there!" << endl;
    exit(1);
}
while(! sales.eof()) {
    read and process next record
}
```

As each record is read, a "current position pointer" gets advanced through the file. When the last record has been read, the position pointer is at the end of the file. The test `sales.eof()` would return true the next time it is checked. So the while loop can be terminated correctly.

The `eof()` scheme is a pain with text files. The problem is that text files will contain trailing whitespace characters, like "returns" on the end of lines or tab characters or spaces; see Figure 9.2B. There may be no more data in the file, but because there are trailing whitespace characters the program will not have reached the end of file.

If you tried code like the following:

```
ifstream kids("theHeights.dat", ios::in | ios::nocreate);
if(!kids.good()) {
    cout << "File not there!" << endl;
    exit(1);
}
while(! kids.eof()) {
    double height;
    char gender_tag;
    kids >> height >> gender_tag;
    ...
}
```

You would run into problems. When you had read the last data entry (148 m) the input position pointer would be just after the final 'm'. There remain other characters still to be read – a space, a return, another space, and two more returns. The position marker is not "at the end of the file". So the test `kids.eof()` will return false and an attempt will be made to execute the loop one more time.

But, the input statement `kids >> height >> gender_tag;` will now fail – there are no more data in the file. (The read attempt will have consumed all remaining characters so when the failure is reported, the "end of file" condition will have been set.)

You can hack around such problems, doing things like reading ahead to remove "white space characters" or "breaking" out of a loop if an input operation fails and sets the end of file condition.

But, really, there is no point fighting things like this. Use the `eof()` check on record files, use counts or sentinel data with text files.

9.7 MORE FORMATTING OPTIONS

The example in section 8.5.1 introduced the `setprecision()` format manipulator from the `iomanip` library. This library contains some additional "manipulators" for changing output formats. There are also some functions in the standard `iostream` library than can be used to change the ways output and input are handled.

You won't make much use of these facilities, but there are odd situations where you will need them.

The following `iomanip` manipulators turn up occasionally:

setw(int)	sets a width for the next output
setfill(int)	changes the fill character for next output

along with some manipulators defined in the standard iostream library such as

hex	output number in hexadecimal format
dec	output number in decimal format

The manipulators setprecision(), hex, and dec are "sticky". Once you set them they remain in force until they are reset by another call. The width manipulators only affect the next operation. Fill affects subsequent outputs where fill characters are needed (the default is to print things using the minimum number of characters so that the filler character is normally not used).

The following example uses several of these manipulators:

```
int main()
{
    int            number = 901;

    cout << setw(10) << setfill('#') << number << endl;
    cout << setw(6) << number << endl;

    cout << dec << number << endl;
    cout << hex << number << endl;
    cout << setw(12) << number << endl;
    cout << setw(16) << setfill('@') << number << endl;

    cout << "Text" << endl;
    cout << 123 << endl;

    cout << setw(8) << setfill('*') << "Text" << endl;

    double d1 = 3.141592;
    double d2 = 45.9876;
    double d3 = 123.9577;

    cout << "d1 is " << d1 << endl;

    cout << "setting precision 3 " << setprecision(3)
        << d1 << endl;
    cout << d2 << endl;
    cout << d3 << endl;
    cout << 4214.8968 << endl;

    return EXIT_SUCCESS;
}
```

The output produced is:

#####901	<i>901, 10 spaces, # as fill</i>
###901	<i>901, 6 spaces, continue # fill</i>
901	<i>just print decimal 901</i>
385	<i>print it as a hexadecimal number</i>
#####385	<i>hex, 12 spaces, # still is filler</i>

```

@@@@@@@@@@@@385  change filler
Text              print in minimum width, no filler
7b               still hex output!
****Text         print text, with width and fill
d1 is 3.141592    fiddle with precision on doubles
setting precision 3 3.142
45.988
123.958
4.215e+03

```

There are alternative mechanisms that can be used to set some of the format options shown above. Thus, you may use *Alternative form of specification*

```

cout.width(8)      instead of   cout << setw(8)
cout.precision(4) ..          cout << setprecision(4)
cout.fill('$')    ..          cout << setfill('$')

```

There are a number of other output options that can be selected. These are defined by more of those tokens (again, these are actually defined bit patterns); they include: *Other strange format options*

```

ios::showpoint      (printing of decimal point and trailing 0s)
ios::showpos        (require + sign with positive numbers)
ios::uppercase      (make hex print ABCDE instead of abcde)

```

These options are selected by telling the output stream object to "set flags" (using its `setf()` function) and are deselected using `unsetf()`. The following code fragment illustrates their use:

```

int main()
{
    long    number = 8713901;

    cout.setf(ios::showpos);
    cout << number << endl;
    cout.unsetf(ios::showpos);
    cout << number << endl;
    cout << hex << number << endl;
    cout.setf(ios::uppercase);
    cout << number << endl;
    return EXIT_SUCCESS;
}

```

The code should produce the output:

```

+8713901  positive sign forced to be present
8713901   normally it isn't shown
84f6ad    usual hex format
84F6AD    format with uppercase specified

```

Two related format controls are:

```
ios::left
```

```
ios::right
```

these control the "justification" of data that is printed in a field of specified width:

```
int main()
{
    cout.width(24);
    cout.fill('!');
    cout << "Hello" << endl;
    cout.setf(ios::left, ios::adjustfield);
    cout.width(24);
    cout << "Hello" << endl;
    cout.setf(ios::right, ios::adjustfield);
    cout.width(24);
    cout << "Hello" << endl;

    return EXIT_SUCCESS;
}
```

should give the output:

```
!!!!!!!!!!!!!!!!!!!!Hello
Hello!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!Hello
```

Note the need to repeat the setting of the field width; unlike fill which lasts, a width setting affects only the next item. As shown by the output, the default justification is right justified. The left/right justification setting uses a slightly different version of the `setf()` function. This version requires two separate data items – the left/right setting and an extra "adjustfield" parameter.

You can specify whether you want doubles printed in "scientific" or "fixed form" styles by setting:

```
ios::scientific
ios::fixed
```

as illustrated in this code fragment:

```
int main()
{
    double number1 = 0.00567;
    double number2 = 1.746e-5;
    double number3 = 9.43214e11;
    double number4 = 5.71e93;
    double number5 = 3.08e-47;

    cout << "Default output" << endl;

    cout << number1 << ", " << number2 << ", "
        << number3 << endl;
    cout << "          " << number4 << ", " << number5 << endl;

    cout << "Fixed style" << endl;
```



```

    cout.setf(ios::fixed, ios::floatfield);

    cout << number1 << ", " << number2 << ", "
          << number3 << endl;
// cout << "      " << number4 ;
    cout << number5 << endl;

    cout << "Scientific style" << endl;
    cout.setf(ios::scientific, ios::floatfield);

    cout << number1 << ", " << number2 << ", "
          << number3 << endl;
    cout << "      " << number4 << ", " << number5 << endl;

    return EXIT_SUCCESS;
}

```

This code should produce the output:

```

Default output
0.00567, 1.746e-05, 9.43214e+11
    5.71e+93, 3.08e-47
Fixed style
0.00567, 0.000017, 943214000000
0
Scientific style
5.67e-03, 1.746e-05, 9.43214e+11
    5.71e+93, 3.08e-47

```

The required style is specified by the function calls `setf(ios::fixed, ios::floatfield)` and `setf(ios::scientific, ios::floatfield)`. (The line with the code printing the number 5.71e93 in "fixed" format style is commented out – some systems can't stand the strain and crash when asked to print such a large number using a fixed format. Very small numbers, like 3.08e-47, are dealt with by just printing zero.)

The various formatting options available with streams should suffice to let you layout any results as you would wish. Many programmers prefer to use the alternative `stdio` library when they have to generate tabular listings of numbers etc. Your IDE environment should include documentation on the `stdio` library; its use is illustrated in all books on C.

*Maybe you should
use `stdio` instead!*

You don't often want to change formatting options on input streams. One thing you might occasionally want to do is set an input stream so that you can actually read white space characters (you might need this if you were writing some text processing program and needed to read the spaces and return characters that exist in an input file). You can change the input mode by unsetting the `ios::skipws` option, i.e.

*Changing options for
input streams*

```

cin.unsetf(ios::skipws);

```

By default, the `skipws` ("skip white space") option is switched on. (You may get into tangles if you unset `skipws` and then try to read a mixture of character and numeric data.)

The following program illustrates use of this control to change input options:

```
int main()
{
    /*
    Program to count the number of characters preceding a
    period '.' character.
    */
    int    count = 0;
    char ch;

    // cin.unsetf(ios::skipws);
    cin >> ch;
    while(ch != '.') {
        count++;
        cin >> ch;
    }
    cout << endl;
    cout << "I think I read " << count << " characters."
        << endl;
    return EXIT_SUCCESS;
}
```

Given the input:

Finished, any questions? OK, go; wake that guy in the back row,
he must have been left here by the previous lecturer.

the output would normally be:

I think I read 95 characters.

but if the `cin.unsetf(ios::skipws);` statement is included, the output would be:

I think I read 116 characters.

because the spaces and carriage returns will then have been counted as well as the normal letters, digits, and punctuation characters.

9.8 EXAMPLE

Problem

Botanists studying soya bean production amassed a large amount of data relating to different soya plantations. These data had been entered into a file so that they could be analyzed by a program.

Some soya plantations fail to produce a harvestable crop. The botanists had no idea of the cause of the problem – they thought it could be disease, or pest infestation, or poor nutrition, or maybe climatic factors. The botanists started by recording information on representative plants from plots in each of the plantations.

Each line of the file contained details of one plant. These details were encoded as 0 or 1 integer values. The first entry on each line indicated whether the plant was rated healthy (1) or unhealthy (0) at harvest time. Each of the other entries represented an attribute considered in the study. These attributes included "late spring frosts", "dry spring", "beetles", "nitrogen deficient soil", "phosphate deficient soil", etc; there were about twenty attributes in the study (for simplicity we will reduce it to five).

The file contains hundreds of lines with entries. Examples of individual entries from the file were:

```
0  1      0      0      0      0
```

(an unhealthy plant that had experienced a late spring frosts but no other problems), and

```
1  1      0      1      0      0
```

(a plant that was healthy despite both a late spring frost and beetles in summer).

The botanists wanted these data analyzed so as to identify any correlations between a plant's health and any chosen one of the possible attributes.

Consider a small test with results for 100 plants. The results might show that 45 were unhealthy, and that beetles had been present for 60 of them. If we assume that the presence of beetles is unrelated to plant health, then there should be a simple random chance distribution of beetles on both healthy and the unhealthy plants. So here, 60% of the 45 diseased plants (i.e. 27 plants) should have had beetle infestations as should 33 of the healthy plants.

Expected distribution

	beetles	clean	
unhealthy	27	18	= 45
healthy	33	22	= 55
	60	40	

While it is unlikely that you would see exactly this distribution, if health and beetle presence are unrelated then the observed results should be something close to these values.

The actual observed distribution might be somewhat different. For example one might find that 34 of the unhealthy plants had had lots of beetles:

Observed distribution

	beetles	clean	
unhealthy	34	11	= 45
healthy	26	29	= 55
	60	40	

Results such as these would suggest that beetles, although certainly not the sole cause of ill health, do seem to make plants less healthy.

Of course, 100 samples is really quite small; the differences between expected and observed results could have arisen by chance. Larger differences between observed and expected distributions occur with smaller chance. If the differences are very large, they are unlikely to have occurred by chance. It does depend on the number of examples studied; a smallish difference may be significant if it is found consistently in very large numbers of samples.

Statisticians have devised all sorts of formulae for working out "statistics" that can be used to estimate the strength of relationships among values of different variables. The botanists' problem is one that can be analyzed using the χ^2 statistic (pronounced \approx "kiye-squared") This statistic gives a single number that is a measure of the overall difference between observed and expected results and which also takes into account the size of the sample.

The χ^2 statistic is easily computed from the entries in the tables of observed and expected distributions:

$$\chi^2 = \sum_{i=1}^4 (O_i - E_i)^2 / E_i$$

O_i value in observed distribution

E_i value in expected distribution

Σ sum is over the four entries in the table

For the example above,

$$\begin{aligned} \chi^2 &= (34-27)^2 / 27 + (11-18)^2 / 18 + (26-33)^2 / 33 \\ &\quad + (29-22)^2 / 22 \\ &\approx 8 \end{aligned}$$

If the observed values were almost the same as the expected values ($O_i \approx E_i$) then the value of χ^2 would be small or zero. Here the "large" value of χ^2 indicates that there is a significant difference between expected and observed distributions of attribute values and class designations. It is very unlikely that such a difference could have occurred solely by chance, i.e. there really is some relation between beetles and plant health.

You can select a chance level at which to abandon the assumption that an attribute (e.g. beetle presence) and a class (e.g. unhealthy) are unrelated. For example, you could say that if the observed results have a less than 1 in 20 chance of occurring then the assumption of independence should be abandoned. You could be more stringent and demand a 1 in 100 chance. Tables exist that give a limit value for the χ^2 statistic associated with different chance levels. In a case like this, if the χ^2 statistic exceeds ≈ 3.8 then an assumption of independence can be abandoned provided that you accept that a 1 in 20 chance as too unlikely. Someone demanding a 1 in 100 chance would use a threshold of ≈ 5.5 .

The botanists would be able to get an idea of what to focus on by performing a χ^2 analysis to measure the relation between plant health and each of the attributes

for which they had recorded data. For most attributes, the observed distributions would turn out to be pretty similar to the expected distributions and the χ^2 values would be small (0...1). If any of the attributes had a large χ^2 value, then the botanists would know that they had identified something that affected plant health.

Specification:

1. The program is to process the data in the file "plants.dat".

This file contains several hundred lines of data. Each line contains details of one plant. The data for a plant take the form of six integer values, all 0 or 1;. The first is the plant classification – unhealthy/healthy. The remaining five are the values recorded for attributes of that plant.

This data value is terminated by a sentinel. This consist of a line with a single number, -1, instead of a class and set of five attributes.

2. The program is to start by prompting for and reading input from `cin`. This input is to be an integer in the range 1 to 5 which identifies the attribute that is to be analyzed in this run of the program.
3. The program is to then read all the data records in the file, accumulating the counts that provide the values for the observed distribution and overall counts of the number of healthy plants etc.
4. When all the data have been read, the program is to calculate the values for the expected distribution.
5. The program is to print details of both expected and observed distributions; these should be printed in tabular form with data entries correctly aligned.
6. Finally, the program is to calculate and print the value of the χ^2 statistic.

Program design

A first iteration through the design process gives a structure something like the following: *Preliminary design*

```
open the data file for input
prompt for and get number identify attribute of interest

read the first class value (or a sentinel if someone
    is trying to trick you with an empty file!)

initialize all counters

while class != sentinel
    read all five attribute values
    select value for the attribute of interest
```

```

        increment appropriate counters
        read next class value or sentinel

calculate four values for expected distribution

print tables of observed and expected distributions

calculate  $\chi^2$ 

print  $\chi^2$ 

```

***Second iteration
through design
process***

There are several points in the initial design outline that require elaboration.

What counters are needed and how can the "observed" and "expected" tables be represented? How can the "attribute of interest" be selected? How should an "appropriate counter" be identified and updated? All of these issues must be sorted out before the complete code is written. However, it will often be the case that the easiest way to record a design decision will be to sketch out a fragment of code.

Each of the tables really needs four variables:

Observed	attribute		Expected	attribute	
	0	1		0	1
unhealthy 0	Obs00	Obs01		Exp00	Exp01
healthy 1	Obs10	Obs11		Exp10	Exp11

Three additional counters are needed to accumulate the data needed to calculate the "expected" values Exp00 ... Exp11; we need a count of the total number of plants, a count of healthy plants, and a count for the number of times that the attribute of interest was true.

Selecting the attribute of interest is going to be fairly easy. The value (1...5) input by the user will have been saved in some integer variable `interest`. In the while loop, we can have five variables `a1 ... a5`; these will get values from the file using an input statement like:

```
infile >> a1 >> a2 >> a3 >> a4 >> a5;
```

We can have yet another variable, `a`, which will get the value of the attribute of interest. We could set `a` using a switch statement:

```

switch(interest) {
case 1: a = a1; break;
...
case 5: a = a5; break;
}

```

(Use of arrays would provide an alternative mechanism for holding the values and selecting the one of interest.)

The values for variables `a` and `pclass` (plant's 0/1 class value) have to be used to update the counters. There are several different ways that these update operations could be coded.

Select an approach that makes the meaning obvious even if this results in somewhat wordier and less efficient code than one of the alternatives. The updating of the counters is going to be a tiny part of the overall processing cost (most time will be devoted to conversion of text input into internal numeric data). There is no point trying to "optimize" this code and make it "efficient" because this is not the code where the program will spend its time.

The counts could be updated using "clever" code like:

```
// Update count of cases with attribute true
atotalpos += a;
healthy += pclass; // and count of healthy plants

Obs11 += pclass && a;
Obs10 += pclass && !a;
```

It should take you less than a minute to convince yourself that those lines of code are appropriate. But fifteen seconds thought per line of code? That code is too subtle. Try something cruder and simpler:

```
#define TRUE 1
...
atotalpos += (a == TRUE) ? 1 : 0;
healthy += (pclass == TRUE) ? 1 : 0;

Obs11 += ((pclass == TRUE) && (a == TRUE)) ? 1 : 0;
```

or even

```
if(a == TRUE) atotalpos++;
...
```

Before implementation, you should also compose some test data for yourself. You will need to test parts of your program on a small file whose contents you know; there is no point touching the botanists' file with its hundreds of records until you know that your program is doing the right thing with data. You would have to create a file with ten to twenty records, say 12 healthy and 6 unhealthy plants. You can pick the attribute values randomly except for one attribute that you make true (1) mainly for the unhealthy plants.

Data

Implementation

This program is just at the level of complexity where it becomes worth building in stages, testing each stage in turn.

The first stage in implementation would be to create something like a program that tries to open an input file, stopping sensibly if the file is not present, otherwise continuing by simply counting the number of records.

First stage partial implementation

```
#include <stdlib.h>
#include <iostream.h>
```

```

#include <fstream.h>

int main()
{
    // Change file name to plants.dat when ready to fly
    ifstream infile("test.dat", ios::in | ios::nocreate);
    int count = 0;

    if(!infile.good()) {
        cout << "Sorry. Couldn't open the file." << endl;
        exit(1);
    }

    int pclass;

    infile >> pclass;

    while(pclass != -1) {
        count++;
        int a1, a2, a3, a4, a5;
        infile >> a1 >> a2 >> a3 >> a4 >> a5;
        // Discard those data values
        // Read 0/1 classification of next plant
        // or sentinel value -1
        infile >> pclass;
    }

    cout << "I think there were " << count
         << " records in that file." << endl;
    return EXIT_SUCCESS;
}

```

***Second stage partial
implementation***

With a basic framework running, you can start adding (and testing) the data processing code. First we could add the code to determine the attribute of interest and to accumulate some of the basic counts.

```

#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>

#define TRUE 1
#define FALSE 0

int main()
{
    // Change file name to plants.dat when ready to fly
    ifstream infile("test.dat", ios::in | ios::nocreate);
    int count = 0;

    if(!infile.good()) {
        cout << "Sorry. Couldn't open the file." << endl;
        exit(1);
    }

    int pclass;

```



```

    int atotalpos = 0;
    int healthy = 0;
    int interest;

    cout << "Please specify attribute of interest"
          "(1...5) : ";
    cin >> interest;

    if((interest < 1) || (interest > 5)) {
        cout << "Sorry, don't understand, wanted"
              " value 1..5" << endl;
        exit(1);
    }

    infile >> pclass;

    while(pclass != -1) {
        count++;
        if(pclass == TRUE)
            healthy++;

        int a1, a2, a3, a4, a5;
        infile >> a1 >> a2 >> a3 >> a4 >> a5;
        int a;
        switch(interest) {
case 1:            a = a1; break;
case 2:            a = a2; break;
case 3:            a = a3; break;
case 4:            a = a4; break;
case 5:            a = a5; break;
        }
        if(a == TRUE)
            atotalpos++;
        infile >> pclass;
    }

    cout << "Data read for " << count << " plants." << endl;
    cout << healthy << " were healthy." << endl;
    cout << "Attribute of interest was " << interest << endl;
    cout << "This was true in " << atotalpos << " cases."
          << endl;
    return EXIT_SUCCESS;
}

```

Note the "sanity check" on the input. This is always a sensible precaution. If the user specifies interest in an attribute other than the five defined in the file there is really very little point in continuing.

The next sanity check is one that you should perform. You would have to run this program using the data in your small test file. Then you would have to check that the output values were appropriate for your test data.

The next stage in the implementation would be to i) define initialized variables that represent the entries for the "observed" and "expected" tables, ii) add code inside the while loop to update appropriate "observed" variables, and iii) provide code that prints the "observed" table in a suitable layout. The variables for the

*Third stage partial
implementation*

"observed" variables could be integers, but it is simpler to use doubles for both observed and expected values. Getting the output in a good tabular form will necessitate output formatting options to set field widths etc. You will almost certainly find it necessary to make two or three attempts before you get the layout that you want.

These extensions require variable definitions just before the while loop:

```
double Obs00, Obs01, Obs10, Obs11;
double Exp00, Exp01, Exp10, Exp11;
Obs00 = Obs01 = Obs10 = Obs11 = Exp00 = Exp01 =
    Exp10 = Exp11 = 0.0;
```

Code in the while loop:

```
if((pclass == FALSE) && (a == FALSE)) Obs00++;
if((pclass == FALSE) && (a == TRUE)) Obs01++;
if((pclass == TRUE) && (a == FALSE)) Obs10++;
if((pclass == TRUE) && (a == TRUE)) Obs11++;
```

Output code like the following just after the while loop:

```
cout.setf(ios::fixed, ios::floatfield);

cout << "Observed distribution" << endl;
cout << "          Attribute " << interest << endl;
cout << "                0          1" << endl;
cout << "Unhealthy" << setw(8) << setprecision(1) << Obs00;
cout << setw(8) << Obs01;
cout << endl;
cout << "Healthy  " << setw(8) << Obs10 << setw(8) << Obs11;
cout << endl;
```

Note that the use of `setprecision()` and `setw()` requires the `iomanip` library, i.e. `#include <iomanip.h>`. You might also chose to use set option `ios::showpoint` to get a value like 7 printed as 7.0.

**Final stage of
implementation**

Finally, you would add the code to compute the expected distributions, print these and calculate the χ^2 statistic. There is a good chance of getting a bug in the code to calculate the expected values:

buggy version

```
Exp10 = healthy*(count-atotalpos)/count;
Exp11 = healthy*atotalpos/count;
```

The formulae are correct in principle. Value `Exp11` is the expected value for the number of healthy plants with the attribute value also being positive. So if, as in the example, there are 55 healthy plants, and of the 100 total plants 60 have beetles, we should get $55 * 60/100$ or 33 as value for `Exp11`. If you actually had those numeric values, there would be no problems and `Exp11` would get the value 33.0. However, if 65 of the 100 plants had had beetles, the calculation would be $55*65/100$ with the correct result 35.75. But this wouldn't be the value assigned to `Exp11`; instead it would get the value 35.

All of the variables `healthy`, `atotalpos`, and `count` are integers. So, by default the calculation is done using integer arithmetic and the fractional parts of the result are discarded.

You have to tell the compiler to code the calculation using doubles. The following code is OK:

```
Exp10 = double(healthy*(count-atotalpos))/count;
Exp11 = double(healthy*atotalpos)/count;
```

bug fixed

The code to print the expected distribution can be obtained by "cutting and pasting" the working code that printed the observed distribution and then changing variable names.

The final few statements needed to calculate the statistic are straightforward:

```
double chisq;
chisq = (Obs00 - Exp00)*(Obs00-Exp00)/Exp00;
chisq += (Obs01 - Exp01)*(Obs01-Exp01)/Exp01;
chisq += (Obs10 - Exp10)*(Obs10-Exp10)/Exp10;
chisq += (Obs11 - Exp11)*(Obs11-Exp11)/Exp11;

cout << "Chisquared value " << chisq << endl;
```

All the variables here are doubles so that there aren't the same problems as for calculating `Exp11` etc. You could make these calculations slightly more efficient by introducing an extra variable to avoid repeated calculation of similar expressions:

```
double chisq;
double temp;
temp = (Obs00 - Exp00);
chisq = temp*temp/Exp00;
temp = (Obs01 - Exp01);
chisq += temp*temp/Exp01;
```

But honestly, it isn't worth your time bothering about such things. The chances are pretty good that your compiler implements a scheme for spotting "common subexpressions" and it would be able to invent such temporary variables for itself.

You would have to run your program on your small example data set and you would need to check the results by hand. For a data set like the following:

```
1 0 0 0 0 0
1 0 1 0 1 0
1 1 0 0 0 1
1 0 1 1 0 1
1 0 1 0 1 1
1 0 1 1 0 1
1 1 0 0 1 0
1 0 1 0 1 0
1 0 0 1 1 0
1 0 1 0 0 1
1 0 0 1 0 1
1 0 0 0 0 1
```

```

0 1 0 1 0 1
0 0 1 0 0 0
0 1 0 1 1 0
0 1 0 1 0 1
0 1 1 0 1 1
0 0 0 1 1 0
0 1 1 0 1 0
-1

```

test runs of the program should produce the following outputs:

```

Observed distribution
Attribute 1
0      1
Unhealthy 2      5
Healthy 10      2
Expected distribution
Attribute 1
0      1
Unhealthy 4.4    2.6
Healthy 7.6     4.4
Chisquared value 5.7

```

and

```

Observed distribution
Attribute 4
0      1
Unhealthy 3      4
Healthy 7      5
Expected distribution
Attribute 4
0      1
Unhealthy 3.7    3.3
Healthy 6.3     5.7
Chisquared value 0.4

```

Once you had confirmed that your results were correct, you could start processing the real file with the hundreds of data records collected by the botanists.

With larger programs, the only way that you can hope to succeed is to complete a fairly detailed design and then implement stage by stage as illustrated in this example.

EXERCISES

1. Modify the bank account program, exercise 3 in Chapter 8, so that it reads transaction details from a file "trans.dat".
2. The file "temperatures.dat" contains details of the oral temperatures of several hundred pre-med, public health, and nursing students. Each line of the file has a number (temperature in degrees Centigrade) and a character ('m' or 'f') encoding the gender of the student. The file ends with a sentinel record (0.0 x). Example:

37.3	m
36.9	f
37.0	m
...	
37.1	f
0.0	x

Write a program to process these data. The program is to calculate and print:

mean (average) temperature, and standard deviation, for all students,
 mean and standard deviation for male students,
 mean and standard deviation for female students.

minimum and maximum temperatures for all students;
 minimum and maximum temperatures for male students;
 minimum and maximum temperatures for female students.

A preliminary study on a smaller sample gave slightly different mean temperatures for males and females. A statistical test is to be made on this larger set of data to determine whether this difference is significant.

This kind of test involves calculation of the "Z" statistic:

Let

μM = average temperature of males

μF = average temperature of females

SM = standard deviation of temperature of males

SF = standard deviation of temperature of females

NM = number of males

NF = number of females

SEM = square of standard error of temperature of males = SM^2 / NM

SEF = square of standard error of temperature of females = SF^2 / NF

$$Z = (\mu M - \mu F) \div \sqrt{SEM + SEF}$$

The statistic Z should be a value with a mean of 0 and a standard deviation of 1.

If there is no significant difference in the two means, then values for Z should be small; there is less than one chance in twenty that you would find a Z with an absolute value exceeding 2.

If a Z value greater than 2 (or less than -2) is obtained, then the statistic suggests that one should NOT presume that the male and female temperatures have the same mean.

Your program should calculate and print the value of this statistic and state whether any difference in means is significant.

10 Functions

The most complex of the programs presented in chapters 6...9 consisted of a single `main()` function with some assignment statements that initialized variables, a loop containing a `switch` selection statement, and a few final output statements. When your data processing task grows more complex than this, you must start breaking it down into separate functions.

We have already made use of functions from the maths library. Functions like `sin()` obviously model the mathematical concept of a function as something that maps (converts) an input value in some domain into an output value in a specified range (so `sin` maps values from the domain $(0..2\pi)$ into the range $(-1.0...1.0)$).

*Functions that
compute values*

We have also used functions from the `iostream` and related libraries. An output statement such as:

```
cout << "Chisquared : " << chisq;
```

actually involves two function calls. The `<<` "takes from" operator is a disguised way of calling a function. The calls here are to a "PrintText" function and to a "PrintDouble" function. Unlike mathematical functions such as `sin()`, these functions don't calculate a value from a given input; instead they produce some desired "side effect" (like digits appearing on the screen). Things like `cout.setf(ios::show point)` again involve a function call executed to achieve a side effect; in this case the side effect is to change the way the numbers would be printed subsequently.

*Functions with "side
effects"*

We will be using an increasing number of different functions from various standard libraries. But now we need more, we need to be able to define our own functions.

If you can point at a group of statements and say "these statements achieve X", then you have identified a potential function. For example, in the program in section 9.8, the following statements were used to "get an integer in a given range"

```
int interest;

cout << "Please specify attribute of interest"
      "(1...5) : ";
```

```

cin >> interest;

if((interest < 1) || (interest > 5)) {
    cout << "Sorry, don't understand, wanted"
         << " value 1..5" << endl;
    exit(1);
}

```

"Get an integer in a given range" – that is a specification of a "function". You have identified a `GetInteger()` operation. It is going to be a function that has to be given two items of data, the low limit and the high limit (1 and 5 in this specific case), and which returns as a result an integer in that range.

Even if, as in the program in section 9.8, this "function" is only needed once, it is worth composing it as a separate function. "Abstracting" the code out and making it a separate function makes the code more intelligible. The program now consists of two parts that can be read, analyzed, and understood in isolation:

```

function to get integer in specified range
    prompt
    input
    check
    ...
    return OK integer

main program
    get checked integer data item using function
    other initialization
    ...

```

The example in section 8.5.2 (Newton's method for roots of a polynomial) illustrates another situation where it would be appropriate to invent a special purpose function. The following expression had to appear at several points in the code where we required the value of the polynomial at a particular value of x :

$$13.5 \cdot \text{pow}(x, 4) - 59 \cdot \text{pow}(x, 3) - 28 \cdot \text{pow}(x, 2) + 16.5 \cdot x + 30$$

It is easier to read and understand code like:

```

fun1 = polyfunction(g1);
fun2 = polyfunction(g2);

```

than code like:

```

double x = g1;
fun1 = 13.5*pow(x,4) - 59*pow(x,3)- 28*pow(x,2)
      + 16.5*x + 30;
x = g2;

```



```
fun2 = 13.5*pow(x,4) - 59*pow(x,3)- 28*pow(x,2)
      + 16.5*x + 30;
```

If you are reading the code with the "calls" to `polyfunction()`, you can see that `fun1` and `fun2` are being initialized with values depending on the input data values `g1`, and `g2`. The *details* of how they are being initialized don't appear and so don't disrupt your chain of thought as you read the code. In addition, the use of a separate function to evaluate the polynomial would actually save some memory. Code using calls to a function would require less instructions than the code with repeated copies of the instructions to evaluate the expression.

A long time ago, function calls were expensive. In the computers of those days, the instruction sets were different and there were limitations on use of CPU registers; together these factors meant that both the processes of setting up a call to a function, and then of getting a result back required many instructions. So, in those days, functions had to be reasonably long to make the overheads of a function call worthwhile. But, that was a very long time ago (it was about the time when the "Beatles", four young men in funny suits, were starting to attract teenagers to the Cavern club in Liverpool).

The overhead for a function call is rarely going to be of any significance on a modern computer architecture. Because functions increase understandability of programs, they should be used extensively. In any case, as explained in 10.6, C++ has a special construct that allows you to "have your cake and eat it". You can define functions and so get the benefit of additional clarity in the code, but at the same time you can tell the compiler to avoid using a function call instruction sequence in the generated code. The code for the body of the function gets repeated at each place where the function would have been called.

10.1 FORM OF A FUNCTION DEFINITION

A function definition will have the following form:

```
return_type
  function_name(
    details of data that must be given to the function
  )
{
  statements performing the work of the function
  return    value of appropriate type ;
}
```

For example:

```
int GetIntegerInRange(int low, int high)
{
```

```

int res;

do {
    cout << "Enter an integer in the range " << low
        << " ... " << high << " :";
    cin >> res;
} while (! ((res >= low) && (res <= high)));
return res;
}

```

The line

```
int GetIntegerInRange(int low, int high)
```

defines the name of the function to be "GetIntegerInRange", the return type `int`, and specifies that the function requires two integer values that its code will refer to as `low` and `high`. The definition of the local variable `res`, and the `do ... while` loop make up the body of the function which ends with the `return res` statement that returns an acceptable integer value in the required range. (The code is not quite the same as that in the example in section 9.8. That code gave up if the data value input was inappropriate; this version keeps nagging the user until an acceptable value is entered.)

Function names

The rules, given in section 6.6, that define names for variables also apply to function names. Function names must start with a letter or an underscore character (`_`) and can contain letters, digits, and underscore characters. Names starting with an underscore should be avoided; these names are usually meant to be reserved for the compiler which often has to invent names for extra variables (or even functions) that it defines. Names should be chosen to summarize the role of a function. Often such names will be built up from several words, like `Get Integer In Range`.

When you work for a company, you may find that there is a "house style"; this style will specify how capitalization of letters, or use of underscores between words, should be used when choosing such composite names. Different "house styles" might require: `GetIntegerInRange`, or `Get_integer_in_range`, or numerous variants. Even if you are not required to follow a specified "house style", you should try to be consistent in the way you compose names for functions.

Argument list

After the function name, you get a parenthesised list of "formal parameters" or argument declarations (the terminology "arguments", "formal parameters" etc comes from mathematics where similar terms are used when discussing mathematical functions). You should note that each argument, like the `low` and `high` arguments in the example, requires its own type specification. This differs from variable definitions, where several variables can be defined together (as in earlier examples where we have had things like `double Obs00, Obs01, Obs10, Obs11;`).

Functions with no arguments

Sometimes, a function won't require arguments. For example, most IDEs have some form of "time used" function (the name of this function will vary). This function, let's call it `TimeUsed()`, returns the number of milliseconds of CPU time that the program has already used. This function simply makes a request to the operating system (using

some special "system call") and interprets the data returned to get a time represented as a long integer. Of course the function does not require any information from the user's program so it has no arguments. Such a function can be specified in two ways:

```
long TimeUsed()
```

i.e. a pair of parentheses with nothing in between (an empty argument list), or as

```
long TimeUsed(void)
```

i.e. a pair of parentheses containing the keyword `void`. ("Void" means empty.) The first form is probably more common. The second form, which is now required in standard C, is slightly better because it makes it even more explicit that the function does not require any arguments.

A call to the `TimeUsed()` function (or any similar function with a void argument list e.g. `int TPSOKTSNW(void)`) just has the form:

```
TPSOKTSNW()
```

*Trap for the unwary
Pascal programmer
(or other careless
coder)*

e.g.

```
if(TPSOKTSNW()) {  
    // action  
    ...  
}
```

In Pascal, and some other languages, a function that takes no arguments is called by just writing its function name without any `()` parentheses. People with experience in such languages sometimes continue with their old style and write things like:

```
if(TPSOKTSNW) {  
    // action  
    ...  
}
```

This is perfectly legal C/C++ code. Its meaning is a bit odd – it means check whether the function `TPSOKTSNW()` has an address. Now if the program has linked successfully, `TPSOKTSNW()` will have an address so the `if` condition is always true. (The function `TPSOKTSNW()` – it was "The_President_Says_OK_To_Start_Nuclear_War()", but that name exceeded the length limits set by the style guides for the project. Fortunately, the code was checked before being put into use.)

10.2 RESULT TYPES

Like mathematical functions that have a value, most functions in programs return a value. This value can be one of the standard data types (`char`, `long`, `double`, etc) or may be a structure type (Chapter 16) or pointer type (introduced in Part IV, Chapter 20).

*Default return type of
int*

If you don't define a return type, it is assumed that it should be `int`. This is a left over from the early days of C programming. You should always remember to define a return type and not simply use this default (which may disappear from the C++ language sometime).

The compiler will check that you do return a value of the appropriate type. You will probably already have had at least one error report from the compiler on some occasion where you forgot to put a `return 0;` statement at the end of your `int main()` program.

A return statement will normally have an expression; some examples are:

```
return res;

return n % MAXSIZE;

return 3.5 + x*(6.1 + x*(9.7 + 19.2*x));
```

You will sometimes see code where the style convention is to have an extra pair of parentheses around the expression:

```
return(res);

return(n % MAXSIZE);

return(3.5 + x*(6.1 + x*(9.7 + 19.2*x)));
```

Don't worry about the extra parentheses; they are redundant (don't go imagining that it is a call to some strange function called `return()`).

*Functions that don't
return results*

If you have a function that is used only to produce a side effect, like printing some results, you may not want to return a result value. In the 1970s and early 1980s when earlier versions of C were being used, there was no provision for functions that didn't return results. Functions executed to achieve side effects were expected to return an integer result code reporting their success or failure. You can still see this style if you get to use the alternate stdio input-output library. The main print function, `printf()`, returns an integer; the value of this integer specifies the number of data items successfully printed. If you look at C code, you will see that very few programmers ever check the success status code returned by `printf()`. (Most programmers feel that if you can't print any results there isn't anything you can do about it; so why bother checking).

void

Since the success/failure status codes were so rarely checked, the language designers gave up on the requirement that all "side effect" functions return an integer success

code. C++ introduced the idea of functions with a `void` return type (i.e. the return value slot is empty). This feature was subsequently retrofitted into the older C language.

For example, suppose you wanted to tidy up the output part the example program from section 8.5.1 (the example on childrens' heights). The program would be easier to follow if there was a small separate function that printed the average and standard deviation. This function would be used purely to achieve the side effect of producing some printout and so would not have a return value.

It could be defined as follows:

```
void PrintStats(int num, double ave, double std_dev)
{
    cout << "The sample size was " << num << endl;
    cout << "Average height " << ave;
    cout << "cm, with a standard deviation of " <<
        std_dev << endl;
    return;
}
```

The `return;` statement at the end can be omitted in a `void` function.

10.3 FUNCTION DECLARATIONS

When you start building larger programs where the code must be organized in separate files, you will often need to "declare" functions. You will have defined your function in one file, but code in another file needs a call to that function. The second file will need to contain a declaration describing the function, its return type, and its argument values. This is necessary so that the compiler (and linker) can know that the function does exist and so that they can check that it is being used correctly (getting the right kinds of data to work with, returning a result that suits the code of the call).

A function declaration is like a function definition – just without the code! A declaration has the form:

```
return_type
function_name(
    details of data that must be given to the function
) ;
```

The end of a *declaration* has to be marked (because otherwise the compiler, having read the argument list, will be expecting to find a `{` begin bracket and some code). So, function declarations end with a semicolon after the closing parenthesis of the argument list. (Similarly, it is a mistake to put a semicolon after the closing parenthesis and before the body in a function *definition*. When the compiler sees the semicolon, it "knows" that it just read a function declaration and so is expecting to find another

**Function
"prototypes"**

declaration, or the definition of a global variable or constant. The compiler gets very confused if you then hit it with the { *code* } parts of a function definition.)

Examples of function declarations (also known as function "prototypes") are:

```
void PrintStats(int num, double ave, double std_dev);
int GetIntegerInRange(int low, int high);
```

In section 6.5.2, part of the math.h header file was listed; this contained several function declarations, e.g.:

```
double sin(double);
double sqrt(double);
double tan(double);
double log10(double);
```

**Most declarations are
in header files**

Most function declarations appear in header files. But, sometimes, a function declaration appears near the start of a file with the definition appearing later in the same file.

Unlike languages such as Pascal (which requires the "main program" part to be the last thing in a file), C and C++ don't have any firm rules about the order in which functions get defined.

For example, if you were rewriting the heights example (program 8.5.1) to exploit functions you could organize your code like this:

```
#include <iostream.h>
#include <math.h>

double Average(double total, int number)
{
    return total/number;
}

double StandardDev(double sumsquares, double average, int num)
{
    return sqrt(
        (sumsquares - num*average*average) /
        (num-1));
}

void PrintStats(int num, double ave, double std_dev)
{
    ...
}

int main()
{
    int        children, boys, girls;
    double     cSum, bSum, gSum;
```

```
double      cSumSq, bSumSq, gSumSq;

...

cout << "Overall results all children:" << endl;
average = Average(cSum, children);
standard_dev = StandardDev(cSumSq, average children);
PrintStats(children, average, standard_dev);

cout << endl;
cout << "Results for girls only:" << endl;

...

return 0;
}
```

Alternatively, the code could be arranged like this:

```
#include <iostream.h>
#include <math.h>
double Average(double total, int number);
double StandardDev(double sumsquares, double average, int num);
void PrintStats(int num, double ave, double std_dev);

int main()
{
    int      children, boys, girls;
    double   cSum, bSum, gSum;
    double   cSumSq, bSumSq, gSumSq;

    ...

    cout << "Overall results all children:" << endl;
    average = Average(cSum, children);
    standard_dev = StandardDev(cSumSq, average children);
    PrintStats(children, average, standard_dev);

    cout << endl;

    cout << "Results for girls only:" << endl;

    ...

    return 0;
}

void PrintStats(int num, double ave, double std_dev)
{
    ...
}
```

Forward declarations

```
double Average(double total, int number)
{
    return total/number;
}

double StandardDev(double sumsquares, double average, int num)
{
    return sqrt(
        (sumsquares - num*average*average) /
        (num-1));
}
```

This version requires declarations for functions `Average()` etc at the start of the file. These declarations are "forward declarations" – they simply describe the functions that are defined later in the file.

These forward declarations are needed for the compiler to deal correctly with the function calls in the main program (`average = Average(cSum, children);` etc). If the compiler is to generate appropriate code, it must know that `Average()` is a function that returns a double and takes two arguments – a double and an integer. If you didn't put in the forward declaration, the compiler would give you an error message like "Function `Average` has no prototype."

Normally, you should use the first style when you are writing small programs that consist of `main()` and a number of auxiliary functions all defined in the one file. Your programs layout will be vaguely reminiscent of Pascal with `main()` at the bottom of the file. There are a few circumstances where the second arrangement is either necessary or at least more appropriate; examples will appear later.

There is another variation for the code organization:

*Prototypes declared
in the body of the
code of caller*

```
#include <iostream.h>
#include <math.h>

int main()
{
    double Average(double total, int number);
    double StandardDev(double sumsquares,
        double average, int num);
    void PrintStats(int num, double ave, double std_dev);

    int      children, boys, girls;
    double    cSum, bSum, gSum;

    ...
    average = Average(cSum, children);
    standard_dev = StandardDev(cSumSq, average children);
    PrintStats(children, average, standard_dev);
    ...
    return 0;
}
```



```
void PrintStats(int num, double ave, double std_dev)
{
    ...
}

double Average(double total, int number)
{
    return total/number;
}

double StandardDev(double sumsquares, double average, int num)
{
    ...
}
```

There is no advantage to this style. It introduces an unnecessary distinction between these functions and those functions, like `sqrt()`, whose prototypes have appeared in the header files that have been `#included`. Of course, as you would realize, a declaration of a function prototype in the body of another function does *not* result in that function being called at the point of declaration!

10.4 DEFAULT ARGUMENT VALUES

Some functions need to be given lots of information, and therefore they need lots of arguments. Sometimes, the arguments may almost always get to be given the same data values.

For example, if you are writing a real window-based application you will not be using the `iostream` output functions, instead you will be using some graphics functions. One might be `DrawString()`, a function with the following specification:

```
void DrawString(char Text[], int TextStyle,
                int TextSize, int Hoffset, int Voffset)
```

This function plots the message `Text` in the current window.

`TextStyle` defines the style, 0 Normal, 1 Italic, 2 Bold, 3 Bold Italic.

`TextSize` defines the size, allowed sizes are 6, 9, 10, 12, 14, 18, 24, 36, 48.

`Hoffset` and `Voffset` are the horizontal offsets for the start of the string relative to the current pen position

In most programs, the current pen position is set before a call to `DrawString()` so, usually, both `Hoffset` and `Voffset` are zero. Graphics programs typically have a default font size for text display, most often this is 12 point. Generally, text is

displayed in the normal ("Roman") font with only limited use of italics and bold fonts. About the only thing that is different in every call to `DrawString()` is the message.

So, you tend to get code like:

```
...
DrawString("Enter your bet amount", 0, 12, 0, 0);
...
DrawString("Press Roll button to start game", 0, 12, 0, 0);
...
DrawString("Lock any columns that you don't want rolled again",
           0, 12, 0, 0);
...
DrawString("Congratulations you won", 3, 24, 0, 0);
...
DrawString("Double up on your bet?", 3, 12, 0, 0);
...
```

C++ permits the definition of default argument values. If these are used, then the arguments don't have to be given in the calls.

If you have a declaration of `DrawString()` like the following:

```
void DrawString(char Text[], int TextStyle = 0,
               int TextSize = 12,
               int Hoffset = 0, int Voffset = 0);
```

This declaration would allow you to simplify those function calls:

*Omitting trailing
arguments that have
default values*

```
...
DrawString("Enter your bet amount");
...
DrawString("Press Roll button to start game");
...
DrawString("Lock any columns that you don't want rolled
again");
...
DrawString("Congratulations you won", 3, 24);
...
DrawString("Double up on your bet?", 3);
...
```

You are permitted to omit the last argument, or last pair of arguments, or last three arguments or whatever if their values are simply the normal default values. (The compiler just puts back any omitted values before it goes and generates code.) You are only allowed to omit trailing arguments, you aren't allowed to do something like the following:

*Can't arbitrarily omit
arguments*

```
DrawString("Congratulations you won", , 24);
```

Default arguments can appear in function declarations, or in the definition of the function (but not both). Either way, details of these defaults must be known to the compiler before the code where a call is made. These requirements affect the layout of programs.

10.5 TYPE SAFE LINKAGE, OVERLOADING, AND NAME MANGLING

The C++ compiler and its associated linking loader don't use the function names that the programmer invented!

Before it starts into the real work of generating instructions, the C++ systematically renames all functions. The names are elaborated so that they include the name provided by the programmer and details of the arguments. The person who wrote the compiler can choose how this is done; most use the same general approach. In this approach, codes describing the types of arguments are appended to the function name invented by the programmer.

Using this kind of renaming scheme, the functions `TimeUsed`, `Average`, `StandardDev`, and `PrintStats`:

```
void TimeUsed(void);
double Average(double total, int number);
double StandardDev(double sumsquares,
                   double average, int num);
void PrintStats(int num, double ave, double std_dev);
```

would be assigned compiler generated (or "mangled") names like:

Mangled names

```
__TimeUsed__fv
__Average__fdfi
__StandardDev_f2dfi
__PrintStats__fif2d
```

You might well think this behaviour strange. Why should a compiler writer take it on themselves to rename your functions?

One of the reasons that this is done is this makes it easier for the compiler and linking loader to check consistency amongst separate parts of a program built up from several files. Suppose for instance you had a function defined in one file: *Type safe linkage*

```
void Process(double delta, int numsteps)
{
    ...
}
```

In another file you might have incorrectly declared this function as:

```
void Process(int numsteps, double delta);
```

and made calls like:

```
Process(11, 0.55);
```

If there were no checks, the program would be compiled and linked together, but it would (probably) fail at run-time and would certainly produce the wrong results because the function `Process()` would be getting the wrong data to work with.

The renaming process prevents such run-time errors. In the first file, the function gets renamed as:

```
__Process__fdfi
```

while in the second file it is:

```
__Process__fifd
```

When the linking loader tried to put these together it would find the mismatch and complain (the error message would probably say something like "Can't find `__Process_fifd`", this is one place where the programmer sometimes gets to see the "mangled" names).

***Suppressing the
name mangling
process for C
libraries***

This renaming occasionally causes problems when you need to use a function that is defined in an old C language library. Suppose the C library has a function `void seed(int)`, normally if you refer to this in a piece of C++ code the compiler renames the function making it `__seed__fi`. Of course, the linking loader can't find this function in the C library (where the function is still called `seed`). If you need to call such a function, you have to warn the C++ compiler *not* to do name mangling. Such a warning would look something like:

```
extern "C" {  
void seed(int);  
}
```

It is unlikely that you will encounter such difficulties when working in your IDE because it will have modernised all the libraries to suit C++. But if you get to use C libraries from elsewhere, you may have to watch out for this renaming problem.

Overloaded names

Another affect of this renaming is that a C++ compiler would see the following as distinct functions:

```
short abs(short);  
long abs(long);  
double abs(double);
```

after all it sees these as declaring functions called:

```
__abs__fs
__abs__fl
__abs__fd
```

This has the advantage that the programmer can use the single function name `abs()` (take the absolute value of ...) to describe the same operation for a variety of different data types. The name `abs()` has acquired multiple (though related) meanings, but the compiler is smart enough to keep them distinct. Such a name with multiple meanings is said to be *overloaded*.

It isn't hard for the compiler to keep track of what is going on. If it sees code like:

```
double x;
...
double v = -3.9 + x*(6.5 + 3.9*x);

if(abs(v) > 0.00001)
...
```

The compiler reasons that since `v` is a double the version of `abs()` defined for doubles (i.e. its function `__abs__fd`) is the one that should be called.

This approach, which is more or less unique to C++, is much more attractive than the mechanisms used in other languages. In most languages, function names must be distinct. So, if a programmer needs an absolute value function (just checking, you do know that that simply means you force any \pm sign to be + don't you?) for several different data types, then several separately named functions must be defined. Thus, in the libraries used by C programs, you have to have:

```
double fabs(double);
long labs(long);
int abs(int);
```

Note that because of a desire for compatibility with C, most versions of the maths library do *not* have the overloaded `abs()` functions. You still have to use the specialized `fabs()` for doubles and so forth. C++ will let you use `abs()` with doubles – but it truncates the values to integers so you lose the fraction parts of the numbers.

Note: math library does NOT have overloaded definitions of `abs()`

Function `abs()` illustrates the use of "overloading" so that a common name is used to describe equivalent operations on different data. There is a second common use for overloading. You will most likely encounter this use in something like a graphics package. There you might well find a set of overloaded `DrawRectangle()` functions:

Overloading for alternative interface

```
void DrawRectangle(int left, int top, int width, int height);
void DrawRectangle(Rectangle r);
void DrawRectangle(Point topleft, int width, int height);
void DrawRectangle(Point topleft, Point dimension);
```

There is one basic operation of drawing a rectangle. Sometimes, the data available are in the form of four integers – it is then convenient to use the first overloaded function. Sometimes you may have a "Rectangle" data structure (a simple structure, see Chapter 17, that has constituent data elements defining top-left and bottom-right corners); in this case you want the second variant. Sometimes your data may be in the form of "Point" data structures (these contain an 'x' integer value, and a 'y' integer value) and either the third or fourth form of the function would be convenient. Here, overloading is used to provide multiple interfaces to a common operation.

Should you use overloading and define multiple functions with the same base name?

Probably not. Treat this as a "read only feature" of C++. You need to be able to read code that uses this feature, but while you are a beginner you shouldn't write such code. Stick to the KISS principle (remember Keep It Simple Stupid). Fancy styles like overloaded functions are more likely to get you into trouble than help you in your initial efforts.

10.6 HOW FUNCTIONS WORK

Section 4.8, on Algol, introduced the idea of a "stack" and showed how this could be used both to organize allocation of memory for the variables of a function, and to provide a mechanism for communicating data between a calling function and the function that it calls. It is worth looking again at these details.

The following code fragment will be used as an example:

```
#include <iostream.h>

Function definition int GetIntegerInRange(int low, int high)
{
    int res;
    do {
        cout << "Enter an integer in the range " << low
            << " ... " << high << " :";
        cin >> res;
    } while (! ((res >= low) && (res <= high)));
    => return res;
}

int main()
{
    const int LOWLIMIT = 1;
    int highval;
    cout << "Please enter high limit for parameter : ";
    cin >> highval;
    ...
    int val;
    ...
}
```

```
⇒ val = GetIntegerInRange(LOWLIMIT, highval);  
...  
}
```

Function call

Imagine that the program is running, the value 17 has been entered for `highval`, the function call (marked \Rightarrow) has been made, and the function has run almost to completion and is about to return a value 9 for `res` (it is at the point marked \Rightarrow). The state of the "stack" at this point in the calculation would be as shown in Figure 10.1.

The operating system will allocate some specific fixed area in memory for a program to use for its stack. Usually, the program starts using up this space starting at the highest address and filling down.

Each routine has a "frame" in the stack. The compiler will have sorted this out and worked out the number of bytes that a routine requires. Figure 10.1 shows most of the stack frame for the main program, and the frame for the called `GetIntegerInRange` function.

Local, "automatic", variables are allocated space on the stack. So the frame for `main()` will include the space allocated for variables such as `highval`; these spaces are where these data values are stored.

The compiler generates "calling" code that first copies the values of arguments onto the stack, as shown in Figure 10.1 where the data values 1 and 17 appear at the top of the "stack frame" for `GetIntegerInRange()`. The next slot in the stack would be left blank; this is where the result of the function is to be placed before the function returns.

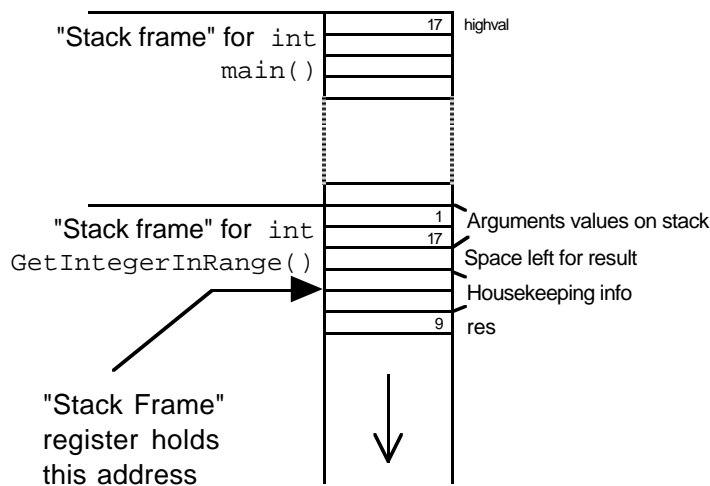


Figure 10.1 Stack during function call

Following the code that organizes the arguments, the compiler will generate a subroutine call instruction. When this gets executed at run-time, more "housekeeping information" gets saved on the stack. This housekeeping information will include the "return address" (the address of the instruction in the calling program that immediately follows the subroutine call). There may be other housekeeping information that gets saved as well.

The first instructions of a function will normally claim additional space on the stack for local variables. The function `GetIntegerInRange()` would have to claim sufficient space to store its `res` variable.

When a program is running, one of the cpu's address registers is used to hold the address of the current stack frame (the address will be that of some part of the housekeeping information). As well as changing the program counter, and saving the old value of the program counter on the stack, the subroutine call instruction will set this stack frame register (or "stack frame pointer `sfp`").

A function can use the address held in this stack frame pointer to find its local variables and arguments. The compiler determines the arrangement of stack frames and can generate address specifications that define the locations of variables by their offsets relative to the stack frame pointer. The `res` local variable of function `GetIntegerInRange()` might be located 8-bytes below the address in the stack frame pointer. The location for the return value might be 12-bytes above the stack frame pointer. The argument `high` might be 16-bytes above the stack frame pointer, and the argument `low` would then be 20-bytes above the stack frame pointer ("`sfp`"). The compiler can work all of this out and use such information when generating code for the function.

The instruction sequences generated for the function call and the function would be along the following lines:

Caller	<pre>// place arguments 1, and highval on stack copy 1 onto stack copy highval onto stack subroutine call copy result from stack into val tidy up stack</pre>
Called function	<pre>claim stack space for locals (i.e. res) loop: code calling the output and input routines, result of input operation put in stack location for res load a CPU register from memory location 8 below sfp (res) load another CPU register from memory 20 above sfp (low) compare registers jump if less to loop // res is too small load other CPU from memory 16 above sfp (high) compare registers jump if greater to loop // res is too large // res in range store value from register into location 12 above sfp (slot)</pre>


```
    for result)
return from subroutine
```

As illustrated, a function has space in the stack for its local variables and its arguments (formal parameters). The spaces on the stack for the arguments are initialized by copying data from the actual parameters used in the call. But, they are quite separate. Apart from the fact that they get initialized with the values of the actual parameters, these argument variables are just like any other local variable. Argument values can be incremented, changed by assignment or whatever. Changes to such value parameters don't have any affect outside the function.

10.7 INLINE FUNCTIONS

A one-line function like:

```
double Average(double total, int number)
{
    return total/number;
}
```

really is quite reasonable. In most circumstances, the overheads involved in calling this function and getting a result returned will be negligible. The fact that the function is defined helps make code more intelligible.

But, sometimes it isn't appropriate to pay the overhead costs needed in calling a function even though the functional abstraction is desired because of the way this enhances intelligibility of code.

When might you want to avoid function call overheads?

Usually, this will be when you are doing something like coding the "inner loop" of some calculation where you may have a piece of code that you know will be executed hundreds of millions of times in a single run of a program, cutting out a few instructions might save the odd minute of cpu time in runs taking several hours. Another place where you might want to avoid the extra instructions needed for a function call would be in a piece of code that must be executed in the minimal time possible (e.g. you are writing the code of some real time system like the controller in your car which has to determine when to inflate the air-bags).

C++ has what it calls "inline" functions:

```
inline double Average(double total, int number)
{
    return total/number;
}
```

This presents the same abstraction as any other function definition. Here `Average()` is again defined as a function that takes a double and an integer as data and which returns

their quotient as a double. Other functions can use this `Average()` function as a primitive operation and so avoid the details of the calculation appearing in the code of the caller.

The extra part is the `inline` keyword. This is a hint to the compiler. The programmer who defined `Average()` in this way is suggesting to the compiler that it should try to minimize the number of instructions used to calculate the value.

Consider a typical call to function `Average()`:

```
double girls_total, boys_total;
int num;
...
...
double ave = Average(girls_total + boys_total, num);
```

The instruction sequence normally generated for the call would be something like the following:

```
// Preparing arguments in calling routine
load double value girls_total into a floating point register
add double value boys_total to register
copy double result "onto stack"
copy integer value num "onto stack"
// make call, subroutine call instruction places more info
// in stack
subroutine call
// Calling code executed after return from routine
extract double result from stack and place
in floating point register
clear up stack
```

The code for `Average()` would be something like:

```
copy double value total from place in stack
to floating point register
copy integer value num into integer register
use "float" instruction to get floating point version of num in
second floating point register
use floating point divide instruction to get quotient (left in
first floating point register)
store quotient at appropriate point in stack
return from subroutine
```

The italicised instructions in those code sequences represent the "overhead" of a function call.

If you specify that the `Average()` be an inline function, the compiler will generate the following code:

```
load double value girls_total into a floating point register
```

```
add double value boys_total to register
load num into an integer register
use "float" instruction to get floating point version of num in
    second floating point register
use floating point divide instruction to get quotient
```

This is a rather extreme example in that the `inline` code does require significantly fewer instructions.

If things really are going to be time critical, you can request `inline` expansion for functions. You should only do this for simple functions. (Compilers interpretations of "simple" vary a little. Basically, you shouldn't declare a function `inline` if it has loops, or if it has complex selection constructs. Compilers are free to ignore your hints and compile the code using the normal subroutine call sequence; most compilers are polite enough to tell you that they ignored your `inline` request.)

Of course, if a compiler is to "expand" out the code for a function `inline`, then it must know what the code is! The definition of an `inline` function must appear in any file that uses it (either by being defined there, or by being defined in a file that gets `#included`). This definition must occur before the point where the function is first used.

10.8 A RECURSIVE FUNCTION

The idea of recursion was introduced in section 4.8. Recursion is a very powerful problem solving strategy. It is appropriate where a problem can be split into two subproblems, one of which is easy enough to be solved immediately and the second of which is simply the original problem but now with the data slightly simplified.

Recursive algorithms are common with problems that involve things like finding ones way through a maze. (Maze walk problem: moving from room where you are to exit room; solution, move to an adjacent room and solve maze walk for that room.) Surprisingly, many real programming problems can be transformed into such searches.

Section 4.8 illustrated recursion with a pseudo-code program using a recursive number printing function. How does recursion help this task? The problem of printing a number is broken into two parts, printing a digit (easy), and printing a (smaller) number (which is the original problem in simpler form). The recursive call must be done first (print the smaller number) then you can print the final digit. Here it is implemented in C++:

```
#include <stdlib.h>
#include <iostream.h>

void PrintDigit(short d)
{
    switch(d) {
        case 0 : cout << '0'; break;
        case 1 : cout << '1'; break;
```

```

        case 2 : cout << '2'; break;
        case 3 : cout << '3'; break;
        case 4 : cout << '4'; break;
        case 5 : cout << '5'; break;
        case 6 : cout << '6'; break;
        case 7 : cout << '7'; break;
        case 8 : cout << '8'; break;
        case 9 : cout << '9'; break;
        }
    }

Recursive function void      PrintNumber(long n)
{
    if(n < 0) {
        cout << "-";
        PrintNumber(-n);
    }
    else {
        long    quotient;
        short   remainder;
        quotient = n / 10;
        remainder = n % 10;
        if(quotient > 0)
            PrintNumber(quotient);
        PrintDigit(remainder);
    }
}

Recursive call in
function

int main()
{
    PrintNumber(9623589); cout << endl;
    PrintNumber(-33); cout << endl;
    PrintNumber(0); cout << endl;
    return 0;
}

```

Enter this program and compile with debugging options on. Before running the program, use the debugger environment in your IDE to set a breakpoint at the first line of the `PrintDigit()` function. Then run the program.

The program will stop and return control to the debugger just before it prints the first digit (the '9' for the number 9623589). Use the debugger controls to display the "stack backtrace" showing the sequence of function calls and the values of the arguments. Figure 10.2 illustrates the display obtained in the Symantec 8 environment.

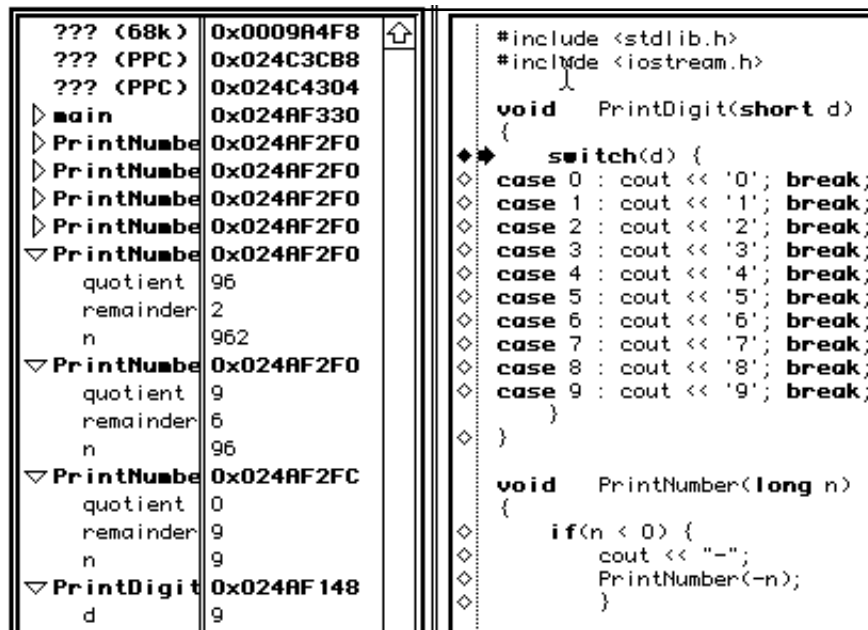


Figure 10.2 A "stack backtrace" for a recursive function.

The stack backtrace shows the sequence of calls: `PrintDigit()` was called from `PrintNumber()` which was called from `PrintNumber()` ... which was called from `main()`. The display shows the contents of the stack frames (and, also, the return addresses) so you can see the local variables and arguments for each level of recursion.

You should find the integrated debugger a very powerful tool for getting to understand how functions are called; it is particularly useful when dealing with recursion.

10.9 EXAMPLES OF SIMPLE FUNCTIONS

10.9.1 GetIntegerInRange

A simple version of this function was introduced in 10.1:

```
int GetIntegerInRange(int low, int high)
{
    int res;
    do {
        cout << "Enter an integer in the range " << low
             << " ... " << high << " :";
```

```

        cin >> res;
    } while (! ((res >= low) && (res <= high)));
    return res;
}

```

Well, it is conceptually correct. But if you were really wanting to provide a robust function that could be used in any program you would have to do better.

What is wrong with the simple function?

First, what will happen if the calling code is mixed up with respect to the order of the arguments. (*"I need that GetIntegerInRange; I've got to give it the top and bottom limits – count = GetIntegerInRange(100,5)."*) A user trying to run the program is going to find it hard to enter an acceptable value; the program will just keep nagging away for input and will keep rejecting everything that is entered.

There are other problems. Suppose that the function is called correctly (`GetIntegerInRange(5,25)`) but the user mistypes a data entry, keying `y` instead of 7. In this case the `cin >> res` statement will fail. Error flags associated with `cin` will be set and the value 0 will be put in `res`. The function doesn't check the error flag. But it does check the value of `res`. Is `res` (0) between 5 and 25? No, so loop back, reprompt and try to read a value for `res`.

That is the point where it starts to be "fun". What data are waiting to be read in the input?

The next input data value is that letter `y` still there waiting to be read. The last read routine invoked by the `cin >>` operation didn't get rid of the `y`. The input routine simply saw a `y` that couldn't be part of a numeric input and left it. So the input operation fails again. Variable `res` is still 0. Consequently, the test on `res` being in range fails again. The function repeats the loop, prompting for an input value and attempting to read data.

Try running this version of the function with such input data (you'd best have first read your IDE manuals to find how to stop an errant program).

Fundamental law of programming

Remember Murphy's law: *If something can go wrong it will go wrong.* Murphy's law is the fundamental law of programming. (I don't know whether it is true, but I read somewhere that the infamous Mr. Murphy was the leader of one of IBM's first programming projects back in the early 1950s.)

You need to learn to program defensively. You need to learn to write code that is not quite as fragile as the simple `GetIntegerInRange()`.

Verify the data given to a function

One defensive tactic that you should always consider is verifying the data given to a function. If the data are wrong (e.g. `high < low`) then the function should not continue.

What should a function do under such circumstances?

In Part IV, we will explore the use of "exceptions". Exceptions are a mechanism that can sometimes be used to warn your caller of problems, leaving it to the caller to sort out a method of recovery. For the present, we will use the simpler approach of terminating the program with a warning message.

A mechanism for checking data, and terminating a program when the data are bad, is so commonly needed that it has been packaged and provided in one of the standard libraries. Your IDE will have a library associated with the `assert.h` header file. *Using assert*

This library provides a "function" known as `assert()` (it may not be implemented as a true function, some systems use "macros" instead). Function `assert()` has to be given a "Boolean" value (i.e. an expression that evaluates to 0, False, or non-zero True). If the value given to `assert()` is False, the function will terminate the program after printing an error message with the general form "Assertion failed in line ... of function ... in file ...".

We would do better starting the function `GetIntegerInRange()` as follows:

```
#include <stdlib.h>
#include <iostream.h>
#include <limits.h>
#include <assert.h>

long GetIntegerInRange(long low = LONG_MIN,
    long high = LONG_MAX)
{
    assert(low <= high);
```

The function now starts with the `assert()` check that `low` and `high` are appropriate. Note the `#include <assert.h>` needed to read in the declaration for `assert()`. (Another change: the function now has default values for `low` and `high`; the constants `LONG_MAX` etc are taken from `limits.h`.)

You should always consider using `assert()` to validate input to your functions. You can't always use it. For example, you might be writing a function that is supposed to search for a specific value in some table that contains data "arranged in ascending order"; checking that the entries in the table really were in ascending order just wouldn't be worth doing.

But, in this case and many similar cases, validating the arguments is easy. Dealing with input errors is a little more difficult.

Some errors you can't deal with. If the input has gone "bad", there is nothing you can do. (This is going to be pretty rare; it will necessitate something like a hardware failure, e.g. unreadable file on a floppy disk, to make the input go bad.)

Handling input errors

If all the data in a file has been read, and the function has been told to read some more, then once again there is nothing much that can be done.

So, for bad input, or end of file input, the function should simply print a warning message and terminate the program.

If an input operation has failed but the input stream, `cin`, is not bad and not at end of file, then you should try to tidy up and recover. The first thing that you must do is clear the fail flag on `cin`. If you don't do this, anything else you try to do to `cin` is going to be ignored. So, the first step toward recovery is:

Clearing the failure flag

```
cin.clear();
```

(The `clear()` operation is another of the special functions associated with streams like the various format setting `setf()` functions illustrated in 9.7.)

Next, you have to deal with any erroneous input still waiting to be read (like the letter `y` in the discussion above). These characters will be stored in some data area owned by the `cin` stream (the "input buffer"). They have to be removed.

We are assuming that input is interactive. (Not much point prompting a file for new data is there!) If the input is interactive, the user types some data and ends with a return. So we "know" that there is a return character coming.

We can get rid of any queued up characters in the input buffer if we tell the `cin` stream to "ignore" all characters up to the return (coded as `'\n'`). An input stream has an `ignore()` function, this takes an integer argument saying how many characters maximum to skip, and a second argument identifying the character that should mark the end of the sequence that is to be skipped. A call of the form:

```
cin.ignore(SHRT_MAX, '\n');
```

should jump over any characters (up to 30000) before the next return.

All of this checking and error handling has to be packaged in a loop that keeps nagging away at the user until some valid data value is entered. One possible coding for this loop is:

<p><i>Prompting</i></p> <p><i>Trying to read</i></p> <p><i>Successful read?</i> <i>Return an OK value</i></p> <p><i>Warn on out of range</i> <i>and loop again</i></p> <p><i>Look for disasters</i></p>	<pre> for(;;) { cout << "Enter a number in the range " << low << " to " << high << " : "; int val; cin >> val; if(cin.good()) if((val >= low) && (val <= high)) return val; else { cout << val << " is not in required range." << endl; continue; } if(cin.bad()) { cout << "Input has gone bad, can't read data; " << "quitting." << endl; exit(1); } if(cin.eof()) { cout << "Have unexpected end of file; quitting." << endl; exit(1); } } </pre>
---	---


```

cin.clear();
cout << "Bad data in input stream, flushing buffer."
      << endl;

cin.ignore(SHRT_MAX, '\n');
}

```

*Must just be bad data
in input, clear up*

This is an example of a "forever" loop. The program is stuck in this loop until either an acceptable value is entered or something goes wrong causing the program to terminate. The loop control statement:

```
for(;;)
```

has nothing in the termination test part (i.e. nothing terminates this loop).

After the read (`cin >> val`), the first check is whether `cin` is "good". It will be good if it was able to read an integer value and store this value in variable `val`. Of course, the value entered might have been an integer, but not one in the acceptable range. So, that had better be checked. Note the structure of the nested ifs here. *If* `cin` is good *then if* the number is in range return *else* complain and loop.

Here the return statement is in the middle of the body of the loop. This style will offend many purists (who believe that you should only return from the end point of the function). However, it fits logically with the sequence of tests and the code in this form is actually less complex than most variants.

Note also the use of `continue`. This control statement was introduced in section 7.8 but this is the first example where it was used. If the user has entered an out of range value, we need to print a warning and then repeat the process of prompting for and reading data. So, after the warning we want simply to "continue" with the next iteration of the for loop.

The remaining sections of the loop deal with the various input problems already noted and the attempt to clean up if inappropriate data have been entered.

Although it won't survive all forms of misuse, this version of `GetIntegerInRange()` is a little more robust. If you are writing functions that you hope to use in many programs, or which you hope to share with coworkers, you should try to toughen them like this.

10.9.2 Newton's square root algorithm as a function

This second example is a lot simpler! It is merely the program for finding a square root (7.2.2) recoded to use a function:

```

#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

```

*Sanity check on
argument value*

```
double NewtRoot(double x)
{
    double r;
    const double SMALLFRACTION = 1.0E-8;
    assert(x > 0.0);

    r = x / 2.0;
    while(fabs(x - r*r) > SMALLFRACTION*x) {
        cout << r << endl;
        r = 0.5 *(x / r + r);
    }

    return r;
}

int main()
{
    double x;
    cout << "Enter number : ";
    cin >> x;

    cout << "Number was : " << x << ", root is "
        << NewtRoot(x) << endl;
    return EXIT_SUCCESS;
}
```

Once again, it is worth putting in some checking code in the function. The `assert()` makes certain that we haven't been asked to find the root of a negative number.

10.10 THE RAND() FUNCTION

The `stdlib` library includes a function called `rand()`. The documentation for `stdlib` describes `rand()` as a "simple random number generator". It is supposed to return a random number in the range 0 to 32767.

What is your idea of a "random number" generator? Probably you will think of something like one of those devices used to pick winners in lotteries. A typical device for picking lottery winners has a large drum containing balls with the numbers 1 to 99 (or maybe some smaller range); the drum is rotated to mix up the balls and then some mechanical arrangement draws six or so balls. The order in which the balls are drawn doesn't matter; you win the lottery if you had submitted a lottery coupon on which you had chosen the same numbers as were later drawn. The numbers drawn from such a device are pretty random. You would have won this weeks lottery if you had picked the numbers 11, 14, 23, 31, 35, and 39; last week you wished that you had picked 4, 7, 17, 23, 24, and 42.

The `rand()` function don't seem to work quite that way. For example, the program:

```
int main()
{
    for(int i = 0;i<20;i++)
        cout << rand() << endl;
    return 0;
}
```

run on my Macintosh produces the numbers:

```
16838
5758
10113
17515
31051
5627
23010
...
...
25137
25566
```

Exactly the same numbers are produced if I run the program on a Sun workstation (though I do get a different sequence on a PowerPC).

There is an auxiliary function that comes with `rand()`. This is `srand()`. The function `srand()` is used to "seed the random number generator". Function `srand()` takes an integer as an argument. Calling `srand()` with the argument 5 before the loop generating the "random numbers":

*"Seeding" the
random number
generator*

```
srand(5);
for(int i = 0;i<20;i++)
    cout << rand() << endl;
```

and (on a Macintosh with Symantec 7.06) you will get the sequence

```
18655
8457
10616
31877
...
...
985
32068
24418
```

Try another value for the "seed" and you will get a different sequence.

For a given seed, the same numbers turn up every time.

Arbitrary numbers? Yes. Random numbers? No. It is the same sequence of arbitrary numbers every time (and since most versions of stdlib are identical, it is the same sequence on most computers). A lottery company wouldn't be wise to try to computerize their selection process by using `rand()` like this. Smart punters would soon learn which were the winning numbers because these would be the same every week.

The term "random number generator" is misleading. The `rand` function has a more correct description; *"rand() uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to $(2^{15})-1$ ".*

What does that mathematical mumbo-jumbo mean? It means that these "pseudo-random numbers" are generated by an algorithmic (non-random) mathematical process but that as a set they share some mathematical properties with genuinely randomly picked numbers in the same range.

Some of these properties are:

- The counts for the frequency of each of the possible last digits, obtained by taking the number modulo 10 (i.e. `rand() % 10`), will be approximately equal.
- If you look at pairs of successive pseudo-random numbers, then the number times that the second is larger will be approximately equal to the number of times that the first is larger.
- You will get about the same number of even numbers as odd numbers.
- If you consider a group of three successive random numbers, there is a one in eight chance that all will be odd, a one in eight chance that all are even, a three in eight chance that you will get two odds and one even. These are the same (random) chances that you would get if you were tossing three coins and looking for three heads etc.

Although the generated numbers are not random, the statistical properties that they share with genuinely random numbers can be exploited.

If you've lost your dice and want a program to roll three dice for some game, then the following should be (almost) adequate:

```
int main()
{
    // Roll three dice for game
    char ch;
    ...
    do {
        for(int i=0; i < 3; i++) {
            int roll = rand() % 6;
            // roll now in range 0..5
            roll++; // prefer 1 to 6
            cout << roll << " ";
        }
        cout << endl;
    } while(ch != 'q');
```

```

        cout << "Roll again? ";
        cin >> ch;
    } while ((ch == 'Y') || (ch == 'y'));
    return 0;
}

```

About one time in two hundred you will get three sixes; you have the same chance of getting three twos. So the program does approximate the rolling of three fair dice.

But it is a bit boring. Every time you play your game using the program you get the same dice rolls and so you get the same sequence of moves.

```

3 5 4
Roll again? y
2 2 6
Roll again? y
1 4 1
Roll again? y
1 2 6
Roll again? n

```

You can extend the program so that each time you run it you get a different sequence of arbitrary numbers. You simply have to arrange to seed the random number generator with some "random" value that will be different for each run of the program.

Of course, normal usage requires everything to be the same every time you run the same program on the same data. So you can't have anything in your program that provides this random seed; but you can always ask the operating system (OS) about other things going on in the machine. Some of the values you can get through such queries are likely to be different on different runs of a program.

Finding a seed

Most commonly, program's ask the OS for the current time of day, or for some related time measure like the number of seconds that the machine has been left switched on. These calls are OS dependent (i.e. you will have to search your IDE documentation for details of the name of the timing function and its library). Just for example, I'll assume that your system has a function `TickCount()` that returns the number of seconds that the machine has been switched on. The value returned by this function call can be used to initialize the random number generator:

```

int main()
{
    // Roll three dice for game
    char ch;
    srand(TickCounter());
    do {
        for(int i=0; i < 3; i++) {
            int roll = rand() % 6;
            ...
        }
    }
}

```

With this extension to "randomize" the number sequence, you have a program that will generate a different sequence of dice rolls every time you run it.

If "TickCounter()" is random enough to seed the generator, why not just use it to determine the roll of the dice:

```
do {  
    for(int i=0; i < 3; i++) {  
        int roll = TickCounter() % 6;  
        roll++; // prefer 1 to 6
```

Of course this doesn't work! The three calls to `TickCounter()` are likely to get exactly the same value, if not identical then the values will be in increasing order. A single call to `TickCounter()` has pretty much random result but inherently there is no randomness in successive calls; the time value returned is monotonically increasing.

When developing and testing a program, you will often chose to pass a constant value in the call to `srand()`. That way you get the same sequence of numbers every time. This can help debugging because if you find something going wrong in your program then, after adding trace statements, you can rerun the program on identical data. When the program has been developed, you arrange either to get the user to enter the seed or you get a seed value from a system call like `TickCounter()`.

10.11 EXAMPLES

10.11.1 π -Canon

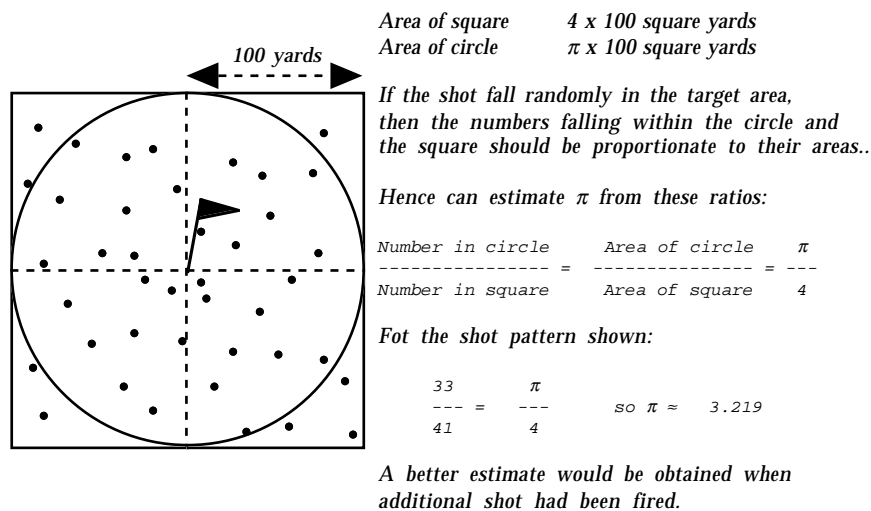
A mathematics professor once approached a colonel of his state's National Guard asking for the Guards assistance in a new approach to estimating the value of π .

The professor reckoned that if a cannon was fired at a distant target flag, the balls would fall in a random pattern around the target. After a large number of balls had been fired, he would be able to get an estimate of π by noting where the balls had fallen. He used the diagram shown in Figure 10.3, to explain his approach to estimating π :

The colonel refused. He reckoned that his gunners would put every shot within 50 paces of the target flag and that this would lead to the result $\pi = 4$ which everyone knew was wrong.

Write a computer program that will allow the maths. professor to simulate his experiment.

The program should let the professor specify the number of shots fired. It should simulate the experiment printing estimates of π at regular intervals along with a final estimate when all ammunition has been expended.

Figure 10.3 Estimating π .

Design

The design process appropriate to problems of this complexity is known as "top-down functional decomposition". We know the overall function (the "top function" of the program), it is to run the simulation and calculate π . But obviously, that involves a lot of work. It is far too much work to be encoded in a single main function. So, we proceed by iterating through the problem repeatedly, at each iteration we try to refine our model and abstract out distinct subproblems that can be analyzed separately.

The problem obviously involves random numbers. For each "shot" we will need two random numbers, one to give the "x" position relative to the flag, the second to give the "y" position. We can arrange it so that all shots fall in the square by selecting the range for the random numbers (just have to tell the professor that we are ignoring bad shots that fall too far from the target). When we have the x, y coordinates of where the shot is supposed to have landed, we can calculate the distance from the flag. If this distance is less than 100.0, the shot adds to the count of those falling in the circle. We can get an estimate of π whenever we need by just doing the calculation using our counts of total shots fired and shots landing in the circle.

Based on the problem description, we can identify a rough framework for the program:

First iteration through design

```
//initialization phase
  get professor to say how much ammunition he has
  get some seed for the random number generator
loop until ammunition exhausted
  increment count of shots fired
```

```

    get x, y coordinates
    test position and, if appropriate,
        update count of shots in circle
    maybe print next estimate of  $\pi$  (not for every shot,
        What is frequency of these reports?)
    print final estimate of  $\pi$ 

```

Some "functions" appear immediately in this description, for example, "estimate π ", "test position", "get coordinate". All of these tasks involve several calculation steps that clearly belong together, can be thought about in isolation, and should be therefore separate functions. Even something like "initialization phase" could probably be a function because it groups related operations.

**Second iteration
through design**

We should now start to identify some of these functions:

```

get ammunition
    prompts professor for inputs like amount of ammunition,
        and seed for random number generator
    use seed to seed the random number generator
    return amount of ammunition

 $\pi$ -estimator function
    must be given total shot count and circle shot count
    calculates  $\pi$  from professor's formula
    returns  $\pi$  estimate

get-coordinate function
    use random number generator, converts 0..32767 result
        into a value in range -100.0 to +100.0 inclusive
    returns calculated value

print  $\pi$ -estimate
    produce some reasonably formatted output,
    probably should show  $\pi$  and the number of shots used to
        get this estimate, so both these values will
        be needed as arguments

test in circle
    given an x and y coordinate pair test whether they
        represent a point within 100.0 of origin
    return 0 (false) 1 (true) result

main function
    get ammunition
    loop until ammunition exhausted (How checked?)
        increment count of shots fired
        (Where did this get initialized?)
    x = get-coordinate
    y = get-coordinate
    get x,y coordinates
    if test in circle (x, y)
        increment circle count

```



```

        maybe print next estimate of  $\pi$  (not for every shot,
            What is frequency of these reports?)
    call print  $\pi$  estimate

```

There are some queries. These represent issues that have to be resolved as we proceed into more detailed design.

This problem is still fairly simple so we have already reached the point where we can start thinking about coding. At this point, we compose function prototypes for each of the functions. These identify the data values that must be given to the function and the type of any result:

*Third iteration
through design
process*

```

long      GetAmmunition(void);

double    PiEstimate(long squareshot, long circleshot);

double    GetCoordinate(void);

void      PrintPIEstimate(double pi, long shotsused);

int       TestInCircle(double x, double y);

int       main();

```

*Composing the
function prototypes*

Final stage design and implementation

Next, we have to consider the design of each individual function. The process here is identical to that in Part II where we designed the implementations of the little programs. We have to expand out our english text outline for the function into a sequence of variable definitions, assignment statements, loop statements, selection statements etc.

Most of the functions that we need here are straightforward:

```

long GetAmmunition(void)
{
    long    munition;
    long    seed;
    cout << "Please Professor, how many shots should we fire?";
    // Read shots, maybe we should use that GetIntegerInRange
    // to make certain that the prof enters a sensible value
    // Risk it, hope he can type a number
    cin >> munition;
    // Need seed for random number generator,
    // Get prof to enter it (that way he has chance of
    // getting same results twice). Don't confuse him
    // with random number stuff, just get a number.
    cout << "Please Professor, which gunner should lay the
gun?"

```

```

        << endl;
        cout << "Please enter a gunner identifier number 1..1000
: ";
        cin >> seed;
        srand(seed);
        return munition;
    }

    int TestInCircle(double x, double y)
    {
        // Use the sqrt() function from math.h
        // rather than NewtRoot()!
        // Remember to #include <math.h>
        double distance = sqrt(x*x + y*y);
        return distance < 100.0;
    }

    double PiEstimate(long squareshot, long circleshot)
    {
        return 4.0*double(circleshot)/squareshot;
    }

```

The `GetCoordinate()` function is slightly trickier; we want a range -100.0 to +100.0. The following should suffice:

```

double GetCoordinate(void)
{
    int    n = rand();
    // Take n modulo 201, i.e. remainder on dividing by 201
    // That should be a number in range 0 to 200
    // deduct 100
    n = n % 201;
    return double(n-100);
}

```

The `main()` function is where we have to consider what data values are needed for the program as a whole. We appear to need the following data elements:

```

long counter for total shots
long counter for shots in circle
long counter for available munitions

double for  $\pi$  estimate

```

within the loop part of `main()` we also require:

```

double x coordinate
double y coordinate

```

We have to decide how often to generate intermediate printouts of π estimates. If professor is firing 1000 shots, he would probably like a report after every 100; if he can only afford to pay for 100 shots, he would probably like a report after every 10. So it seems we can base report frequency on total of shots to be fired. We will need a couple of variables to record this information; one to say frequency, the other to note how many shots fired since last report. So, `main()` will need two more variables.

```
long counter for report frequency
long counter for shots since last report
```

We can now complete a draft for `main()`:

```
int main()
{
    long    fired = 0, ontarget = 0, ammunition;
    double pi = 4.0; // Use the colonel's estimate!
    long    freq, count;

    ammunition = GetAmmunition();

    freq = ammunition / 10;
    count = 0;

    for(; ammunition > 0; ammunition--) {
        double x, y;
        fired++;
        x = GetCoordinate();
        y = GetCoordinate();
        if(TestInCircle(x, y))
            ontarget++;

        count++;
        if(count == freq) {
            // Time for another estimate
            pi = PiEstimate(fired, ontarget);
            PrintPIEstimate(pi, fired);
            // reset count to start over
            count = 0;
        }
    }

    cout << "We've used all the ammunition." << endl;
    cout << "Our final result:" << endl;
    pi = PiEstimate(fired, ontarget);
    PrintPIEstimate(pi, fired);
    return 0;
}
```

Complete the implementation of the program (you need to provide `PrintPIEstimate()`) and run it on your system.

The results I got suggest that the prof. would have to apply for, and win, a major research grant before he could afford sufficient ammunition to get an accurate result:

```
Estimate for pi, based on results of 1000 shots, is 3.24800000  
Estimate for pi, based on results of 10000 shots, is 3.12160000
```

10.11.2 Function plotter

Problem

When doing introductory engineering and science subjects, you often need plots of functions, functions such as

$$f(x) = a \cdot \cos(b \cdot x + c) \cdot \exp(-d \cdot x)$$

This function represents an exponentially decaying cosine wave. Such functions turn up in things like monitoring a radio signal that starts with a pulse of energy and then decays away. The parameters a , b , c , and d determine the particular characteristics of the system under study.

It is sometimes useful to be able to get rough plots that just indicate the forms of such functions. These plots can be produced using standard iostream functions to print characters. The width of the screen (or of a printer page) can be used to represent the function range; successive lines of output display the function at successive increasing values of x . Figure 10.4 illustrates such a plot.

Specification

Write a program that will plot function values using character based output following the style illustrated in Figure 10.4. The program is to plot values of a fixed function $f(x)$ (this function can only be changed by editing and recompilation). The program is to take as input a parameter R that defines the range of the function, the page (screen) width is to be used to represent the range $-R$ to $+R$. Values of the function are to be plotted for x values, starting at $x = 0.0$, up to some maximum value X . The maximum, and the increments for the x -axis, are to be entered by the user when the program runs.

Design

"Plotting a function" – the "top function" of this program is again simple and serves as the starting point for breaking the problem down.

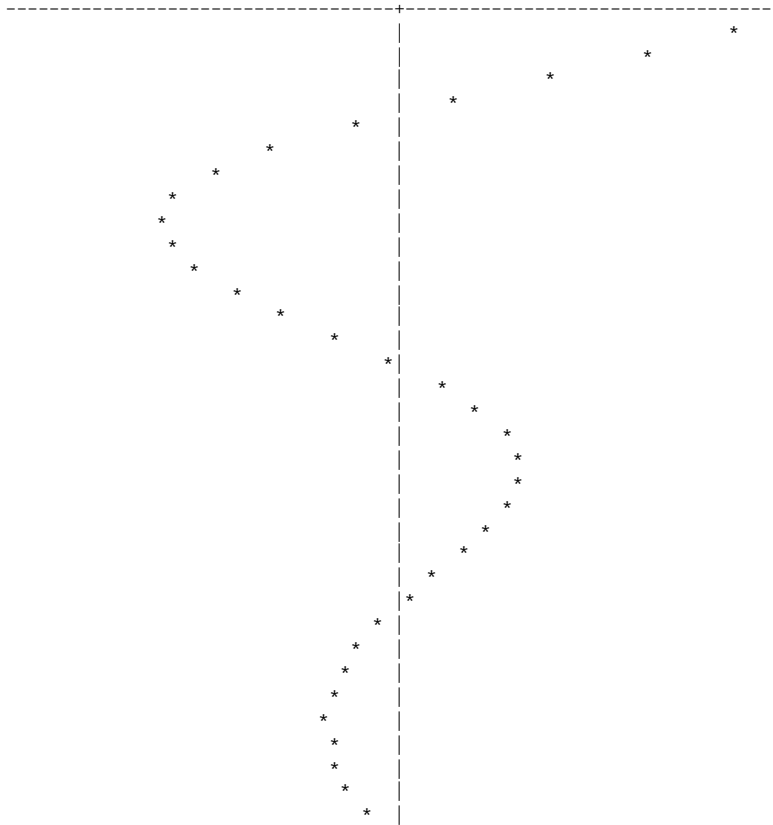


Figure 10.4 Character based plotting of a function.

A rough framework for the program would be:

*First iteration
through design*

```
//initialization phase
    range data, maximum x value, and x increment
    initialize current x to 0.0
//
    Plot axis
// main loop
loop until x exceeds maximum
    work out function value
    plot current value (and x-axis marker)
    increment x
```

The obvious functions are: "work out function value", "plot current value", and "plot axis".

**Second iteration
through design**

These functions can start to be elaborated:

```

fun
    Given current x value as data, works out value of
expression
    returns this value as a double
    (expression will probably contain parameters like a, b,
...;
    but these can be defined as constants as they aren't to be
    changed by user input)

plotaxis
    prints a horizontal rule (made from '-' characters) with a
    centre marker representing the zero point of the
    range
    probably should also print labels, -R and +R, above the
rule
    so the value for the range should be given as an
    argument
    (? width of screen ? will need this defined as a constant
    that could be adjusted later to suit different
    screen/page sizes)

plot function
    given a value for f(x), and details of the range,
    this function must first work out where on line
    a mark should appear
    a line should then be output made up from spaces, a |
    character to mark the axis, and a * in the
    appropriate position
  
```

Functions `main()`, `plotaxis()`, and `fun()` look fairly simple. You wouldn't need to iterate further through the design process for these; just have to specify the prototypes and then start on coding the function bodies.

Function prototypes

```

int main();
void PlotAxis(double range);
double fun(double x);
void PlotFunctionValue(double value, double range);
  
```

However, the description of `PlotFunctionValue()` seems a bit complex. So, it is too soon to start thinking about coding. The various steps in `PlotFunctionValue()` must be made more explicit.

**Third iteration
through design
process**

The width of the screen is meant to represent the range $-R$ to R with R the user specified range. A value v for $f(x)$ has to be scaled; the following formula defines a suitable scaling:

$$\text{position relative to centre} = v/R * \text{half_width}$$

Example, screen width 60 places, range represented -2.0 to +2.0, value to be plotted 0.8:

```
position relative to centre    = 0.8/2.0*30
                              = 12 places right of centre
```

Example, screen width 60 places, range represented -2.0 to +2.0, value to be plotted -1.5:

```
position relative to centre    = -1.5/2.0*30
                              = -22.5
                              i.e. 22.5 places left of centre
```

A C++ statement defining the actual character position on the line is:

```
int where = floor((value/range*HALF) + 0.5) + HALF;
```

The constant `HALF` represents half the screen width. Its value will depend on the environment used.

Since we now know how to find where to plot the '*' character we need only work out how to do the correct character output.

There are a number of special cases that may occur:

- The value of `where` is outside of the range `1..WIDTH` (where `WIDTH` is the screen width). This will happen if the user specified range is insufficient. If the value is out of range, we should simply print the '|' vertical bar character marking the x-axis itself.
- `where`'s value is `HALF`, i.e. the function value was 0.0. We don't print the usual '|' vertical bar axis marker, we simply plot the '*' in the middle of the page.
- `where` is less than `HALF`. We need to output the right number of spaces, print the '*', output more spaces until half way across the page, print the axis marker.
- `where` is greater than `HALF`. We need to output spaces until half way across page, print the axis marker, then output additional spaces until at point where can print the '*' symbol.

There are smarter ways to deal with the printout, but breaking it down into these possibilities is about the simplest.

The problem has now been simplified to a point where coding can start.

Implementation

There isn't much to the coding of this example and the complete code is given below.

Conditional compilation directives

The code is used to provide a first illustration of the idea of "conditional compilation". You can include in the text of your program various "directives" that instruct the compiler to take special actions, such as including or omitting a particular piece of code.

Conditional compilation for debugging and customisation

There are two common reasons for having conditionally compiled code. The first is that you want extra "tracing" code in your program while you are developing and debugging it. In this case, you will have conditionally compiled sections that have extra output statements ("trace" output); examples will be shown later. This code provides a rather simple instance of the second situation where conditional compilation is required; we have a program that we want run on different platforms – Symantec, Borland-compiled DOS application, and Borland-compiled EasyWin application. These different environments require slightly different code; but rather than have three versions of the code, we can have a single file with conditionally compiled customizations.

The differences are very small. The default font in an EasyWin window is rather large so, when measured in terms of characters, the window is smaller. If you compile and run the program as a DOS-standard program it runs, and exits back to the Borland IDE environment so quickly that you can't see the graph it has drawn! The DOS version needs an extra statement telling it to slow down.

The directives for conditional compilation, like most other compiler directives, occur on lines starting with a # character. (The only other directives we've used so far has been the #include directive that loads a header file, and the #define directive that defines a constant.) Here we use the #if, #elif ("else if"), #else, and #endif directives and use #define in a slightly different way. These directives are highlighted in bold in the code shown below; the italicised statements are either included or omitted according to the directives.

```
#include <math.h>
#include <iostream.h>
#include <stdlib.h>

#define SYMANTEC

#if defined(SYMANTEC)
const int WIDTH = 71;
#elif defined(EASYWIN)
const int WIDTH = 61;
#else
const int WIDTH = 81;
#endif

#if defined(DOS)
#include <dos.h>
#endif
```


The line `#define SYMANTEC` "defines" SYMANTEC as a compile time symbol (it doesn't get a value, it just exists); this line is specifying that this version of the code is to be compiled for the Symantec environment. The line would be changed to `#define EASYWIN` or `#define DOS` if we were compiling for one of the other platforms.

The `#if defined()` ... `#elif defined()` ... `#else` ... `#endif` statements select the WIDTH value to be used depending on the environment. The other `#if` ... `#endif` adds an extra header file if we are compiling for DOS. The DOS version uses the standard `sleep()` function to delay return from the program. Function `sleep()` is defined in most environments; it takes an integer argument specifying a delay period for a program. Although it is widely available, it is declared in different header files and is included in different libraries on different systems.

```
const int HALF = 1 + (WIDTH/2);

const double a = 1.0;
const double b = 1.0;
const double c = 0.5;
const double d = 0.2;
```

The definition of constant HALF illustrates compile time arithmetic; the compiler will have a value for WIDTH and so can work out the value for HALF. The other constants define the parameters that affect the function that is being plotted.

```
void PlotAxis(double range)
{
    cout << -range;
    for(int i=0;i<WIDTH-10;i++)
        cout << ' ';
    cout << range << endl;
    for(i=1;i<HALF;i++)
        cout << '-';
    cout << '+';
    for(i=HALF+1;i<=WIDTH;i++)
        cout << '-';
    cout << endl;
}
```

The code for `PlotAxis()` is simple; it prints the range values, using the loop printing spaces to position the second output near the right of the window. Note that variable `i` is defined in the first for statement; as previously explained, its scope extends to the end of the block where it is defined. So, the other two loops can also use `i` as their control variables. (Each of the cases is coded as a separate compound statement enclosing a for loop. Each for loop defines `int i`. Each of these `i`'s has scope that is limited to just its own compound statement.)

```
void PlotFunctionValue(double value, double range)
{
```

*Determine position
on line
Special case 1, out of
range*

```
int where = floor(value/range*HALF) + HALF;

if((where < 1) || (where > WIDTH)) {
    for(int i=1; i<HALF;i++)
        cout<< ' ';
    cout << '|' << endl;
    return;
}
```

*Special case 2, on
axis*

```
if(where == HALF) {
    for(int i=1; i<HALF;i++)
        cout<< ' ';
    cout << '*' << endl;
    return;
}
```

*Special case 3, left of
axis*

```
if(where < HALF) {
    for(int i=1; i<where;i++)
        cout<< ' ';
    cout << '*';
    for(i=where+1;i<HALF;i++)
        cout << ' ';
    cout << '|' << endl;
    return;
}
```

*Last case, right of
axis*

```
for(int i=1; i<HALF;i++)
    cout<< ' ';
cout << '|';
for(i=HALF+1;i<where;i++)
    cout << ' ';
cout << '*';
cout << endl;

return;
}
```

Your instructor may dislike the coding style used in this version of `PlotFunctionValue()`; multiple returns are often considered poor style. The usual recommended style would be as follows:

```
void PlotFunctionValue(double value, double range)
{
    int where = floor(value/range*HALF) + HALF;

    if((where < 1) || (where > WIDTH)) {
        for(int i=0; i<HALF;i++)
            cout<< ' ';
        cout << '|' << endl;
    }
    else
```

```

    if(where == HALF) {
        ...
    }
    else
    if(where < HALF) {
        ...
    }
    else {
        ...
    }
    return;
}

```

Personally, I find long sequences of if ... else if ... else if ... else constructs less easy to follow. When reading these you have to remember too much. I prefer the first style which regards each case as a filter that deals with some subset of the data and finishes. You need to be familiar with both styles; use the coding style that your instructor prefers.

```

double fun(double x)
{
    double val;
    val = a*cos(b*x+c)*exp(-d*x);
    return val;
}

int main()
{
    double r;
    double xinc;
    double xmax;

    cout << "Range on 'y-axis' is to be -R to R" << endl;
    cout << "Please enter value for R : ";
    cin >> r;

    cout << "Range on 'x-axis' is 0 to X" << endl;
    cout << "Please enter value for X : ";
    cin >> xmax;

    cout << "Please enter increment for X-axis : ";
    cin >> xinc;

    PlotAxis(r);

    double x = 0.0;
    while(x<=xmax) {
        double f = fun(x);
        PlotFunctionValue(f, r);
        x += xinc;
    }
}

```

```

    }

    #if defined(DOS)
        // delay return from DOS to Borland IDE for
        // a minute so user can admire plotted function
        sleep(60);
    #endif

    return 0;

}

```

There are no checks on input values, no assertions. Maybe there should be. But there isn't that much that can go seriously wrong here. The only problem case would be the user specifying a range of zero; that would cause an error when the function value is scaled. You can't put any limit on the range value; it doesn't matter if the range is given as negative (that flips the horizontal axis but does no real harm). A negative limit for `xmax` does no harm, nothing gets plotted but no harm done.

EXERCISES

- 1 Write a program that will roll the dice for a "Dungeons and Dragons" (DD) game.

DD games typically use several dice for each roll, anything from 1 to 5. There are many different DD dice 1...4, 1...6, 1...8, 1...10, 1...12, 1...20. All the dice used for a particular roll will be of the same type. So, on one roll you may need three six sided dice; on the next you might need one four sided dice.

The program is to ask how many dice are to be rolled and the dice type. It should print details of the points on the individual dice as well as a total score.

- 2 Write a "Chinese fortune cookie" program.

The program is to loop printing a cookie message (a joke, a proverb, cliché, adage). After each message, the program is to ask the user for a Y(es) or N(o) response that will determine whether an additional cookie is required.

Use functions to select and print a random cookie message, and to get the Y/N response.

You can start with the standard fortunes – "A new love will come into your life.", "Stick with your wife.", "Too much MSG, stomach pains tomorrow.", "This cookie was poisoned.". If you need others, buy a packet of cookies and plagiarise their ideas.

11 Arrays

Suppose that, in the example on children's heights (example in 8.5.1), we had wanted to get details of all children whose heights were more than two standard deviations from the mean. (These would be the tallest 5% and the smallest 5% of the children).

We would have had problems. We couldn't have identified these children as we read the original data. All the data have to be read before we can calculate the standard deviation and the mean. We could use a clumsy mechanism where we read the data from file once and do the first calculations, then we "rewind the file" and read the data values a second time, this time checking each entry against the mean. (This scheme sort of works for files, but "rewinding" a human user and requesting the rekeying of all interactive input is a less practical proposition.)

Really, we have to store the data as they are read so we can make a second pass through the stored data picking out those elements of interest.

But how can these data be stored?

A scheme like the following is obviously impractical:

```
int main()
{
    double average, stand_dev;
    double height1, height2, height3, height4, ...,
           height197, height198, ...,
           height499, height500;
    cout << "Enter heights for children" << endl;
    cin >> height1 >> height2 >> height3 >> ... >> height500;
```

Quit apart from the inconvenience, it is illogical. The program would have had to be written specifically for the data set used because you would need the same number of variables defined, and input statements executed, as you have values in your data set.

Collections of data elements of the same type, like this collection of different heights, are very commonly needed. They were needed back in the earliest days of programming when all coding was done in assembly language. The programmers of

*The beginnings in
assembly language
programs*

the early 1950s got their assemblers to reserve blocks of memory locations for such data collections; an "assembler directive" such as

```
HEIGHTS      ZBLOCK 500
```

would have reserved 500 memory locations for the collection of heights data. The programmers would then have used individual elements in this collection by writing code like:

```
load address register with the address where heights data start
add on "index" identifying the element required
// now have address of data element required
load data register from address given in address register
```

By changing the value of the 'index', this code accesses different elements in the collection. So, such code inside loops can access each element in turn.

*High level languages
and "arrays"*

Such collections were so common that when FORTRAN was introduced in 1956 it made special provision for "arrays". Arrays were FORTRAN's higher level abstraction of the assembler programmers' block of memory. A FORTRAN programmer could have defined an array using a statement like:

```
DIMENSION HEIGHTS(500)
```

and then used individual array elements in statements like:

```
HEIGHTS(N) = H;
...
IF(HEIGHTS(I) .LT. HMAX) GOTO 50
...
```

The definition specifies that there are 500 elements in this collection of data (numbered 1 to 500). The expressions like `HEIGHTS(N)` and `HEIGHTS(I)` access chosen individual elements from the collection. The integers `N` and `I` are said to "index" into the array. The code generated by the compiler would have been similar to the fragment of assembler code shown above. (The character set, as available on the card punches and line printers of the 1950s and 1960s, was limited, so FORTRAN used left (and right) parentheses for many purposes including array indexing; similarly a text string such as `.LT.` would have had to be used because the `<` character would not have been available.)

*Arrays with bounds
checks and chosen
subscript ranges*

As languages evolved, improved versions of the array abstraction were generally adopted. Thus, in Pascal and similar 1970s languages, arrays could be defined to have chosen subscript ranges:

```
heights : array [1..500] of real;
...
```

```
ch_table : array['a'..'z'] of char;
```

Individual array elements could be accessed as before by subscripting:

```
heights[n] := h;  
...  
if(heights[i] > hmax) hmax := heights[i];
```

The compilers for these languages generally produced more complex code that included checks based on the details of the subscript range as given in the definition. The assembly language generated when accessing array elements would typically include extra instructions to check that the index was in the appropriate range (so making certain that the requested data element was one that really existed). Where arrays were to be used as arguments of functions, the function declaration and array definition had to agree on the size of the array. Such checks improve reliability of code but introduce run-time overheads and restrict the flexibility of routines.

When C was invented, its array abstraction reverted more to the original assembly language programmer's concept of an array as a block of memory. This was partly a consequence of the first version of C being implemented on a computer with a somewhat primitive architecture that had very limited hardware support for "indexing". But the choice was also motivated by the applications intended for the language. C was intended to be used instead of assembly language by programmers writing code for components of an operating system, for compiler, for editors, and so forth. The final machine code had to be as efficient as that obtained from hand-coded assembly language, so overheads like "array bounds checking" could not be countenanced. *C's arrays*

Many C programmers hardly use the array abstraction. Rather than array subscripting, they use "pointers" and "pointer arithmetic" (see Part IV) to access elements. Such pointer style code is often closely related to the optimal set of machine instructions for working through a block of memory. However, it does tend to lose the abstraction of an array as a collection of similar data elements that can be accessed individual by specifying an index, and does result in more impenetrable, less readable code.

The C++ language was intended to be an improvement on C, but it had to maintain substantial "backwards compatibility". While it would have been possible to introduce an improved array abstraction into C++, this would have made the new language incompatible with the hundreds of thousands of C programs that had already been written. So a C++ array is just the same as a C array. An array is really just a block of memory locations that the programmer can access using subscripting or using pointers; either way, access is unchecked. There are no compile time, and no run time checks added to verify correct usage of arrays; it is left to the programmer to get array code correct. Inevitably, array usage tends to be an area where many bugs occur. *C++'s arrays*

11.1 DEFINING ONE DIMENSIONAL ARRAYS

Simple C++ variables are defined by giving the type and the name of the variable. Arrays get defined by qualifying the variable name with square brackets [and], e.g.:

```
double    heights[500];
```

"Vector" or "table" This definition makes `heights` an array with 500 elements. The terms *vector* and *table* are often used as an alternative names for a singly subscripted array like `heights`.

Array index starts at zero! The low-level "block of memory" model behind C's arrays is reflected in the convention that array elements start with 0. The definition for the `heights[]` array means that we can access `heights[0] ... heights[499]`. This is natural if you are thinking in terms of the memory representation. The first data element is where the `heights` array begins so to access it you should not add anything to the array's address (or, if you prefer, you add 0).

This "zero-base" for the index is another cause of problems. Firstly, it leads to errors when adapting code from examples written in other languages. In FORTRAN arrays start at 1. In Pascal, the programmer has the choice but zero-based Pascal arrays are rare; most Pascal programs will use 1-based arrays like FORTRAN. All the loops in FORTRAN and Pascal programs tend to run from 1..N and the array elements accessed go from 1 .. N. It is quite common to find that you have to adapt some FORTRAN or Pascal code. After all, there are very large collections of working FORTRAN routines in various engineering and scientific libraries, and there are numerous text books that present Pascal implementations of standard algorithms. When adapting code you have to be careful to adjust for the different styles of array usage. (Some people chicken out; they declare their C arrays to have N+1 elements, e.g. `double x[N+1]`, and leave the zeroth element, `x[0]`, unused. This is OK if you have one dimensional arrays of simple built in types like doubles; but you shouldn't take this way out for arrays of structures or for multidimensional arrays.)

A second common problem seems just to be a consequence of a natural, but incorrect way of thinking. You have just defined an array, e.g. `xvals[NELEMENTS]`:

```
const int  NELEMENTS    = 100;
double     xmax;
double     xvals[NELEMENTS];
double     ymax;
...
```

So, it seems natural to write something like:

```
xvals[NELEMENTS] = v;    // WRONG, WRONG, WRONG
```

Of course it is wrong, there is no data element `xvals[NELEMENTS]`. However, this code is acceptable to the compiler; the instructions generated for this assignment will be

executed at run-time without any complaint. The value actually changed by this assignment will be (probably) either `xmax`, or `ymax` (it depends on how your compiler and, possibly, link-loader choose to organize memory). A program with such an error will appear to run but the results are going to be incorrect.

In general, *be careful with array subscripts!*

You should use named constants, or `#defined` values, when defining the sizes of arrays rather than writing in the number of array elements. So, the definitions:

Constants in array definitions

```
const int  MAXCHILDREN = 500;
...
double    heights[MAXCHILDREN];
```

are better than the definition given at the start of this section. They are better because they make the code clearer (`MAXCHILDREN` has some meaning in the context of the program, 500 is just a "magic number"). Further, use of a constant defined at the start of the code makes it easier to change the program to deal with larger numbers of data values (if you have magic numbers, you have to search for their use in array definitions and loops).

Technically, the `[` and `]` brackets together form a `[]` "postfix operator" that modifies a variable in a definition or in an expression. (It is a "postfix operator" because it comes after, "post", the thing it fixes!) In a definition (or declaration), the `[]` operator changes the preceding variable from being a simple variable into an array. In an expression, the `[]` operator changes the variable reference from meaning the array as a whole to meaning a chosen element of the array. (Usually, you can only access a single array element at a time, the element being chosen using indexing. But, there are a few places where we can use an array as a whole thing rather than just using a selected element; these will be illustrated shortly.)

The [] operator

Definitions of variables of the same basic type can include both array definitions and simple variable definitions:

```
const int  MAXPOINTS = 50;
...
double     xcoord[MAXPOINTS], xmax, ycoord[MAXPOINTS], ymax,
           zcoord[MAXPOINTS], zmax;
```

This defines three simple data variables (`xmax`, `ymax`, and `zmax`), and three arrays (`xcoord`, `ycoord`, and `zcoord`) each with 50 elements.

Arrays can be defined as local, automatic variables that belong to functions:

Local and global arrays

```
int main()
{
    double heights[MAXCHILDREN];
    ...
}
```

Local (automatic)

But often you will have a group of functions that need to share access to the same array data. In such situations, it may be more convenient to define these shared arrays as "global" data:

```
Global    double    xcoord[MAXPOINTS], ycoord[MAXPOINTS],
              zcoord[MAXPOINTS];

void ScalePoints(double scaling)
{
    // change all points entries by multiplying by scaling
    for(int i=0;i<MAXPOINTS;i++) {
        xcoord[i] *= scaling;
        ycoord[i] *= scaling;
        ...
    }

void RotatePointsOnX(double angle)
{
    // change all points to accommodate rotation about X axis
    // by angle
    ...
```

Variables declared outside of any function are "globals". They can be used in all the other functions in that file. In fact, in some circumstances, such variables may be used by functions in other files.

static qualifier You can arrange things so that such arrays are accessible by the functions in one file while keeping them from being accessed by functions in other files. This "hiding" of arrays (and simple variables) is achieved using the `static` qualifier:

```
"file scope"    static double xccord[MAXPOINTS];
                  static double yccord[MAXPOINTS];
                  static double zccord[MAXPOINTS];

void ScalePoints(double scaling)
{
    ...
```

Here, `static` really means "file scope" – restrict access to the functions defined in this file. This is an unfortunate use of the word `static`, which C and C++ also use to mean other things in other contexts. It would have been nicer if there were a separate specialized "filescope" keyword, but we must make do with the way things have been defined.

Caution on using "globals" and "file scope" variables Some of you will have instructors who are strongly opposed to your use of global data, and even of file scope data. They have reasons. Global data are often a source of problems in large programs. Because such variables can be accessed from almost anywhere in the program there is no control on their use. Since these data values can be changed by any function, it is fairly common for errors to occur when programmers

writing different functions make different assumptions about the usage of these variables.

However for small programs, file scope data are safe, and even global data are acceptable. (Small programs are anything that a single programmer, scientist, or engineer can create in a few weeks.) Many of the simple programs illustrated in the next few chapters will make some use of file scope and/or global data. Different ways of structuring large programs, introduced in Parts IV and V, can largely remove the need for global variables.

11.2 INITIALIZING ONE DIMENSIONAL ARRAYS

Arrays are used mainly for data that are entered at run time or are computed. But sometimes, you need arrays that are initialized with specific data. Individual data elements could be initialized using the = operator and a given value:

```
long      count = 0;
long      min = LONG_MAX;
long      max = LONG_MIN;
```

Of course, if we are to initialize an array, we will normally need an initial value for each element of the array and we will need some way of showing that these initializing values form a group. The grouping is done using the same { begin and } end brackets as are used to group statements into compound statements. So, we can have initialized arrays like:

```
short grade_points[6] = { 44, 49, 64, 74, 84, 100 };
char grade_letters[6] = { 'F', 'E', 'D', 'C', 'B', 'A' };
```

As shown in these examples, an array definition-initialization has the form:

```
type array_name [ number_elements ] = {
    initial_val , initial_val , ...,
    initial_val
};
```

The various initializing values are separated by commas. The } end bracket is followed by a semicolon. This is different from the usage with compound statements and function definitions; there you don't have a semicolon. Inevitably, the difference is a common source of minor compilation errors because people forget the semicolon after the } in an array initialization or wrongly put one in after a function definition.

Generally, if an array has initial values assigned to its elements then these are going to be constants. Certainly, the marks and letter grades aren't likely to be changed when the program is run. Usually, initialized arrays are `const`:

```
const short grade_points[6] = { 44, 49, 64, 74, 84, 100 };  
const char grade_letters[6] = { 'F', 'E', 'D', 'C', 'B', 'A' };
```

You don't have to provide initial values for every element of an array. You could have a definition and initialization like the following:

```
double      things[10] = { 17.5, 19.2, 27.1 };
```

Here, `things[0]`, `things[1]`, and `things[2]` have been given explicit initial values; the remaining seven `things` would be initialized to zero or left uninitialized. (Generally, you can only rely on zero initialization for global and file scope variables.) There aren't any good reasons why you should initialize only part of an array, but it is legal. Of course, it is an error to define an array with some specified number of elements and then try to use more initial values than will fit!

Naturally, there are short cuts. If you don't feel like counting the number of array elements, you don't have to. An array can be defined with an initialization list but no size in the `[]`; for example:

```
int mystery[] = { -3, 101, 27, 11 };
```

The compiler can count four initialization values and so concludes that the array is `mystery[4]`.

Seems crazy? It is quite useful when defining arrays containing entries that are character strings, for example a table of error messages. Examples are given in 11.7.

11.3 SIMPLE EXAMPLES USING ONE-DIMENSIONAL ARRAYS

11.3.1 Histogram

Lets get back to statistics on childrens' heights. This time we aren't interested in gender specific data but we want to produce a simple histogram showing the distribution of heights and we want to have listed details of the tallest and smallest 5% of children in the sample.

Specification:

1. The program is to read height and gender values from the file `heights.dat`. The height values should be in the range 100 ... 200, the gender tags should be `m` or `f`. The gender tag information should be discarded. The data set will include records for at least ten children; averages and standard deviations will be defined (i.e. no need to check for zero counts).

2. The input is to be terminated by a sentinel data record with a height 0 and an arbitrary m or f gender value.
3. When the terminating sentinel data record is read, the program should print the following details: total number of children, average height, standard deviation in height.
4. Next the program is to produce a histogram showing the numbers of children in each of the height ranges 100...109 cm, 110...119 cm, ..., 190...199 cm. The form of the histogram display should be as follows:

```

100  |
110  | *
120  | ***
130  | *****
...

```

5. Finally, the program is to list all heights more than two standard deviations from the mean.

Program design

The program breaks down into the following phases:

Preliminary design

```

get the data
calculate mean and standard deviation
print mean and standard deviation
produce histogram
list extreme heights

```

Each of these is a candidate for being a separate function (or group of functions).

The main data for the program will consist of an array used to store the heights of the children, and a count that will define the number of children with records in the data file. The array with the heights, and the count, will be regarded as data that are shared by all the functions. This will minimize the amount of data that need to be passed between functions.

Decision in favour of "shared" data

The suggested functions must now be elaborated:

First iteration through design

```

get_data
    initialization - zeroing of some variables, opening file
    loop reading data, heights saved, gender-tags discarded

mean
    use shared count and heights array,
    loop summing values
    calculate average

```

```

std_dev
    needs mean, uses shared count and heights array
    calculate sum of squares etc and calculate
        standard deviation

histogram
    use shared count and heights array, also need
        own array for the histogram
    zero out own array
    loop through entries in main array identifying
        appropriate histogram entry to be incremented
    loop printing histogram entries

extreme_vals
    needs to be given average and standard deviation
    uses shared count and heights array

    loop checking each height for extreme value

main
    get_data
    m = mean(), print m (+ labels and formatting)
    sd = std_dev(), print sd
    histogram
    extreme_vals(m,sd)

```

*Second iteration,
further
decomposition into
simpler functions*

The "get_data" and "histogram" functions both appear to be moderately elaborate and so they become candidates for further analysis. Can these functions be simplified by breaking them down into subfunctions?

Actually, we will leave the histogram function (though it probably should be further decomposed). Function getdata really involves two separate activities – organizing file input, and then the loop actually reading the data. So we can further decompose into:

```

open_file
    try to open named file,
        if fail terminate program

read_data
    read first data (height and gender tag)
    while not sentinel value
        save height data
        read next data

get_data
    initialize count to zero
    call open_file
    call read_data

```

The ifstream used for input from file will have to be another shared data element because it is used in several functions.

The final stage before coding involves definition of i) constants defining sizes for arrays, ii) shared variables, and iii) the function prototypes.

*Third iteration,
finalize shared data,
derive function
prototypes*

```
const int MAXCHILDREN = 500;

static double s_heights[MAXCHILDREN];
static short s_count;

static ifstream s_input;

void open_file(void);

void read_data(void);

void get_data(void);

double mean(void);

double std_dev(double mean);

void histogram(void);

void extreme_vals(double mean, double sdev);

int main();
```

The shared data have been defined as `static`. As this program only consists of one source file, this "file scoping" of these variables is somewhat redundant. It has been done on principle. It is a way of re-emphasising that "these variables are shared by the functions defined in this file". The names have also been selected to reflect their role; it is wise to have a naming scheme with conventions like 's_' at the start of the name of a static, 'g_' at the start of the name for a global.

Implementation

The file organization will be:

```
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <assert.h>

const int MAXCHILDREN = 500;

static double s_heights[MAXCHILDREN];
static short s_count;
```

#included headers

*Definitions of
constants and static
(shared) variables*

Function definitions

```
static ifstream s_input;

void open_file(void)
{
    ...
}

void read_data(void)
{
    ...
}

...

main()
int main()
{
    ...
    return 0;
}
```

open_file() The `open_file()` routine would specify the options `ios::in | ios::nocreate`, the standard options for a data input file, in the `open()` call. The `ifstream` should then be checked to verify that it did open; use `exit()` to terminate if things went wrong.

```
void open_file(void)
{
    s_input.open("heights.dat",ios::in | ios::nocreate);
    if(!s_input.good()) {
        cout << "Couldn't open the file. Giving up."
              << endl;
        exit(1);
    }
    return;
}
```

read_data() The `read_data()` routine uses a standard while loop to read and process input data marked by a terminal sentinel record, (i.e. *read first record; while not sentinel record { process record, read next record }*). The data get copied into the array in the statement `s_heights[s_count] = h;`.

```
void read_data(void)
{
    double h;
    char   gender_tag;

    s_input >> h >> gender_tag;
    while(h > 0.0) {
        if(s_count < MAXCHILDREN)
            s_heights[s_count] = h;
        else
```

Store data element in array


```

        if(s_count == MAXCHILDREN)
            cout << "Ignoring some "
                "data records" << endl;
        s_count++;
        s_input >> h >> gender_tag;
    }
    if(s_count>MAXCHILDREN) {
        cout << "The input file contained " << s_count
            << " records." << endl;
        cout << "Only the first " << MAXCHILDREN <<
            " will be processed." << endl;
        s_count = MAXCHILDREN;
    }
    s_input.close();
}

```

*Report if data
discarded*

Note the defensive programming needed in `read_data()`. The routine must deal with the case where there are more entries in the file than can be stored in the array. It can't simply assume that the array is large enough and just store the next input in the next memory location because that would result in other data variables being overwritten. The approach used here is generally suitable. When the array has just become full, a warning gets printed. When the loop terminates a more detailed explanation is given about data elements that had to be discarded.

Because it can rely on auxiliary functions to do most of its work, function `get_data()` *get_data()* itself is nicely simple:

```

void get_data(void)
{
    s_count = 0;
    open_file();
    read_data();
}

```

The `mean()` and `std_dev()` functions both involve simple for loops that accumulate the sum of the data elements from the array (or the sum of the squares of these elements). Note the for loop controls. The index is initialized to zero. The loop terminates when the index equals the number of array elements. This is required because of C's (and therefore C++'s) use of zero based arrays. If we have 5 elements to process, they will be in array elements [0] ... [4]. The loop must terminate before we try to access a sixth element taken from [5]. *mean() and std_dev()*

```

double mean(void)
{
    double sum = 0.0;
    for(int i=0; i<s_count; i++)
        sum += s_heights[i];
    return sum / s_count;
}

```

```
double std_dev(double mean)
{
    double sumSq = 0.0;
    for(int i=0; i<s_count; i++)
        sumSq += s_heights[i]*s_heights[i];
    return sqrt((sumSq - s_count*mean*mean)/(s_count-1));
}
```

histogram() Function `histogram()` is a little more elaborate. First, it has to have a local array. This array is used to accumulate counts of the number of children in each 10cm height range.

Assumptions about the particular problem pervade the routine. The array has ten elements. The scaling of heights into array indices utilizes the fact that the height values should be in the range $100 < \text{height} < 200$. The final output section is set to generate the labels 100, 110, 120, etc. Although quite adequate for immediate purposes, the overall design of the routine is poor; usually, you would try to create something a little more general purpose.

Note that the local array should be zeroed out. The main array of heights was not initialized to zero because the program stored values taken from the input file and any previous data were just overwritten. But here we want to keep incrementing the counts in the various elements of the `histo[]` array, so these counts had better start at zero.

The formula that converts a height into an index number is:

```
int index = floor((s_heights[i] - 100.0)/10.0);
```

Say the height was 172.5, this should go in `histo[7]` (range 170 ... 179); $(172.5 - 100)/10$ gives 7.25, the `floor()` function converts this to 7.

The code uses `assert()` to trap any data values that are out of range. The specification said that the heights would be in the range 100 ... 200 but didn't promise that they were all correct. Values like 86 or 200 would result in illegal references to `histo[-2]` or to `histo[10]`. If the index is valid, the appropriate counter can be incremented (`histo[index] ++` ; - this applies the `++`, increment operator, to the contents of `histo[index]`).

Nested for loops

Note the "nested" for loops used to display the contents of the `histo[]` array. The outer loop, with index `i`, runs through the array elements; printing a label and terminating each line. The inner loop, with the `j` index, prints the appropriate number of stars on the line, one star for each child in this height range. (If you had larger data sets, you might have to change the code to print one star for every five children.)

```
void histogram(void)
{
    int    histo[10];
    for(int i = 0; i < 10; i++)
        histo[i] = 0;
```

*Initialize elements of
local array to zero*

```

    for(i = 0; i < s_count; i++) {
        int index = floor((s_heights[i] - 100.0)/10.0);
        assert((index >= 0) && (index < 10));
        histo[index]++;
    }
    cout << "Histogram of heights : " << endl;
    for(i=0;i<10; i++) {
        cout << (10*(i+10)) << " |";
        for(int j = 0; j<histo[i]; j++)
            cout << '*';
        cout << endl;
    }
}

```

Convert value to index and verify

Nested for loops

The function that identifies examples with extreme values of height simply involves a loop that works through successive entries in the `heights[]` array. The absolute value of the difference between a height and the mean is calculated and if this value exceeds twice the standard deviation details are printed.

```

void extreme_vals(double mean, double sdev)
{
    double two_std_devs = sdev*2.0;
    cout << "Heights more than two standard deviations from"
          << " mean: " << endl;
    int n = 0;
    for(int i=0;i<s_count;i++) {
        double diff = s_heights[i] - mean;
        diff = fabs(diff);
        if(diff > two_std_devs) {
            cout << i << " : " << s_heights[i] << endl;
            n++;
        }
    }
    if(n==0)
        cout << "There weren't any!" << endl;
}

```

Note the introduction of a local variable set to twice the standard deviation. This avoids the need to calculate `sdev*2.0` inside the loop. Such a definition is often not needed. If `two_std_devs` was not defined, the test inside the loop would have been `if(diff > sdev*2.0) ...`. Most good compilers would have noticed that `sdev*2.0` could be evaluated just once before the loop; the compiler would have, in effect, invented its own variable equivalent to the `two_std_devs` explicitly defined here.

Because the real work has been broken down amongst many functions, the `main()` function is relatively simple. It consists largely of a sequence of function calls with just a little code to produce some of the required outputs.

```

int main()
{
    get_data();
    double avg = mean();
    double sd = std_dev(avg);
    cout << "Number of records processed : " << s_count <<
endl;
    cout << "Mean (average) : " << avg << endl;
    cout << "Standard deviation : " << sd << endl;
    histogram();
    extreme_vals(avg, sd);
    return 0;
}

```

11.3.2 Plotting once more

The plotting program, presented in section 10.11.2, can be simplified through the use of an array.

We can use an array to hold the characters that must be printed on a line. Most of the characters will be spaces, but the mid-point character will be the '|' axis marker and one other character will normally be a '*' representing the function value.

The changes would be in the function PlotFunctionValue(). The implementation given in section 10.11.2 had to consider a variety of special cases – the '*' on the axis, the '*' left of the axis, the '*' to the right of the axis. All of these special cases can be eliminated. Instead the code will fill an array with spaces, place the '|' in the central element of the array, then place the '*' in the correct element of the array; finally, the contents of the array are printed. (Note the -1s when setting array elements to compensate for the zero-based array indexing.)

*Determine element to
be set
Define array and fill
with blanks*

Place centre marker

*Place value mark
provided it is in range*

*Print contents of
character array*

```

void PlotFunctionValue(double value, double range)
{
    int where = floor(value/range*HALF) + HALF;

    char line_image[WIDTH];
    for(int i=0;i<WIDTH;i++)
        line_image[i] = ' ';

    line_image[HALF-1] = '|';

    if((where>0) && (where < WIDTH))
        line_image[where-1] = '*';

    for(i=0;i<WIDTH;i++)
        cout << line_image[i];

    cout << endl;
}

```

```

    return;
}

```

This version of `PlotFunctionValue()` is significantly simpler than the original. The simplicity is a consequence of the use of more sophisticated programming constructs, in this case an array data structure. Each new language feature may seem complex when first encountered, but each will have the same effect: more powerful programming constructs allow simplification of coding problems.

11.4 ARRAYS AS ARGUMENTS TO FUNCTIONS

The functions illustrated so far have only had simple variables as arguments and results. What about arrays, can arrays be used with functions?

Of course, you often need array data that are manipulated in different functions. The histogram example, 11.3.1, had a shared filescope (static) array for the heights and this was used by all the functions. But use of filescope, or global, data is only feasible in certain circumstances. The example in 11.3.1 is typical of where such shared data is feasible. But look at the restrictions. There is only one array of heights. All the functions are specifically designed to work on this one array.

*Limitations of shared
global and filescope
data*

But, generally, you don't have such restrictions. Suppose we had wanted histograms for all children, and also for boys and girls separately? The approach used in 11.3.1 would have necessitated three different variations on `void histogram(void)`; there would be `c_histogram()` that worked with an array that contained details of the heights of all children, `b_histogram()` – a version that used a different filescope array containing the heights for boys, and `g_histogram()` – the corresponding function for processing the heights of girls. Obviously, this would be unacceptably clumsy.

Usually, you will have many arrays. You will need functions that must be generalized so that they can handle different sets of data, different arrays. There has to be a general purpose `histogram()` function that, as one of its arguments, is told which array contains the data that are to be displayed as a histogram. So, there needs to be some way of "passing arrays to functions".

But arrays do present problems. A C (C++) array really is just a block of memory. When you define an array you specify its size (explicitly as in `x_coord[100]` or implicitly as in `grades[] = { 'f', 'e', 'd', 'c', 'b', 'a' };`), but this size information is not "packaged" with the data. Arrays don't come with any associated data specifying how many elements they have. This makes it more difficult for a compiler to generate instructions for functions that need argument arrays passed by value (i.e. having a copy of the array put onto the stack) or that want to return an array. A compiler might not be able to determine the size of the array and so could not work out the amount of space to be reserved on the stack, nor the number of bytes that were to be copied.

*Reasons for
restrictions on arrays
as arguments to
functions*

No arrays as value arguments or as results

The authors of the first C compilers decided on a simple solution. Arrays can't be passed by value. Functions can not return arrays as results. The language is slightly restricted, but the problems of the compiler writer have gone. What is more, a prohibition on the passing of arrays as value arguments or results preemptively eliminated some potentially inefficient coding styles. The copying operations involved in passing array arguments by value would have been expensive. The doubling of the storage requirement, original plus copy on the stack, could also have been a problem.

This decision restricting the use of arrays with functions still stands. Which makes C, and therefore C++, slightly inconsistent. Simple variables, struct types (Chapter 16), and class instances (Chapter 19) can all be passed by value, and results of these type can be returned by functions. Arrays are an exception.

Arrays are always passed by reference

When an array is passed as an argument to a function, the calling routine simply passes the address of the array; i.e. the caller tells the called function where to find the array in memory. The called function then uses the original array. This mechanism, where the caller tells the called function the address of the data, is called *pass by reference*.

As well as passing the array, the calling function (almost always) has to tell the called function how many elements are in the array. So most functions that take arrays are going to have prototypes like:

```
type function_name(array argument, integer argument)
```

For example, we could take the `mean()` function from 11.3.1 and make it more useful. Instead of

```
double mean(void); // finds mean of data in s_heights
```

we can have

```
double mean(double data_array[], int numelements);
```

This version could be used any time we want to find the mean of the values in an array of doubles. Note, the array argument is specified as an "open array" i.e. the `[]` contains no number.

```
double mean(double data_array[], int numelements)
{
    double sum = 0.0;
    for(int i=0; i<numelements; i++)
        sum += data_array[i];
    return sum / numelements;
}
```

This improved version of `mean()` is used in the following code fragment:

```

int main()
{
    double heights[100];
    double h;
    int n = 0;
    cout << "Enter heights:" << endl;
    cin >> h; // Not checking anything here
    while(h > 0.0) {
        heights[n] = h;
        n++;
        cin >> h;
    }
    double m = mean(heights, n);

    cout << "The mean is " << m << endl;

    return 0;
}

```

*Function call with
array as argument*

Figure 11.1 illustrates the stack during a call to `mean()`. The hexadecimal numbers, e.g. , shown are actual addresses when the program was run on a particular machine. By convention, data addresses are always shown in hexadecimal ("hex").

The instruction sequence generated for the call `m = mean(heights, n)` would be something like:

```

load register with the address of array heights (i.e. address
of heights[0])
store in stack
load register with value of n
store in stack
subroutine call
load register with value in return slot of stack
store in m
clean up stack

```

The instructions in function `mean()` would be something like:

```

load an address register a0 with contents of
    first argument slot of stack frame
load integer register d0 with contents of
    second argument slot of stack frame
zero contents of double register f0
zero second integer register d1
loop
    compare integer registers d0 and d1
    if contents equal jump to end

```

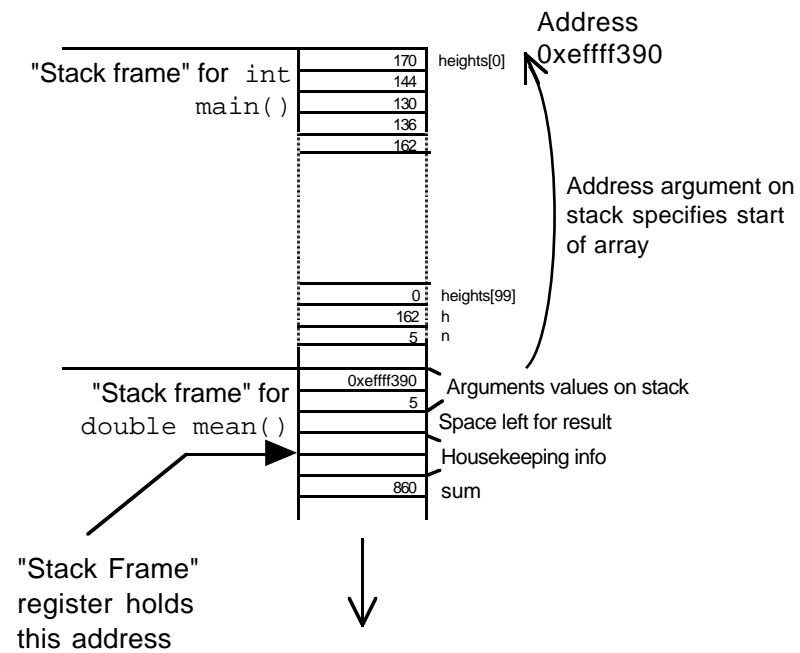


Figure 11.1 Stack with when calling `double mean(double data_array[], int numitems)`.

```

copy content d1 to register d2
multiply d2 by 8
copy content of a0 to address register a1
add content of d2 to a1
add double from memory at address a1 to register f0
add 1 to d1
jump to loop
end
convert integer in d0 to double in f1
divide f0 by f1
store f0 in return slot of stack frame

```

This code uses the array identified by its address as provided by the calling routine. Typically, it would hold this "base" address in one of the CPU's address registers. Then when array elements have to be accessed, the code works out the offset relative to this base address to get the actual address of a needed element. The data are then fetched from the memory location whose address has just been determined. (The bit about "multiplying an integer index value by 8" is just part of the mechanics of converting the index into a byte offset as needed by the hardware addressing mechanisms of a machine. Programmers may think in terms of indexed arrays, machines have to use

byte addresses so these have to be calculated. The value of the 'multiplier', the 8 in this code, depends on the type of data stored in the array; for an array of longs, the multiplier would probably be 4. The operation wouldn't actually involve a multiplication instruction, instead the required effect would be achieved using alternative quicker circuits.)

const and non const arguments

The function `mean()` doesn't change the data array it is given; it simply reads values from that array. However, functions can change the arrays that they work on. So you could have the function:

```
void change_cm_to_inches(double h[], int n)
{
    // I'm too old fashioned to be happy with cm,
    // change those height values to inches
    for(int i = 0; i < n ; i++)
        h[i] /= 2.54;
}
```

If called with the `heights[]` array, this function would change all the values (or, at least, the first `n` values) in that array. (Coding style: no `return` – that is OK in a `void` function, though I prefer a function to end with `return`; the smart `h[i] /= 2.54;` – it is correct, I still find it easier to read `h[i] = h[i] / 2.54;`)

Function `mean()` treats its array argument as "read only" data; function `change_cm_to_inches()` scribbles all over the array it was given. That is a substantial difference in behaviour. If you are going to be calling a function and giving it an array to work with, you would usually like to know what is going to happen to the contents of that array.

C++ lets you make this distinction explicit in the function prototype. Since `mean()` doesn't change its array, it regards the contents as – well constant. So, it should state this in its prototype:

```
double mean(const double data_array[], int numelements);
```

As well as appearing in function declarations, this `const` also appears where the function is defined:

```
double mean(const double data_array[], int numelements)
{
    double sum = 0.0;
    ...
}
```

The compiler will check the code of the function to make certain that the contents of the array aren't changed. Of course, if you have just declared an argument as `const`, it is unlikely that you will then write code to change the values.

The main advantage of such `const` (and default non `const`) declarations is for documenting functions during the design stage. In larger projects, with many programmers, each programmer works on individual parts and must make use of functions provided by their colleagues. Usually, all they see of a function that you have written is the prototype that was specified during design. So this prototype must tell them whether an array they pass to your function is going to be changed.

11.5 STRINGS: ARRAYS OF CHARACTERS

Sometimes you want arrays of characters, like the grade symbols used in one of the examples on array initialization:

```
const char grade_letters[6] = { 'F', 'E', 'D', 'C', 'B', 'A' };
```

The example in 11.8.2 uses a character array in a simple letter substitution approach to the encryption of messages. If you were writing a simple word processor, you might well use a large array to store the characters in the text that you were handling.

Most arrays of characters are slightly specialized. Most arrays of character are used to hold "strings" – these may be data strings (like names of customers), or prompt messages from the program, or error messages and so forth. All these strings are just character arrays where the last element (or last used element) contains a "null" character.

"null character"

The null character has the integer value 0. It is usually represented in programs as `'\0'`. All the standard output routines, and the string library discussed below, expect to work with null terminated arrays of characters. For example, the routine inside the `iostream` library that prints text messages uses code equivalent to the following:

```
void PrintString(const char msg[])
{
    int i = 0;
    char ch;
    while((ch = msg[i]) != '\0') { PrintCharacter(ch); i++; }
}
```

(The real code in the `iostream` library uses pointers rather than arrays.) Obviously, if the message `msg` passed to this function doesn't end in a null character, this loop just keeps printing out the contents of successive bytes from memory. Sooner or later, you are going to make a mistake and you will try to print a "string" that is not properly terminated; the output just fills up with random characters copied from memory.

You can define initialized strings:

```
const char eMsg1[] = "Disk full, file not saved.";
const char eMsg2[] = "Write error on disk.";
```

The compiler checks the string and determines the length, e.g. "Write ... disk." is 20 characters. The compiler adds one for the extra null character and, in effect, turns the definition into the following:

```
const char eMsg2[21] = {
    'W', 'r', 'i', 't', 'e',
    ' ', 'e', 'r', 'r', 'o',
    'r', ' ', 'o', 'n', ' ',
    'd', 'i', 's', 'k', '.',
    '\0'
};
```

If you want string data, rather than string constants, you will have to define a character array that is going to be large enough to hold the longest string that you use (you have to remember to allow for that null character). For example, suppose you needed to process the names of customers and expected the maximum name to be thirty characters long, then you would define an array:

```
char customername[31];
```

(Most compilers don't like data structures that use an odd number of bytes; so usually, a size like 31 gets rounded up to 32.)

You can initialize an array with a string that is shorter than the specified array size:

```
char filename[40] = "Untitled";
```

The compiler arranges for the first nine elements to be filled with the character constants 'U', 'n', ..., 'e', 'd', and '\0'; the contents of the remaining 31 elements are not defined.

Input and output of strings

The contents of the character array containing a string, e.g. `filename`, can be printed with a single output statement:

```
cout << filename;
```

(The compiler translates "*cout takes from character array*" into a call to a `PrintString()` function like the one illustrated above. Although `filename` has 40 characters, only those preceding the null are printed.)

cin >> character array The input routines from the `iostream` library also support strings. The form "*cin gives to character array*", e.g.:

```
cin >> filename;
```

is translated into a call to a `ReadString()` function which will behave something like the following code:

```
void ReadString(char target[])
{
    char ch;
    // Skip white space
    ch = ReadCharacter();
    while(ch == ' ')
        ch = ReadCharacter();

    // read sequence of characters until next space
    int i = 0;
    while(ch != ' ') {
        target[i] = ch;
        i++;
        ch = ReadCharacter();
    }
    // Terminate with a null
    target[i] = '\0';
}
```

The input routine skips any leading "whitespace" characters (not just the ' ' space character, whitespace actually includes things like tab characters and newlines). After all leading whitespace has been consumed, the next sequence of characters is copied into the destination target array. Copying continues until a whitespace character is read.

Caution, area where bugs lurk

Of course there are no checks on the number of characters read! There can't be. The compiler can't determine the size of the target array. So, it is up to the programmer to get the code correct. An array used in an input statement must be large enough to hold the word entered. If the target array is too small, the extra characters read just overwrite other variables. While not a particularly common error, this does happen; so be cautious. (You can protect yourself by using the `setw()` "manipulator". This was illustrated in section 9.7 where it was shown that you could set the width of an output field. You can also use `setw()` to control input; for example, `cin >> setw(5) >> data` will read 4 characters into array `data`, and as the fifth character place a '\0'.)

The `iostream` library code is written with the assumption that any character array filled in this way is going to be used as a string. So, the input routine always places a null character after the characters that were read.

getline()

With "`cin >> array`", input stops at the first white space character. Sometimes, it is necessary to read a complete line containing several words separated by spaces. There is an alternative input routine that can be used to read whole lines. This routine,

`getline()`, is part of the repertoire of things that an `istream` object can do. Its prototype is (approximately):

```
getline(char target[], int maximum_characters,  
        char delimiter = '\n');
```

Function `getline()` is given the name of the character array that should hold the input data, a limit on the number of characters to read, and a delimiter character. A call to `getline()` has a form like:

```
cin.getline(target, 39, '\n');
```

(read this as "cin object: use your `getline` function to fill the array `target` with up to 39 characters of input data").

The delimiter character can be changed. Normally, `getline()` is used to read complete lines and the delimiter should be the `'\\n'` character at the end of the line. But the function can be used to read everything up to the next tab character (use `'\\t'` as the delimiter), or any other desired character. (Caution. The editors used with some development environments use `'\\r'` instead of `'\\n'`; if you try reading a line from a text data file you may not find the expected `'\\n'`.)

Characters read from input are used to fill the target array. Input stops when the delimiter character is read (the delimiter character is thrown away, it doesn't get stored in the array). A null character is placed in the array after the characters that were read.

The integer limit specifies the maximum number of characters that should be read. This should be one less than the size of the target array so as to leave room for the null character. Input will stop after this number of characters have been read even though the specified delimiter has not been encountered. If input stops because this limit is reached, all unread characters on the input line remain in the input buffer. They will probably confuse the *next* input operation! (You can clear left over characters by using `cin.ignore()` whose use was illustrated earlier in section 8.2.)

The string library and the ctype functions

There is a standard library of routines for manipulating strings. The header file `string.h` *string.h* contains declarations for all the string functions. Some of the more commonly used functions are:

```
strcpy(char destination[], const char source[]);  
strncpy(char destination[], const char source[], int numchars);  
  
strcat(char target[], const char source[]);  
strncat(char target[], const char source[], int numchars);  
  
int strcmp(const char firststring[], const char secondstring);
```

```
int strncmp(const char firststring[], const char secondstring,
            int num_chars_to_check);

int strlen(const char string[]);
```

(These aren't the exact function prototypes given in the `string.h` header file! They have been simplified to use only the language features covered so far. The return types for some functions aren't specified and character arrays are used as arguments where the actual prototypes use pointers. Despite these simplifications, these declarations are "correct".)

**You can't assign
arrays:**

The following code does not compile:

```
char baskin_robertson[50]; // Famous for ice creams
cout << "Which flavour? (Enter 1..37) : ";
int choice;
cin >> choice;
switch(choice) {
case 1: baskin_robertson = "strawberry"; break;
case 2: baskin_robertson = "chocolate"; break;
...
case 37: baskin_robertson = "rum and raisin"; break;
default: baskin_robertson = "vanilla"; break;
}
```

If you try to compile code like that you will get some error message (maybe "lvalue required"). In C/C++, you can not do array assignments of any kind, and those "cases" in the switch statement all involve assignments of character arrays.

You can't do array assignments for the same reason that you can't pass arrays by value. The decision made for the early version of C means that arrays are just blocks of memory and the compiler can not be certain how large these blocks are. Since the compiler does not know the size of the memory blocks, it can not work out how many bytes to copy.

strcpy()

However, you do often need to copy a string from some source (like the string constants above) to some destination (the character array `baskin_robertson[]`). The function `strcpy()` from the string library can be used to copy strings. The code given above should be:

```
switch(choice) {
case 1: strcpy(baskin_robertson, "strawberry"); break;
...
default: strcpy(baskin_robertson, "vanilla"); break;
}
```

Function `strcpy()` takes two arguments, the first is the destination array that gets changed, the second is the source array. Note that the function prototype uses the

const qualifier to distinguish between these roles; `destination[]` is modified so it is not const, `source[]` is treated as read only so it is const.

Function `strcpy()` copies all the characters from the source to the destination, and adds a `'\0'` null character so that the string left in the destination is properly terminated. (Of course, there are no checks! What did you expect? It is your responsibility to make certain that the destination array is sufficient to hold the string that is copied.)

Functions `strncpy()` copies a specified number of characters from the source to the destination. It fills in elements `destination[0]` to `destination[numchars-1]`. If the source string is greater than or equal to the specified length, `strncpy()` does not place a `'\0'` null character after the characters it copied. (If the source string is shorter than the specified length, it is copied along with its terminating null character.) You can use `strncpy()` as a "safe" version of `strcpy()`. If your destination array is of size `N`, you call `strncpy()` specifying that it should copy `N-1` characters and then you place a null character in the last element:

```
const int cNAME_SIZE = 30;
...
char flavour[cNAME_SIZE];
...
strncpy(flavour, othername, cNAME_SIZE-1);
flavour[cNAME_SIZE-1] = '\0';
```

The `strcat()` function appends ("catenates") the source string onto any existing string in the destination character array. Once again it is your responsibility to make sure that the destination array is large enough to hold the resulting string. An example use of `strcat()` is:

```
char order[40] = "pizza with ";
...
strcat(order, "ham and pineapple");
```

changes the content of character array `order` from:

```
pizza with \0.....
```

to:

```
pizza with ham and pineapple\0.....
```

Function `strncat()` works similarly but like `strncpy()` it transfers at most the specified number of characters.

Function `strlen()` counts the number of characters in the string given as an argument (the terminating `'\0'` null character is not included in this count). *strlen()*

These functions compare complete strings (`strcmp()`) or a specified number of characters within strings (`strncmp()`). These functions return 1 if the first string (or *strcmp() and strncmp()*

first few characters) are "greater" than the second string, 0 if the strings are equal, or -1 if the second string is greater than the first.

Collating sequence of characters

The comparison uses the "collating" sequence of the characters, normally this is just their integer values according to the ASCII coding rules:

```

32: ' '  33: '!'  34: '"'  35: '#'  36: '$'  37: '%'
38: '&'  39: '''  40: '('  41: ')'  42: '*'  43: '+'
44: ','  45: '-'  46: '.'  47: '/'  48: '0'  49: '1'
50: '2'  51: '3'  52: '4'  53: '5'  54: '6'  55: '7'
56: '8'  57: '9'  58: ':'  59: ';'  60: '<'  61: '='
62: '>'  63: '?'  64: '@'  65: 'A'  66: 'B'  67: 'C'
...
86: 'V'  87: 'W'  88: 'X'  89: 'Y'  90: 'Z'  91: '['
92: '\'  93: ']'  94: '^'  95: '_'  96: '`'  97: 'a'
...

```

For example, the code:

```

char msg1[] = "Hello World";
char msg2[] = "Hi Mom";

cout << strcmp(msg1, msg2) << endl;

```

should print -1, i.e. msg1 is less than msg2. The H's at position 0 are equal, but the strings differ at position 1 where msg2 has i (value 105) while msg1 has e (value 101).

ctype

Another group of functions, that are useful when working with characters and strings, have their prototypes defined in the header file ctype.h. These functions include:

```

int isalnum(int);           // letter or digit?
int isalpha(int);           // letter?
int iscntrl(int);           // control character
int isdigit(int);           // digit?
int islower(int);           // One of 'a'...'z'?
int ispunct(int);           // Punctuation mark?
int isspace(int);           // Space
int isupper(int);           // One of 'A'...'Z'?
int tolower(int);           // Make 'A'...'Z' into 'a'...'z'
int toupper(int);           // Make 'a'...'z' into 'A'...'Z'
int isprint(int);           // One of printable ASCII set?

```

11.6 MULTI-DIMENSIONAL ARRAYS

You can have higher dimensional arrays. Two dimensional arrays are common. Arrays with more than two dimensions are rare.

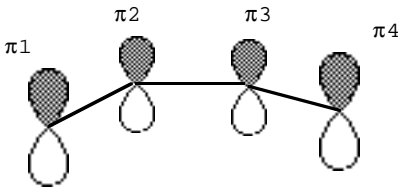
Two dimensional arrays are used:

for all kinds of tables, e.g. a spreadsheet:

Salesperson					
	1	2	3	4	5
Week					
1	12	6	10	9	23
2	9	11	12	4	18
3	13	9	8	7	19
4	8	4	6	6	13
5	3	2	7	7	15
6	9	11	13	10	19
7	0	0	0	0	0

to represent physical systems such as electronic interactions between atoms in a molecule:

π -Orbital overlap integrals			
1.00	0.52	0.08	0.01
0.52	1.00	0.44	0.08
0.08	0.44	1.00	0.52
0.01	0.08	0.52	1.00



or something like light intensities in a grey scale image (usually a very large 1000x1000 array):

200	200	180	168	100	160
200	190	180	170	99	155
200	194	170			
...							
...							
10	10	...					
10	8	4			10

and in numerous mathematical, engineering, and scientific computations, e.g calculating coordinates of the positions of objects being moved:

$$\begin{array}{c}
 \text{Scale axes} \qquad \text{Translate (move)} \qquad \text{Rotate about z} \\
 \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1.0 \end{pmatrix}
 \begin{pmatrix} 1.0 & 0 & 0 & dx \\ 0 & 1.0 & 0 & dy \\ 0 & 0 & 1.0 & dz \\ 0 & 0 & 0 & 1.0 \end{pmatrix}
 \begin{pmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 1.0 \end{pmatrix}
 \begin{pmatrix} 1.0 \\ 0.5 \\ 2.0 \\ 1.0 \end{pmatrix}
 \begin{array}{l} \text{Point to be} \\ \text{repositioned} \end{array}
 \end{array}$$

Although two dimensional arrays are not particularly common in computing science applications, they are the data structures that are most frequently needed in scientific and engineering applications. In computing science applications, the most common two dimensional arrays are probably arrays of fixed length strings (since strings are arrays of characters, an array of strings can be regarded as a two dimensional array of characters).

11.6.1 Definition and initialization

In C (C++) two dimensional arrays are defined by using the `[]` operator to specify each dimension, e.g:

```
char      screen[25][80];
double    datatable[7][11];
double    t_matrix[4][4];
```

The first `[]` defines the number of rows; the second defines the number of columns. So in the example, `screen[][]` is a two dimensional array of characters that could be used to record the information displayed in each character position of a terminal screen that had 25 rows each with 80 columns. Similarly, `datatable[][]` is an array with 7 rows each of 11 columns and `t_matrix[][]` is a square 4x4 array.

As in the case of one dimensional arrays, the array indices start at zero. So, the top-left corner of the `screen[][]` array is `screen[0][0]` and its bottom right corner is `screen[24][79]`.

Two dimensional arrays can be initialized. You give all the data for row 0 (one value for each column), followed by all the data for row 1, and so on:

```
double mydata[4][5] = {
    { 1.0, 0.54, 0.21, 0.11, 0.03 },
    { 0.34, 1.04, 0.52, 0.16, 0.09 },
    { 0.41, 0.02, 0.30, 0.49, 0.19 },
    { 0.01, 0.68, 0.72, 0.66, 0.17 }
};
```

You don't have to put in the internal { begin and } end brackets around the data for each individual row. The compiler will understand exactly what you mean if you write that initialization as:

```
double mydata[4][5] = {  
    1.0, 0.54, 0.21, 0.11, 0.03, 0.34, 1.04, 0.52, 0.16, 0.09,  
    0.41, 0.02, 0.30, 0.49, 0.19, 0.01, 0.68, 0.72, 0.66, 0.17  
};
```

However, leaving out the brackets around the row data is likely to worry human readers – so put the { } row bracketing in!

With one dimensional arrays, you could define an initialized array and leave it to the compiler to work out the size (e.g. `double vec[] = { 2.0, 1.5, 0.5, 1.0 };`). You can't quite do that with two dimensional arrays. How is the compiler to guess what you want if it sees something like:

```
double guess[][] = {  
    1.0, 0.54, 0.21, 0.11, 0.03, 0.34, 1.04, 0.52, 0.16, 0.09,  
    0.41, 0.02, 0.30, 0.49, 0.19, 0.01, 0.68, 0.72, 0.66, 0.17  
};
```

You could want a 4x5 array, or a 5x4 array, or a 2x10 array. The compiler can not guess, so this is illegal.

You must at least specify the number of columns you want in each row, but you can leave it to the compiler to work out how many rows are needed. So, the following definition of an initialized array is OK:

```
double mydata[][5] = {  
    { 1.0, 0.54, 0.21, 0.11, 0.03 },  
    { 0.34, 1.04, 0.52, 0.16, 0.09 },  
    { 0.41, 0.02, 0.30, 0.49, 0.19 },  
    { 0.01, 0.68, 0.72, 0.66, 0.17 }  
};
```

The compiler can determine that `mydata[][5]` must have four rows.

At one time, you could only initialize aggregates like arrays if they were global or filescope. But that restriction has been removed and you can initialize automatic arrays that are defined within functions (or even within blocks within functions).

11.6.2 Accessing individual elements

Individual data elements in a two dimensional array are accessed by using two [] operators; the first [] operator picks the row, the second [] operator picks the column. Mostly, two dimensional arrays are processed using nested for loops:

*Nested for loops for
processing two
dimensional arrays*

```
const int NROWS = 25;
const int NCOLS = 80;

char screen[NROWS][NCOLS];

// initialize the "screen" array to all spaces
for(int row = 0; row < NROWS; row++)
    for(int col = 0; col < NCOLS; col++)
        screen[row][col] = ' ';
```

Quite often you want to do something like initialize a square array so that all off-diagonal elements are zero, while the elements on the diagonal are set to some specific value (e.g. 1.0):

```
double t_matrix[4][4];
...
for(int i = 0; i < 4; i++) {
    for(int j = 0; j < 4; j++)
        t_matrix[i][j] = 0.0;
    t_matrix[i][i] = 1.0; // set element on diagonal
}
```

*The "cost" of
calculations*

These "cost" of these double loops is obviously proportional to the product of the number of rows and the number of columns. As you get to work with more complex algorithms, you need to start estimating how much each part of your program will "cost".

The idea behind estimating costs is to get some estimate of how long a program will run. Estimates can not be very precise. After all, the time a program takes run depends on the computer (120MHz Pentium based machine runs a fair sight faster than 15MHz 386). The time that a program takes also depends on the compiler that was used to compile the code. Some compilers are designed to perform the compilation task quickly; others take much longer to prepare a program but, by doing a more detailed compile time analysis, can produce better code that results in a compiled program that may run significantly faster. In some cases, the run time of a program depends on the actual data values entered.

Since none of these factors can be easily quantified, estimates of how long a program will take focus solely on the number of data elements that must be processed. For some algorithms, you need a reasonable degree of mathematical sophistication in order to work out the likely running time of a program implementing the algorithm. These complexities are not covered in this book. But, the estimated run times will be described for a few of the example programs.

These estimates are given in a notation known as "big-O". The algorithm for initializing an MxN matrix has a run time:

$$\text{initialize matrix} = O(M \times N)$$

or for a square matrix

$$\text{initialize matrix} = O(N \times N) \quad \text{or} \quad O(N^2)$$

Such run time estimates are independent of the computer, the compiler, and the specific data that are to be processed. The estimates simply let you know that a 50x50 array is going to take 25 times as long to process as a 10x10 array.

One of the most frequent operations in scientific and engineering programs has a cost $O(N^3)$. This operation is "matrix multiplication". Matrix multiplication is a basic step in more complex mathematical transforms like "matrix inversion" and "finding eigenvalues". These mathematical transforms are needed to solve many different problems. For example, you have a mechanical assembly of weights and springs you can represent this using a matrix whose diagonal elements have values related to the weights of the components and whose off diagonal elements have values that are determined by the strengths of the springs joining pairs of weights. If you want to know how such an assembly might vibrate, you work out the "eigenvalues" of the matrix; these eigenvalues relate to the various frequencies with which parts of such an assembly would vibrate.

Restricting consideration to square matrices, the "formula" that defines the product of two matrices is:

$$\begin{pmatrix} \sum_{i=0}^M a_{0i} * b_{i0} & \sum_{i=0}^M a_{0i} * b_{i1} & \dots & \dots & \dots & \sum_{i=0}^M a_{0i} * b_{iM} \\ \sum_{i=0}^M a_{1i} * b_{i0} & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \sum_{i=0}^M a_{Mi} * b_{i0} & \dots & \dots & \dots & \dots & \sum_{i=0}^M a_{Mi} * b_{iM} \end{pmatrix} =$$

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & \dots & \dots & a_{0M} \\ a_{10} & a_{11} & \dots & \dots & \dots & a_{1M} \\ a_{20} & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{M0} & a_{M1} & \dots & \dots & \dots & a_{MM} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} & b_{02} & \dots & \dots & b_{0M} \\ b_{10} & b_{11} & \dots & \dots & \dots & b_{1M} \\ b_{20} & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ b_{M0} & b_{M1} & \dots & \dots & \dots & b_{MM} \end{pmatrix}$$

(These are "NxN" arrays with subscripts running from 0 ... M, where $M = N - 1$). Each element in the product matrix is the sum of N product terms; these product terms combine an element from a row in the first array and an element from a column in the second array. For example, if $N = 6$, then entry `product[2][3]` is given by:

```
product[2][3] = a[2][0]*b[0][3] + a[2][1]*b[1][3] +
               a[2][2]*b[2][3] + a[2][3]*b[3][3] +
               a[2][4]*b[4][3] + a[2][5]*b[5][3];
```

Calculating such a product involves a triple loop:

***$O(N^3)$ algorithm:
matrix multiplication***

```
for(int row = 0; row < N; row++)
    for(int col = 0; col < N; col ++) {
        double prdct = 0.0;
        for(int i = 0; i < N; i++)
            prdct += a[row][i] * b[i][col];
        product[row][col] = prdct;
    }
```

(The variable `prdct` is introduced as a minor optimization. The code would be slower if you had `product[row][col] += a[row][i] * b[i][col]` because in this form it would be necessary to reevaluate the subscripts each time it needed to update `product[row][col]`.)

The $O(N^3)$ cost of this triple loop code means that if you had a program that involved mainly matrix multiplication that ran in one minute for a data set of size 10, then you would need more than two hours to process a data set of size 50.

***Another trap for
Pascal programmers***

Just a warning, you must use two separate `[]` operators when accessing an element in a two dimensional array. Now in languages like Pascal, elements of two dimensional arrays are accessed using square brackets surrounding a list of array indices separate by commas:

```
data[i,j]           Pascal style array subscripting
```

Now, in C/C++ you can write code like the following:

```
double d[10][10];
...
...
...  d[ i, j] ...
```

That form `d[i, j]` will get past the compiler. But, like several earlier examples of legal but freaky constructs, its meaning is odd.

As explained in section 7.7, a comma separated list can be used to define a set of expressions that must be evaluated in sequence. So `i, j` means "evaluate `i`, e.g. getting the value 4, and throw this value away, then evaluate `j`, eg. getting the value 9, and use this value as the final value for the complete expression".

Consequently,

```
d[ 4, 9]
```

in C++ actually means

```
d[9]
```

What is `d[9]`?

The expression `d[9]` means the tenth row of the `d[][]` array (tenth because row 0 is the first row so row 9 is the tenth row).

In C++ it is legal to refer to a complete row of an array. There aren't many places where you can do anything with a complete row of an array, but there are a few. Consequently, `d[9]` is a perfectly legal data element and the compiler would be happy if it saw such an expression.

Now, if you do accidentally type `d[i, j]` instead of `d[i][j]` you are going to end up getting an error message from the compiler. But it won't be anything like "You've messed up the array subscripts again.". Instead, the error message will be something like "Illegal operand type." Your code would be something like: `double t = 2.0 * d[i, j];`. The compiler would see `2.0*`... and expect to find a number (an int, or a long, or a double). It wouldn't be able to deal with "`d[j]` row of array" at that point and so would think that the data type was wrong.

Where might you use an entire row of an array?

There aren't many places where it is possible (and even fewer where it is sensible). But you could do something like the following. Suppose you have an array representing the spreadsheet style data shown at the start of this section. Each row represents the number of sales made by each of a set of salespersonnel in a particular week. You might want the average sales, and have a function that finds the average of the elements of an array:

Using a row of a two dimensional array

```
double average(const double d[], int n);

double salesfigures[20][10];
...
// Get average sales in week j
cout << "Average for this week " <<
    average(salesfigures[j], 10);
...
```

The `average()` function can be given a complete row of the array as shown in this code fragment.

While this is technically possible, it really isn't the kind of code you want to write. It is "clever" code using the language to its full potential. The trouble with such code is that almost everyone reading it gets confused.

11.6.3 As arguments to functions

The example just shown at the end of the preceding section illustrated (but discouraged) the passing of rows of a two-dimensional array to a function that expected a one dimensional array. What about passing the entire array to a function that wants to use a two dimensional array?

This is feasible, subject to some minor restrictions. You can't have something like the following:

```
void PrintMatrix(double d[][], int nrows, int ncols);
```

The idea would be to have a `PrintMatrix()` function that could produce a neatly formatted printout of any two dimensional array that it was given – hence the "doubly open" array `d[][]`.

The trouble is that the compiler wouldn't be able to work out the layout of the array and so wouldn't know where one row ended and the next began. Consequently, it wouldn't be able to generate code to access individual array elements.

You *can* pass arrays to functions, but the function prototype has to specify the number of columns in the arrays that it can deal with:

```
void PrintMatrix(double d[][5], int nrows);
```

The compiler will check calls made to `PrintMatrix()` to make certain that if you do call this function then any array you pass as an argument has five columns in each row.

11.7 ARRAYS OF FIXED LENGTH STRINGS

We may have a program that needs to use lots of "Messages", which for this example can be character arrays with 30 characters. We may need many of the Message things to hold different strings, and we may even need arrays of "Messages" to hold collections of strings. Before illustrating examples using arrays of strings, it is worth introducing an additional feature of C++ that simplifies the definition of such variables

Typedefs C and C++ programs can have "typedef" definitions. Typedefs don't introduce a new type; they simply allow the programmer to introduce a *synonym* for an existing type. Thus, in `types.h` header used in association with several of the libraries on Unix you will find typedefs like the following:

```
typedef long      daddr_t;
typedef long      id_t;
typedef long      clock_t;
typedef long      pid_t;
typedef long      time_t;
```


The type `daddr_t` is meant to be a "disk address"; a `pid_t` is a "process type". They are introduced so that programmers can specify functions where the prototype conveys a little bit more about the types of arguments expected, e.g.:

```
void WriteArray(double d[], long n_elems, daddr_t where);
```

The fact that `where` is specified as being a `daddr_t` makes it a little clearer that this function needs to be given a disk location.

Variables of these "types" are actually all long integers. The compiler doesn't mind you mixing them:

```
time_t a = 3;
pid_t b = 2;
id_t r = a*b;
cout << r << endl;
```

So these typedef "types" don't add anything to the compiler's ability to discriminate different usage of variables or its ability to check program correctness.

However, typedefs aren't limited to defining simple synonyms, like alternative words for "long". You can use typedefs to introduce synonyms for derived types, like arrays of characters:

```
typedef char Message[30];
```

This typedef makes "Message" a synonym for the type `char ... [30]` (i.e. the type that specifies an array of 30 characters). After this typedef has been read, the compiler will accept the definition of variables of type `Message`:

```
Message m1;
```

and arrays of `Message` variables::

```
Message error_msgs[10];
```

and declarations of functions that process `Message` arguments:

```
void PrintMessage(Message to_print);
```

Individual Messages, and arrays of Messages, can be initialized:

```
Message improbable = "Elvis lives in Wagga Wagga";
```

```
Message errors[] = {
    "Disk full",
    "Disk write locked",
```

```

        "Write error",
        "Can't overwrite file",
        "Address error",
        "System error"
    };

```

(Each Message in the array `errors[]` has 30 characters allocated; for most, the last ten or more characters are either uninitialized or also to '\0' null characters because the initializing strings are shorter than 30 characters. If you try to initialize a Message with a string longer than 30 characters the compiler will report something like "Too many initializers".)

As far as the compiler is concerned, these Messages are just character arrays and so you can use them anywhere you could have used a character array:

```

#include <iostream.h>
#include <string.h>

typedef char Message[30];

Message m1;
Message m2 = "Elvis lives in Wagga Wagga";

Message errors[] = {
    "Disk full",
    "Disk write locked",
    "Write error",
    "Can't overwrite file",
    "Address error",
    "System error"
};

    cout << Message void PrintMessage(const Message toprint)
    {
        cout << toprint << endl;
    }

    strlen() and Message int main()
    {
        cout << strlen(m2) << endl;

        strcpy() and Message PrintMessage(errors[2]);
        strcpy(m1, m2);

        PrintMessage(m1);
        return 0;
    }

```

This little test program should compile correctly, run, and produce the output:

```

26
Write error
Elvis lives in Wagga Wagga

```

A `Message` isn't anything more than a character array. You can't return any kind of an array as the result of a function so you can't return a `Message` as the result of a function. Just to restate, typedefs simply introduce synonyms, they don't really introduce new data types with different properties.

As in some previous examples, the initialized array `errors[]` shown above relies on the compiler being able to count the number of messages and convert the open array declaration `[]` into the appropriate `[6]`.

You might need to know the number of messages; after all, you might have an option in your program that lists all possible messages. Of course you can count them:

```

const int NumMsgs = 6;

...

Message errors[NumMsgs] = {
    "Disk full",
    "...",
    "System error"
};

```

but that loses the convenience of getting the compiler to count them. (Further, it increases the chances of errors during subsequent revisions. Someone adding a couple of extra error messages later may well just change `errors[NumMsgs]` to `errors[8]` and fail to make any other changes so introducing inconsistencies.)

You can get the compiler work out the number of messages and record this as a constant, or you can get the compiler to provide data so you can work this out at run time. You achieve this by using the "`sizeof()`" operator. *sizeof()*

Although `sizeof()` looks like a call to a function from some library, technically `sizeof()` is a built in operator just like `+` or `%`. It can be applied either to a type name or to a variable. It returns the number of bytes of memory that that type or variable requires:

```

int n = sizeof(errors); // n set to number of bytes needed for
                        // the variable (array) errors
int m = sizeof(Message); // m set to number of bytes needed to
                        // store a variable of type Message

```

Using such calls in your code would let you work out the number of Messages in the errors array:

```

cout << "Array 'errors' : " << sizeof(errors) << endl;

```

```
cout << "Data type Message :" << sizeof(Message) << endl;
int numMsgs = sizeof(errors)/sizeof(Message);
cout << "Number of messages is " << numMsgs << endl;
```

But you can get this done at compile time:

```
Message errors[] = {
    "Disk full",
    "...",
    "System error"
};

const int NumMsgs = sizeof(errors) / sizeof(Message);
```

This might be convenient if you needed a second array that had to have as many entries as the errors array:

```
Message errors[] = {
    "Disk full",
    ...
};

const int NumMsgs = sizeof(errors) / sizeof(Message);

int err_counts[NumMsgs];
```

11.8 EXAMPLES USING ARRAYS

11.8.1 Letter Counts

Problem

The program is to read a text file and produce a table showing the total number of occurrences for each of the letters a ... z.

This isn't quite as futile an exercise as you think. There are some uses for such counts.

For example, before World War 1, the encryption methods used by diplomats (and spies) were relatively naive. They used schemes that are even simpler than the substitution cipher that is given in the next example. In the very simplest encryption schemes, each letter is mapped onto a cipher letter e.g. a -> p, b -> n, c -> v, d -> e, e -> r, These naive substitution ciphers are still occasionally used. But they are easy to crack as soon as you have acquired a reasonable number of coded messages.

The approach to cracking substitution ciphers is based on the use of letter frequencies. Letter 'e' is the most common in English text; so if a fixed substitution like e -> r is used, then letter 'r' should be the most common letter in the coded text. If you

tabulate the letter frequencies in the encoded messages, you can identify the high frequency letters and try matching them with the frequently used letters. Once you have guessed a few letters in the code, you look for patterns appearing in words and start to get ideas for other letters.

More seriously, counts of the frequency of occurrence of different letters (or, more generally, bit pattern symbols with 8 or 16 bits) are sometimes needed in modern schemes for compressing data. The original data are analyzed to get counts for each symbol. These count data are sorted to get the symbols listed in order of decreasing frequency of use. Then a compression code is made up which uses fewer bits for high frequency symbols than it uses for low frequency symbols.

Specification:

1. The program is to produce a table showing the number of occurrences of the different letters in text read from a file specified by the user.
2. Control characters, punctuation characters, and digits are to be discarded. Upper case characters are to be converted to lower case characters so that the final counts do not take case into consideration.
3. When all data have been read from the file, a report should be printed in the following format:

a: 2570,	b: 436,	c: 1133,	d: 1203,	e: 3889,
f: 765,	g: 640,	h: 1384,	i: 2301,	j: 30,
k: 110,	l: 1296,	m: 782,	n: 2207,	o: 2108,
p: 594,	q: 51,	r: 1959,	s: 2046,	t: 2984,
u: 989,	v: 335,	w: 350,	x: 125,	y: 507,
z: 71,				

Design:

The program will be something like:

First iteration

```

get user's choice of file, open the file, if problems give up
zero the counters
loop until end of file
    get character
    if letter
        update count
print out counters in required format

```

Again, some "functions" are immediately obvious. Something like "zero the counters" is a function specification; so make it a function.

Second iteration The rough outline of the program now becomes:

```

open file
    prompt user for filename
    read name
    try opening file with that name
    if fail give up

initialize
    zero the counters

processfile
    while not end of file
        read character
        if is_alphabetic(character)
            convert to lower
            increment its counter

printcounters
    line count = 0
    for each letter a to z
        print letter and associated count in appropriate fields
        increment line count
        if line count is 4 print newline and zero line count

main
    open file
    initialize
    processfile
    printcounters

```

Third iteration At this stage it is necessary to decide how to organize the array of letter counts. The array is going to be:

```
long          lettercounts[26]
```

We seem to need an array that you can index using a letter, e.g. `lettercounts['a']`; but that isn't going to be correct. The index is supposed to be a number. The letter 'a' interpreted as a number would be 97. If we have an 'a', we don't want to try incrementing the non-existent `lettercounts[97]`, we want to increment `lettercounts[0]`. But really there is no problem here. Letters are just integers, so you can do arithmetic operations on them. If you have a character `ch` in the range 'a' to 'z' then subtracting the constant 'a' from it will give you a number in the range 0 to 25.

We still have to decide where to define the array. It could be global, filescope, or it could belong to `main()`. If it is global or filescope, all routines can access the array. If it belongs to `main()`, it will have to be passed as a (reference) argument to each of the other routines. In this case, make it belong to `main()`.

The only other point that needs further consideration is the way to handle the filename in the "open file" routine. Users of Macs and PCs are used to "File dialog" windows popping up to let them explore around a disk and find the file that they want to open. But you can't call those run-time support routines from the kinds of programs that you are currently writing. All you can do is ask the user to type in a filename. There are no problems provided that the file is in the same directory ("folder") as the program. In other cases the user is supposed to type in a qualified name that includes the directory path. Most Windows users will have seen such names (e.g. c:\bc4\examples\owlexamples\proj1\main.cpp) but most Mac users will never have seen anything like "HD:C++ Examples:Autumn term:Assignment 3:main.cp". Just let the user type in what they will; use this input in the file open call. If the name is invalid, the open operation will fail.

The array used to store the filename (a local to the `openfile` function) should allow for one hundred characters; this should suffice even if the user tried to enter a path name. The file name entered by the user should be read using `getline()` (this is actually only necessary on the Mac, Mac file names can include spaces and so might not be read properly with a "cin >> character_array" style of input).

The `ifstream` that is to look after the file had better be a `filescope` variable.

This involves nothing more than finalising the function prototypes before we start coding: **Fourth Iteration**

```
void Openfile(void);

void Initialize(int counts[]);

void Processfile(int counts[]);

void PrintCounters(const int counts[]);

int main()
```

and checking the libraries needed so that we remember to include the correct header files:

checking letters and case conversion	<code>ctype.h</code>
i/o	<code>iostream.h</code>
files	<code>fstream.h</code>
formatting	<code>iomanip.h</code>
termination on error	<code>stdlib.h</code>

Note the difference between the prototype for `PrintCounters()` and the others. `PrintCounters()` treats the counts array as read only; it should state this in its prototype, hence the `const`. None of the functions using the array takes a count argument; in this program the array size is fixed.

Implementation:

We might as well define a constant `ALPHABETSIZE` to fix the size of arrays. The "includes", filescopes, and const declaration parts of the program are then:

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <ctype.h>
#include <stdlib.h>

const int ALPHABETSIZE = 26;

static ifstream infile;
```

***Reusable OpenFile()
routine***

The `OpenFile()` routine should be useful in other programs (the prompt string would need to be replaced). Standard "cut and paste" editing will suffice to move this code to other programs; in Chapter 13 we start to look at building up our own libraries of useful functions.

```
void Openfile(void)
{
    const int kNAMEsize = 100;
    char    filename[kNAMEsize];

    cout << "Enter name of file with text for letter"
          " frequency analysis" << endl;

    cin.getline(filename, kNAMEsize-1, '\n');

    infile.open(filename, ios::in | ios::nocreate);

    if(!infile.good()) {
        cout << "Can't open that file. Quitting." << endl;
        exit(1);
    }
}
```

The check whether the file opening was unsuccessful could have been coded `if(!infile)` but such shortcuts really make code harder to read the more long winded `if(!infile.good())`.

The `Initialize()` function is pretty simple:

```
void Initialize(int counts[])
{
    // counts is supposed to be an array of
    //      ALPHABETSIZE elements
    for(int i = 0; i < ALPHABETSIZE; i++)
        counts[i] = 0;
```



```
}
```

In `Processfile()` we need all the alphabetic characters and don't want whitespace. It would have been sufficient to use `"infile >> ch"` (which would skip whitespace in the file). However the code illustrates the use of the `get()` function; many related programs will want to read all characters and so are likely to use `get()`.

```
void Processfile(int counts[])
{
    while(!infile.eof()) {
        char ch;
        infile.get(ch);
        if(!isalpha(ch))
            continue;

        ch = tolower(ch);
        int ndx = ch - 'a';
        counts[ndx]++;
    }
    infile.close();
}
```

The `isalpha()` function from `ctype.h` can be used to check for a letter. Note the use of "continue" in the loop. Probably your instructor will prefer the alternative coding:

```
...
infile.get(ch);
if(isalpha(ch)) {
    ch = tolower(ch);
    int ndx = ch - 'a';
    counts[ndx]++;
}
}
```

It is just a matter of style.

The statement:

```
ch = tolower(ch);
```

calls `tolower()` for all the alphabetic characters. You might have expected something like:

```
if(isupper(ch))
    ch = tolower(ch);
```

but that is actually more expensive. Calling `tolower()` for a lower case letter does not harm.

Function `PrintCounters()` has a loop processing all 26 characters. The `for(;;)` statement illustrates some common tricks. The main index runs from 'a' to 'z'. The subsidiary counter `j` (used for formatting lines) is defined along with the control index `i` in the start up part of the loop. Variable `i` is an integer so `cout << i` would print a number; its forced to print it as a character by the coding `cout << char(i)`.

```
void PrintCounters(const int counts[])
{
    for(int i='a', j=0; i<='z'; i++) {
        cout << char(i) << ":" << setw(5) <<
            counts[i-'a'] << ",    ";
        j++;
        if(j == 5) { j = 0; cout << endl; }
    }
    cout << endl;
}
```

The `main()` function is nice and simple as all good mains should be:

```
int main()
{
    int lettercounts[ALPHABETSIZE];
    Openfile();
    Initialize(lettercounts);
    Processfile(lettercounts);
    PrintCounters(lettercounts);
    return 0;
}
```

Someone can read that code and immediately get an overall ("abstract") idea of how this program works.

11.8.2 Simple encryption

Problem

Two programs are required; the first is to encrypt a clear text message, the second is to decrypt an encrypted message to get back the clear text.

The encryption mechanism is a slightly enhanced version of the substitution cipher noted in the last example. The encryption system is for messages where the clear text is encoded using the ASCII character set. A message will normally have printable text characters and a few control characters. The encryption scheme leaves the control characters untouched but substitutes for the printable characters.

Characters whose integer ASCII codes are below 32 are control characters, as is the character with the code 127. We will ignore characters with codes 128 or above (these

will be things like π , α , fi, \ddagger , etc.). Thus, we need only process characters whose codes are in the range 32 (space) to 126 (~) inclusive. The character value will be used to determine an index into a substitution array.

The program will have an array containing the same characters in randomized order, e.g.:

```
static int substitutions[] = {
    'U', 'L', 'd', '{', 'p', 'Q', '@', 'M',
    '-', ']', 'O', 'r', 'c', '6', '^', '#',
    'Y', 'w', 'W', 'v', '!', 'F', 'b', ')',
    '+', 'h', 'J', 'f', 'C', '8', 'B', '7',
    '.', 'k', '2', 'u', 'Z', '9', 'K', 'o',
    '3', 'x', ' ', '\', '=', '&', 'N', '*',
    'l', 'z', '(', '\'', 'R', 'P', ':', '1',
    '4', '0', 'e', '$', '_', '}', 'j', 't',
    '?', 'S', 'q', '>', ';', 'T', 'y', 'i',
    '\\', 'A', 'D', 'V', '5', '|', '<', '/',
    'E', 'g', 'm', ',', '[', 'H', '%', 'a',
    's', 'n', 'I', 'X', '~', '"', 'G'
};
```

(Note the way of handling problem characters like the single quote character and the backslash character.; these become `'` and `\`.)

You can use this to get a simple substitution cipher. Each letter of the message is converted to an index (by subtracting 32) and the letter in the array at that location is substituted for the original.

Letter substitution for enciphering

```
int ch = message[i];
int ndx = ch - 32;
int subs = substitutions[ndx];
cout << char(subs)
```

Using this simple substitution cipher a message:

```
Coordinate your attack with that of Colonel K.F.C. Sanders. No
support artillery available. Instead, you will be reinforced
by Rocket Battery 57.
```

comes out as

```
u//m;A<S[TUn/HmUS[ [S>VUaA[ \U[ \S[U/yUu/5/<T5U'^K^u^U`S<;Tm,^UUN/
U,HEE/m[USm[A55TmnUS%SA5Sq5T^UUx<,[TS;cUn/HUaA55UqTUmTA<y/m>T;U
qnU(/>VT[U2S[[TmnUF)^
```

(Can you spot some of the give away patterns. The message ends with `^` and `^UU` occurs a couple of times. You might guess `^` was `'` and `U` was space. That guess

would let you divide the text into words. You would then focus on two letter words N/, qn, etc.)

You might imagine that it was unnecessary to analyze letter frequencies and spend time trying to spot patterns. After all decrypting a message just involves looking up the encrypted letters in a "reverse substitution" table. The entry for an encrypted letter gives the original letter:

```
'J', '4', '}', '/', '[', 'v', 'M', 'K',
'R', '7', 'O', '8', 's', '(', '@', 'O',
'Y', 'P', 'B', 'H', 'X', 'l', '-', '?',
'=', 'E', 'V', 'd', 'n', 'L', 'c', '\',
'&', 'i', '>', '<', 'j', 'p', '5', '~',
'u', 'z', ':', 'F', '!', ' ', 'N', '*',
'U', '%', 'T', 'a', 'e', ' ', 'k', '2',
'{' , '0', 'D', 't', 'h', ')', '.', '\',
'S', 'w', '6', ' ', '"', 'Z', ';', 'q',
'9', 'g', '^', 'A', 'W', 'r', 'y', 'G',
'$', 'b', '+', 'x', '-', 'C', '3', 'l',
'I', 'f', 'Q', '#', 'm', ']', '|',
```

(The underlined entries correspond to indices for 'u' and '/', the first characters in the example encrypted message.)

Reversing the letter substitution process

If you have the original substitutions table, you can generate the reverse table:

```
void MakeRTable(void)
{
    for(int i = 0; i < len; i++) {
        int tem = substitutions[i];
        tem -= ' ';
        rtable[tem] = i+32;
    }
}
```

and then use it to decrypt a message:

```
void Decrypt(char m[])
{
    char ch;
    int i = 0;
    while(ch = m[i]) {
        int t = rtable[ch - 32];
        cout << char(t);
        i++;
    }
}
```

Of course, the enemy's decryption specialist doesn't know your original substitution table and so must guess the reverse table. They can fill in a guessed table, try it on the message text, and see if the "decrypted" words were real English words.

How could the enemy guess the correct table? Why not just try different possible tables? A program could easily be devised to generate the possible tables in some systematic way, and then try them on the message; the success of the decryption could also be automatically checked by looking up "words" in some computerized dictionary.

The difficulty there relates to the rather large number of possible tables that would have to be tried. Suppose all messages used only the letters a, b, c; the enemy decrypter would have to try the reverse substitution tables bac, bca, cba, cab, acb, (but presumably not abc itself). If there were four letters, there would be more possibilities: abcd, abdc, acbd, acdb, ..., dbca; in fact, 24 possibilities. For five letters, it is 120 possibilities. The number of possibilities is the number of different permutations of the set of letters. There are $N!$ (i.e. factorial(N)) permutations of N distinct symbols.

Because there are $N!$ permutations to be tried, a dumb "try everything" program $O(N!)!$ would have a running time bounded by $O(N!)$. Now, the value of $N!$ grows rather quickly with increasing N :

N	N!	time scale
1	1	1 second
2	2	
3	6	
4	24	
5	120	two minutes ago
6	720	
7	5040	
8	40320	half a day ago
9	362880	
10	3628800	ten weeks ago
11	39916800	
12	479001600	
13	6227020800	more than a lifetime
14	87178291200	
15	1307674368000	back to the stone age!
16	20922789888000	
17	355687428096000	
18	6402373705728000	Seen any dinosaurs yet?

With the letter permutations for a substitution cipher using a full character set, N is equal to 95. Any intelligence information in a message will be "stale" long before the message is decrypted.

Trying different permutations to find the correct one is impractical. But letter counting and pattern spotting techniques are very easy. As soon as you have a few hundred characters of encrypted messages, you can start finding the patterns. Simple character substitution cipher schemes are readily broken.

*You can't just guess,
but the cipher is still
weak*

A safer encryption scheme

A somewhat better encryption can be obtained by changing the mapping after each letter. The index used to encode a letter is:

$$\text{ndx} = (\text{ch} - 32 + k) \% 95;$$

the value k is incremented after each letter has been coded. (The modulo operator is used to keep the value of index in the range 0 to 94. The value for k can start at zero or at some user specified value; the amount it gets incremented can be fixed, or to make things a bit more secure, it can change in some systematic way.) Using this scheme the message becomes:

```
uEgH\<[\~<QQX{^EQ@,%UF^L6dJYQGv7Y-2y!Wbb^b %e,_mJz@FuJC
=iE$r&j`:(``leyx$jR40`SAE`APjq4eTt]X~o|,Hi;\w@LaGO{HnI#m%W@nGQ"
-c"LL8Lh.,WMYrfoDr7.W.3*T\?
```

This cipher text is a bit more secure because there aren't any obvious repeating patterns to give the enemy cryptanalyst a starting point. Apparently, such ciphers are still relatively easy to break, but they aren't trivial.

Specification:

1. Write a program "encrypt" and a program "decrypt" that encode and decode short messages. Messages are to be less than 1000 characters in length.
2. The encrypt program is to prompt the user for a filename, the encrypted message will be written to this file.
3. The encrypt program is then to loop prompting the user to enter the next line of the clear text message; the loop terminates when a blank line is entered.
4. If the line is not blank, the encrypt program checks the message line just entered. If the line contains control characters (apart from newline) or characters from the extended set (those with integer values > 127), a warning message should be printed and the line should be discarded. If the addition of the line would make the total message exceed 1000 characters, a warning message should be printed and the program should terminate.
5. If the line is acceptable, the characters are appended to those already in a main message buffer. The newline character that ended the input line should be replaced by a space.
6. When the input loop terminates, the message now held in a main message buffer should be encrypted according to the mechanism described above, the encrypted message is to be written to the file.

7. The encrypt program is to close the file and terminate.
8. The decrypt program is to prompt the user for a filename, and should then open the file (the decrypt program terminates if the file cannot be opened).
9. The message contained in the file should read and decoded character by character. The decoded message should be printed on cout. Since newlines have been removed during the encrypting process, the decrypt program must insert newlines where needed to keep the output text within page bounds.
10. The two programs should share a "substitutions" array.

Design:

The encrypt program will be something like:

First iteration

```
Open output file
Get the message
Encrypt the message
Save message to the file
Close the file
```

while the decrypt program will be:

```
Open input file
Convert substitutions table to reverse table
Decrypt the message
```

The requirement that the two programs share the same `substitutions` array is easily handled. We can have a file that contains just the definition of the initialized `substitutions` array, and `#include` this file in both programs.

The principal "functions" are immediately obvious for both programs, but some (e.g. "Get the message") may involve additional auxiliary functions.

The rough outline of the encrypt program now becomes:

Second iteration

```
Open file
    prompt user for filename
    read name
    try opening an output file with that name
    if fail give up

Get the message
    initialize message buffer
    get and check lines until the blank line is entered
        adding lines to buffer
```

```

Encrypt
    replace each character in buffer by its encryption

Save
    write string to file

Close
    just close the file

encrypt_main
    Open output file
    Get message
    Encrypt
    Save
    Close

```

while the decrypt program becomes:

```

Open file
    prompt user for filename
    read name
    try opening an input file with that name
    if fail give up

MakeRTTable
    basics of algorithm given in problem statement, minor
    changes needed to incorporate the stuff about
    the changing offset

Decrypt
    initialize
    read first character from file
    while not end of file
        decrypt character
        print character
        maybe formatting to do
        read next character from file

decrypt_main
    Open input file
    MakeRTTable
    Decrypt

```

Some of these function might be "one-liners". The `Close()` function probably will be just something like `outfile.close();`. But it is worth keeping them in as functions rather than absorbing their code back into the main routine. The chances are that sooner or later the program will get extended and the extended version will have additional work to do as it closes up.

The only function that looks as if it might be complex is "Get the message". It has to read input, check input, add valid input to the buffer, check overall lengths etc. That is too much work for one function.

The reading and checking of a single line of input can be factored out into an auxiliary routine. This main "Get the message" routine can call this "Get and check line" from inside a loop.

An input line may be bad (illegal characters), in which case the "GetMessage" controlling routine should discard it. Alternatively, an input line may be empty; an empty line should cause "GetMessage" to terminate its loop. Otherwise, GetMessage will need to know the actual length of the input line so that it can check whether it fits into the buffer. As well as filling in an array of characters with input, function "GetAndCheckLine" can return an integer status flag.

The "Get the message" outline above should be expanded to:

```

GetAndCheckLine
    use getline() to read a complete line
    if linelength is zero (final blank input line) return 0

    check that all characters are "printable", if any are
        not then return -1 to indicate a bad line

    return line length for a good line

GetMessage
    initialize message buffer and count
    n = GetAndCheckLine()
    while n != 0
        if n > 0
            (message is valid)
            check if buffer can hold new line (allow
                for extra space and terminating null)
            if buffer can't then print warning and quit

            copy message to buffer, add extra space
                and null at end
            n = GetAndCheckLine()

```

The bit about "maybe formatting to do" in function Decrypt() needs to be tightened up a bit. We need new lines put in before we reach column 80 (assumed width of the screen), but we want to avoid breaking words. The easiest way is to output a newline if we have just output a space, and our position is already beyond column 60.

At this stage it is necessary to decide how to organize data.

Third iteration

The array with the letter substitutions will get included from its own file. The array can be defined at filescope. This extra file should hold other information that the encrypt and decrypt programs would share. They would both need to know the starting value for any 'k' used to change the encoding of letters and the value of the increment that changes k. All these could be defined in the one file: "table.inc":

```

const int  kSTART = 0;
const int  kINCREMENT = 1;

static int substitutions[] = {
    'U', 'L', 'd', '{', 'p', 'Q', '@', 'M',
    '-', ']', 'O', 'r', 'c', '6', '^', '#',
    ...
    ...
    'E', 'g', 'm', ',', '[', 'H', '%', 'a',
    's', 'n', 'I', 'X', '~', '"', 'G'
};

const int len = sizeof(substitutions) / sizeof(int);

```

(The development environment you use may have specific requirements regarding file names. You want a name that indicates this file is not a header file, nor a separately compiled module; it simply contains some code that needs to be #included into .cp files. A suffix like ".inc" is appropriate if permitted in your environment.)

The `ifstream` that is to look after the file in the decrypt program can be a `filescope` variable as can the `ofstream` object used in the encrypt program. The "reverse table" used in the decrypt program can also be `filescope`.

Other character arrays can be local automatic variables of specific routines, getting passed by reference if required. The encrypt program will require a character array capable of holding a thousand character message; this can belong to its `main()` and get passed to each of the other functions. Function `GetMessage()` will have to define an array long enough to hold one line (≈ 120 characters should be plenty); this array will get passed to the function `GetAndCheckLine()`.

The Open-file functions in both programs will have local arrays to store the filenames entered by the users. The `Openfile()` function from the last example should be easy to adapt.

Fourth Iteration

This involves nothing more than finalising the function prototypes and checking the libraries needed so that we remember to include the correct header files:

Program encrypt:

```

void OpenOutfile(void);

int GetAndCheckLine(char line[], int linelen);

void GetMessage(char txt[]);

void Encrypt(char m[]);

void SaveToFile(const char m[]);

```

```

void CloseFile(void);

int main()

i/o                                iostream.h
files                             fstream.h
termination on error              stdlib.h
copying strings                   string.h

```

Program decrypt:

```

void OpenInputfile(void)

void MakeRTable(void)

void Decrypt(void)

int main()

i/o                                iostream.h
files                             fstream.h
termination on error              stdlib.h

```

Implementation:

The first file created might as well be "table.inc" with its character substitution table. This could simply be copied from the text above, but it would be more interesting to write a tiny auxiliary program to create it.

It is actually quite common to write extra little auxiliary programs as part of the implementation process for a larger program. These extra programs serve as "scaffolding" for the main program as it is developed. Most often, you write programs that produce test data. Here we can generate a substitution table.

"Scaffolding"

The code is simple. The table is initialized so that each character is substituted by itself! Not much security in that, the message would be copied unchanged. But after this initialization, the data are shuffled. There are neater ways to do this, but a simple way is to have a loop that executes a thousand times with each cycle exchanging one randomly chosen pair of data values. The shuffled table can then be printed.

```

#include <iostream.h>
#include <stdlib.h>

const int NSYMS = 95;
const int NSHUFFLES = 1000;
int main()
{
    cout << "Enter seed for random number generator" << endl;
    int s;

```

```

cin >> s;

srand(s);

char table[NSYMS];

for(int i = 0; i < NSYMS; i++)
    table[i] = i + ' ';

// shuffle
for(int j = 0; j < NSHUFFLES; j++ ) {
    int ndx1 = rand() % NSYMS;
    int ndx2 = rand() % NSYMS;

    char temp = table[ndx1];
    table[ndx1] = table[ndx2];
    table[ndx2] = temp;
}

// Output
j = 0;
for(i = 0; i < NSYMS; i++) {
    cout << " " << table[i] << " ", ";
    j++;
    if(j == 8) { cout << endl; j = 0; }
}
cout << endl;
return 0;
}

```

The output from this program has to be edited to deal with special characters. You have to change ' ' to '\\ ' and '\\ ' to '\\\\ '. Then you can substitute the generated table into the outline given above for "table.inc" with its `const` definitions and the definition of the array substitutions.

Encrypt

Program encrypt would start with `#includes` for the header files describing the standard input/output libraries etc, and a `#include` for the file "table.inc".

```

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#include "table.inc"

```

Note that the `#include` statements are different. The system files (the headers that describe the system's libraries like `iostream`) are `#included` with the filename in pointy brackets `<...>`. A project file, like `table.inc`, is `#included` with the filename in double quotes `"..."`. The different brackets/quotes used tell the compiler where to look for the file. If the filename is in quotes, then it should be in the same directory as the program code. If the filename is in `<>` brackets, then it will be located in some standard directory belonging to the development environment.

Program `encrypt` would continue with a definition of the maximum message size, and of a variable representing the output file. Then the functions would be defined.

```
const int  kMSGSIZE = 1000;

static ofstream  outfile;

void OpenOutfile(void)
{
    const int kNAMESIZE = 100;
    char  filename[kNAMESIZE];

    cout << "Enter name of message file" << endl;

    cin.getline(filename, kNAMESIZE-1, '\n');

    outfile.open(filename, ios::out);

    if(!outfile.good()) {
        cout << "Can't open that file. Quitting." << endl;
        exit(1);
    }
}
```

Function `OpenOutFile()` is a minor variation on the `Openfile()` function in the previous example.

Function `GetAndCheckLine()` is passed the line array that should be filled in, along with details of its length. The line is filled using the `cin.getline()` call, and the length of the text read is checked. If the length is zero, a terminating blank line has been entered, so the function can return. Otherwise the for loop is executed; each character is checked using `isprint()` from the `cctype` library; returns 1 (true) if the character tested is a printable character in the ASCII set. If any of the characters are unacceptable, `GetAndCheckLine()` returns -1 as failure indicator. If all is well, a 1 success result is returned.

```
int GetAndCheckLine(char line[], int linelen)
{
    cin.getline(line, linelen - 1, '\n');
    // Check for a line with no characters
    int actuallength = strlen(line);
    if(actuallength == 0)
```

```

        return 0;
    // Check contents
    // all characters should be printable ASCII
    for(int i = 0; i < actuallength; i++)
        if(!isprint(line[i])) {
            cout << "Illegal characters,"
                << " line discarded"; return -1; }
    return actuallength;
}

```

Function `GetMessage()` is passed the array where the main message text is to be stored. This array doesn't need to be blanked out; the text of the message will overwrite any existing contents. The message will be terminated by a null character, so if the message is shorter than the array any extra characters at the end of the array won't be looked at by the encryption function. Just in case no data are entered, the zeroth element is set to the null character.

Function `GetMessage()` defines a local array to hold one line and uses a standard while loop to get and process data terminated by a sentinel. In this case, the sentinel is the blank line and most of the work of handling input is delegated to the function `GetAndCheckLine()`.

```

void GetMessage(char txt[])
{
    int    count = 0;
    txt[count] = '\0';
    const int ll = 120;
    char Line[ll];
    int n = GetAndCheckLine(Line,ll);
    while(n != 0) {
        if(n > 0) {
            count += n;
            if(count > KMSGSIZE-2) {
                cout << "\n\nSorry that message is"
                    << "too long. Quitting." << endl;
                exit(1);
            }
            strcat(txt, Line);
            // add the space to the end of the message
            txt[count] = ' ';
            // update message length
            count++;
            // and replace the null just overwritten
            txt[count] = '\0';
        }
        n = GetAndCheckLine(Line,ll);
    }
}

```

Function `GetMessage()` is the one where bugs are most likely. The problem area will be the code checking whether another line can be added to the array. The array must have sufficient space for the characters of the newline, a space, and a null. Hopefully, the checks are done correctly!

It is easy to get errors where you are one out so that if you do by chance get a 999 character input message you will fill in `txt[1000]`. Of course, there is no element `txt[1000]`, so filling in a character there is an error. You change something else.

Beware of "out by 1" errors

Finding such bugs is difficult. They are triggered by an unusual set of data (most messages are either definitely shorter than or definitely longer than the allowed size) and the error action will often do nothing more than change a temporary automatic variable that was about to be discarded anyway. Such bugs can lurk undetected in code for years until someone makes a minor change so that the effect of the error is to change a variable that is still needed. Of course, the bug won't show just after that change. It still will only occur rarely. But sometime later, it may cause the program to crash.

How should you deal with such bugs?

The first strategy is never to make "out by 1" errors. There is an entire research area on "proof of program correctness" that comes up with mechanisms to check such code and show that it is correct. This strategy is considered elegant; but it means you spend a week proving theorems about every three line function.

Prove correctness

The second strategy is to recognize that such errors are possible, code carefully, and test thoroughly. The testing would focus on the special cases (no message, and a message that is just smaller than, equal to, or just greater than the maximum allowed). (To make testing easier, you might compile a version of the program with the maximum message size set to something much smaller, e.g. 25).

Code carefully and test thoroughly

A final strategy (of poor repute) is to eliminate the possibility of an error causing harm in your program. Here the problem is that you might be one out on your array subscript and so trash some other data. OK. Make the array five elements larger but only use the amount originally specified. You can be confident that that extra "slop" at the end of the array will absorb any over-runs (you know you won't be six out in your subscripting). Quick. Easy. Solved for your program. But remember, you are sending the contents of the array to some other program that has been told to deal with messages with "up to 1000 characters". You might well surprise that program by sending 1001.

Avoid

The `Encrypt()` function is simple. If `kSTART` and `kINCREMENT` are both zero it will work as a simple substitution cipher. If `kINCREMENT` is non zero, it uses the more sophisticated scheme:

```
void Encrypt(char m[])
{
    int k = kSTART;
    for(int i=0; m[i] != '\0'; i++) {
        char ch = m[i];
        m[i] = substitutions[(ch-32 +k ) % len];
        k += kINCREMENT;
    }
}
```

Functions `SaveToFile()` and `CloseFile()` are both tiny to the point of non existence. But they should still both be functions because they separate out distinct, meaningful phases of the overall processing.

```
void SaveToFile(const char m[])
{
    outfile << m;
}

void CloseFile(void)
{
    outfile.close();
}
```

A good `main()` should be simple, little more than a sequence of function calls. Function `main()` declares the array `msg` and passes it to each of the other functions that needs it.

```
int main()
{
    char msg[kMSGSIZE];
    OpenOutfile();
    cout << "Enter text of message:" << endl;
    GetMessage(msg);
    cout << "\nEncrypting and saving." << endl;
    Encrypt(msg);
    SaveToFile(msg);
    CloseFile();
    return 0;
}
```

Decrypt

Program `decrypt` is a bit shorter. It doesn't need quite as many system's header files (reading header files does slow down the compilation process so only include those that you need). It `#includes` the `table.inc` file to get the chosen substitution table and details of the modifying `kSTART` and `kINCREMENT` values. It defines file scope variables for its reverse table and its input file.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

#include "table.inc"

static int rtable[len];
static ifstream infile;
```


The program has an `OpenInputfile()` function that is not shown here; this function is similar to that in the letter counting example. Function `MakeRTable()` makes up the reverse substitution table from the data in the #included file with the definition of array substitutions. The code here is slightly different from that illustrated in the problem description. Rather than have `rtable` hold the character, it holds the index number of the character. This is done to simplify the arithmetic in the `Decrypt()` function.

```
void MakeRTable(void)
{
    for(int i = 0; i < len; i++) {
        int tem = substitutions[i];
        tem -= ' ';
        rtable[tem] = i;
    }
}
```

Function `Decrypt()` reads characters of the encrypted message from the file and outputs the original message characters. Newlines are inserted at convenient points. The decryption mechanism reverses the encryption. The crypto character is looked up in the array, getting back the index of the character selected in the `Encrypt()` routine. This isn't the original character from the clear text message because it will have been offset by the `k` term. So this offset is reversed. That process gives the index number of the original character, adding ' ' converts this index into the ASCII code.

```
void Decrypt(void)
{
    int ch;
    int linepos = 0;
    int k = kSTART;
    ch = infile.get();
    while(ch != EOF) {
        int t = rtable[ch - 32];
        t -= k;
        while(t < 0) t += len;
        cout << char(t + ' ');

        if((t == 0) && (linepos > 60))
            { cout << endl; linepos = 0; }
        k += kINCREMENT;
        ch = infile.get();
    }
    cout << endl;
}
```

The `main()` routine is simply a sequence of function calls:

```

int main()
{
    OpenInputfile();
    MakeRTable();
    Decrypt();
    return 0;
}

```

11.8.3 Simple image processing

Problem

Most pictures input to computers are simply retouched, resized, recoloured, and then displayed or printed. But in some areas like "Artificial Intelligence" (AI), or "Pattern Recognition" (PR), or "Remote Sensing", programs have to be written to interpret the data present in the image.

One simple form of processing needed in some AI and PR programs involves finding the outlines of objects in an image. The image data itself will consist of an array; the elements in this array may be simple numbers in the range 0 (black) ... 255 (white) for a "grey scale" image, or they could be more complex values representing the colour to be shown at each point in the image (grey scale data will be used to simplify the example). A part of an image might be:

```

220 220 218 216 222 220 210 10 12 10 8 ...
220 222 214 218 220 214 216 8 8 8 ...
230 228 220 220 218 216 40 8 7 ...
228 226 226 220 221 200 16 10 4 ...
220 224 218 220 220 60 10 8 5 ...
218 220 226 220 54 16 9 6 ...
215 218 220 200 22 11 8 ...
210 220 214 48 10 8 7 ...
208 216 202 12 8 4 ...
206 210 104 8 8 6 ...
208 212 164 16 8 8 ...
210 216 200 60 20 9 ...
212 220 218 200 32 8 6 ...
...
...

```

This image has a jagged "edge" running approximately diagonally leftwards to a corner and then starting to run to the right. Its position is marked by the sudden change in intensities.

Finding and tracing edges is a little too elaborate to try at this stage. But a related simpler task can be performed. It is sometimes useful to know which row had the largest difference in intensities in successive columns (need to know row, column

number of first of column pair, and intensity difference) and, similarly, which column had the largest difference between successive rows.

These calculations involve just searches through the array considering either rows or columns. If searching the rows, then for each row you find the largest difference between successive column entries and compare this with the largest difference found in earlier rows. If you find a larger value, you update your records of where the maximum difference occurred.

The searches over the rows and over the columns can be handled by rather similar functions – a `ScanRows()` function and a `ScanCols()` function. The functions get passed the image data. They initialize a record of "maximum difference" to zero, and then use a nested double loop to search the array.

There is one slightly odd aspect to these functions – they need to return three data values. Functions usually compute a value i.e. a single number. But here we need three numbers: row position, column position, and maximum intensity difference. So, there is a slight problem.

Multiple results from a function

Later (Chapters 16 and 17) we introduce `struct` types. You can define a `struct` that groups together related data elements, like these three integer values (they don't really have independent meaning, these value belong together as a group). A function can return a `struct`, and that would be the best solution to the current problem.

However, there is an alternative approach that, although not the best, can be used here. This mechanism is required in many other slightly more elaborate problems that will come up later. This approach makes the functions `ScanRows()` and `ScanCols()` into procedures (i.e. `void` return type) and uses "reference" arguments to communicate the results.

As explained in section 11.4, arrays are always "passed by reference". The calling function places the address of the start of the array in the stack. The called function takes this address from the stack and places it in one of the CPU's address registers. When the called function needs to access an element of an array, it uses the contents of this address register while working out where required data are in memory. The data value is then fetched and used, or a new value is stored at the identified location.

Pass by reference again

Simple types, like integers and doubles, are usually passed by value. However, you can request the "pass by reference" mechanism. If you specify pass by reference, the calling function places details of the address of the variable in the stack (instead of its value). The called function, knowing that it is being given an address, can proceed in much the same way as it does when passed an array. Its code arranges to load the address of the argument into an address register. When the argument's value is needed the function uses this address register to help find the data (just like accessing arrays except that there is no need to add on an "index" value).

Simple types as reference arguments

Since the function "knows" where the original data variables are located in memory, it can change them. This makes it possible to have a function that "returns" more than one result. A function can have any number of "reference" arguments. As it runs, it can change the values in those variables that were passed by the calling function. When the function finishes, its local stack variables all disappear, but the changes it made to

the caller's variable remain in effect. So, by changing the values of several reference arguments, a function can "return" several values.

Specifying pass by reference

When you define the function prototype, you chose whether it uses pass by value or pass by reference for simple variables (and for `structs`). If don't specify anything, you get pass by value. If you want pass by reference you write something like:

```
void MultiResultFunction(int& ncount, double& dmax);
```

Reference types

Here, `&` is used as a qualifier that turns a simple type, e.g. `int`, into a "reference type", e.g. `int&`. Apart from a few very rare exceptions, the only place you will see variables of reference types being declared is in function prototypes.

In the C++ source code for `MultiResultFunction()`, the reference variable `ncount` and `dmax` will be treated just like any other `int` and `double` variables:

```
void MultiResultFunction(int& ncount, double& dmax)
{
    int m, d;
    cout << "Current count is " << ncount << endl;
    ...
    cin >> ...;
    ...
    ncount += 2;
    ...
    dmax = 0;
    cin >> d;
    ...
    dmax = (d > dmax) ? d : dmax;
    return;
}
```

Because `ncount` and `dmax` are references, the compiler generates instructions for dealing with them that differ from those used to access ordinary variables like `m` and `d`. The compiler "knows" that the value in the reference variables are essentially addresses defining where to find the data. Consequently, the generated machine code is always of the form "load address register from reference variable, use address register's contents when accessing memory to get data value, use data value, (store back into memory at place specified by contents of address register)".

Input, output, and input-output parameters

Variables passed by values are sometimes referred to as "input parameters"; they provide input to a function but they allow only one way communication. Reference variables allow two way communication. The function can use the values already in these variable (as above where the value of `ncount` is printed) or ignore the initial values (as with `dmax` in the code fragment above). Of course they allow output from a function – that was why we introduced them. So here, `ncount` is an "input output" (or "value result") parameter while `dmax` is an output (or "result") parameter.

There are some languages where the "function prototypes" specify whether a reference argument is an output (result) or an input-output (value result) parameter.

C++ does not have any way of indicating this; but this kind of information should form part of a function's description in the program design documentation.

When passing arrays, we sometimes used `const` to indicate that the function treated the array as "read only" (i.e. the array was an "input parameter"), so why is the following inappropriate?

```
void Testfunction(const int& n)
{
    ...
}
```

This is legal, but it is so much simpler to give `Testfunction()` its own copy of a simple variable like an `int`. The following would be much more appropriate:

```
void Testfunction(int n)
{
    ...
}
```

When we get to `structs`, we will find that these can be passed by value or by reference. Passing a struct by value means that you must make a copy. As this copying process may be expensive, structs are often passed by reference even when they are "input parameters". In such circumstances, it is again appropriate to use the `const` qualifier to flag the role of the argument.

Now, to return to the original problem of the image processing program, we can achieve the required multi-result functions `ScanRows()` and `ScanCols()` by using pass by reference. These functions will be given the image data as `const` array argument along with three integers that will be passed by reference (these will be "output" or "result" arguments because their initial values don't matter).

Specification:

1. Write a program that will analyze "image data" to identify the row with the maximum difference in intensity between successive columns, and the column with the maximum difference in intensity between successive rows.
2. For the purposes of this program, an image is a rectangular array of 40 rows and 50 columns. The intensity data can be encoded as short integers (or as unsigned short integers as all data values should be positive).
3. The program is to prompt the user for a filename; the image data will be read from this file. The file will contain 2000 numbers; the first 50 values represent row 0, the next 50 represent row 1 and so forth. The image data can be assumed to be "correct"; there will be 2000 numbers in the file, all numbers will be positive and in

a limited range. The format will have integer values separated by spaces and newlines.

4. The program is identify the positions of maximum difference and print details in the following style:

```
The maximum difference in a row occurred in row 7
    between column 8 and column 9, where the difference was 156
The maximum difference in a column occurred in column 8
    between row 3 and row 4, where the difference was 60
```

Design:

First iteration The program will be something like:

```
Open input file and load image from file
Check rows for maximum
Print row details
Check columns for maximum
Print column details
```

A program with three functions – `LoadImage()`, `ScanRows()`, and `ScanColumns()` – along with `main()` seems appropriate. The output statements should be simple and can be handled in `main()` rather than made into separate output routines.

Second iteration The specification essentially guarantees that the input will be valid, so there will be no need to check the data read. Provided the file can be opened, a double `for` loop should be able to read the data. The array size, and loop limits, should be defined by `const` data elements; that will make things easier to change (and to test, using and a small array 8x10 rather than 40x50).

The `ScanRows()` and `ScanCols()` will have very similar structures. They have to initialize the "maximum difference" to zero (and the position to 0, 0). Then there will be an outer loop over the number of rows (columns) and an inner loop that will run to one less than the number of columns (rows). In order to find a difference, this inner loop has to compare elements '[0]' to '[N-2]' with elements '[1]' to '[N-1]'. The structure for both these routines will be something like the following (using rows for this example):

```
scan
    initialize max_difference, row, col all to zero
    for row 0 to maxrows - 1 do
        for col = 0 to maxcols -2 do
            difference = image[row][col+1] -
                        image[row][col]
            difference = abs (difference);
            if( difference > max_difference)
                replace max_difference, row, col
```

As this is a simple problem, we have already finished most of the design. work. All *Third iteration* that remains is deciding on data representations and function prototypes.

Here it seems worth while using a "typedef" to introduce the name `Image` as name for a two dimensional array of short integers:

```
const int kROWS = 40;
const int kCOLS = 50;

typedef short  Image[kROWS][kCOLS];
```

"Type Image"

This will make the function prototypes a little clearer – we can specify an `Image` as an argument.

The program only needs one `Image`. This `Image` could be made a filescope variable but it is generally better style to have it defined as local to `main()` and arrange that it is passed to the other functions. (It doesn't make much difference in this simple program; but eventually you might need to generalize and have more than one `Image`. Then you would find it much more convenient if the `ScanRows()` and `ScanCols()` functions used an argument to identify the `Image` that is to be processed.)

Following these decisions, the function prototypes become:

Function prototypes

```
void LoadImage(Image picy);

void ScanRows(const Image picy, int& row, int& col, int& delta)

void ScanCols(const Image picy, int& row, int& col, int& delta)

int main();
```

Obviously `LoadImage()` changes the `Image` belonging to `main()`; the "scan" functions should treat the image as read-only data and so define it as `const`.

The two scan functions have three "reference to integer" (`int&`) arguments as explained in the problem introduction.

The header files will just be `iostream` and `fstream` for the file input, and `stdlib` (needed for `exit()` which will be called if get an open failure on the image file).

Implementation:

There is little of note in the implementation. As always, the program starts with the `#includes` that load the header files that describe the library functions employed; then there are `const` and `typedef` declarations:

```
#include <iostream.h>
#include <fstream.h>
```

```
#include <stdlib.h>

const int kROWS = 40;
const int kCOLS = 50;

typedef short Image[kROWS][kCOLS];
```

In this example, the file opening and input of data are all handled by the same routine so it has the `ifstream` object as a local variable:

```
void LoadImage(Image picy)
{
    const int kNAME_SIZE = 100;
    char filename[kNAME_SIZE];

    cout << "Enter name of image file" << endl;

    cin.getline(filename, kNAME_SIZE-1, '\n');
    ifstream infile;

    infile.open(filename, ios::in | ios::nocreate);
    if(!infile.good()) {
        cout << "Couldn't open file." << endl;
        exit(1);
    }
    for(int i = 0; i < kROWS; i++)
        for(int j = 0; j < kCOLS; j++)
            infile >> picy[i][j];
    infile.close();
}
```

Generally, one would need checks on the data input. (Checks such as "Are all values in the specified 0...255 range? Have we hit the end of file before getting all the values needed?") Here the specification said that no checks are needed.

The input routine would need to be changed slightly if you decided to save space by making `Image` an array of `unsigned char`. The restricted range of intensity values would make it possible to use an `unsigned char` to represent each intensity value and this would halve the storage required. If you did make that change, the input routine would have to use a temporary integer variable: `{ unsigned short temp; infile >> temp; picy[i][j] = temp; }`. If you tried to read into an "unsigned char" array, using `infile >> picy[i][j]`, then individual (non-blank) characters would be read and interpreted as the intensity values to initialize array elements!

The `ScanRows()` function is:

```
void ScanRows(const Image picy, int& row, int& col, int& delta)
{
    delta = 0;
    row = 0;
```



```

        col = 0;

        for(int r = 0; r < kROWS; r++)
            for(int c = 0; c < kCOLS-1; c++) {
                int d = picy[r][c+1] - picy[r][c];
                d = fabs(d);
                if(d>delta) {
                    delta = d;
                    row = r;
                    col = c;
                }
            }
    }
}

```

Function `ScanCols()` is very similar. It just switches the roles of row and column.

Apart from the function calls, most of `main()` consists of output statements:

```

int main()
{
    int    maxrow, maxcol, maxdiff;
    Image thePicture;
    LoadImage(thePicture);
    cout << "Image loaded" << endl;

    ScanRows(thePicture, maxrow, maxcol, maxdiff);
    cout << "The maximum difference in a row occurred in row "
          << maxrow
          << "\n\tbetween column " << maxcol
          << " and column " << (maxcol+1)
          << ", where the difference was "
          << maxdiff << endl;

    ScanCols(thePicture, maxrow, maxcol, maxdiff);
    cout << "The maximum difference in a column occurred in"
          << "column " << maxcol
          << "\n\tbetween row " << maxrow
          << " and row " << (maxrow+1)
          << ", where the difference was "
          << maxdiff << endl;

    return 0;
}

```

You would have to test this program on known test data before trying it with a real image! The values of the consts `kROWS` and `kCOLS` can be changed to something much smaller and an artificial image file with known data values can be composed. As noted in the previous example, it is often necessary to build such "scaffolding" of auxiliary test programs and test data files when developing a major project.

EXERCISES

- 1 The sending of encrypted text between parties often attracts attention – *"What do they want to hide?"*!

If the correspondents have a legitimate reason for being in communication, they can proceed more subtly. Rather than have M. Jones send K. Enver the message

```
'q%V"Q \IH,n,h^ps%Wr"XX)d6J{cM6M.bWu6rhobh7!W.NZ..Cf8
q_lK7.1_'Zt2:q)PP::N_jV{j_XE8_i\ $DaK>yS
```

(i.e. "Kathy Darling, the wife is going to her mother tonight, meet me at Spooners 6pm, Kisses Mike")

this secret message can be embedded in a longer carrier message that appears to be part of a more legitimate form of correspondence. No one is going to look twice at the following item of electronic mail:

```
From: M. Jones
To: K. Enver
```

```
Re: Report on Library committee meeting.
```

```
Hi Kate, could you please check this rough draft report that I've typed; hope
you can call me back today if you have any problems.
```

```
-----
Meeting of the Library Committee, Tuesday August 15th,
Present: D. Ankers, R Bailey, J.P. Cauroma, ...
...
```

Letters from the secret message are substituted for letters in the body of the original (the underlines shown above mark the substitutions, of course the underlining would not really be present). Obviously, you want to avoid any regularity in the pattern of "typing errors" so the embedding is done using a random number generator.

The random number generator is first seeded (the seed is chosen by agreement between the correspondents and then set each time the program is run). The program then loops processing characters from the carrier message and characters from the secret text. In these loops, a call is first made to the random number generator – e.g. `n = (rand() % IFREQ) + 3;` the value obtained is the number of characters from the carrier message that should be copied unchanged; the next character from the secret text is then substituted for the next character of the carrier text. The value for `IFREQ` should be largish (e.g. 50) because you don't want too many "typing errors". Of course, the length of the carrier text has to be significantly greater than $(IFREQ/2)$ times the secret text. "Encrypting" has to be abandoned if the carrier text is too short (you've been left with secret text characters but have hit end of file on the carrier). If the carrier is longer, just substitute a random letter at the next point where a secret text letter is required (you wouldn't want the apparent typing accuracy to

suddenly change). These extra random characters don't matter; the message will just come to an end with something like "Kisses MikePrzyqhn Tuaoa". You can rely on an intelligent correspondent who will stop reading at the point where the extracted secret text no longer makes sense.

Write the programs needed by your good fiends Mike and Kate.

The "encrypting" program is to take as input:

- an integer seed for the random number generator,
- the message (allow them a maximum of 200 characters),
- the name of the file containing the original carrier text,
- the name of the file that is to contain the adjusted text.

The encrypting program should print a warning "Bad File" if the carrier text is too short to embed the secret text; otherwise there should be no output to the screen.

The "decrypting" program is to take as input:

- an integer seed for the random number generator,
- the name of the file containing the incoming message,

it should extract the characters of the secret text and output them on cout.

2. Change the encrypt and decrypt programs so that instead of sharing a fixed substitutions table, they incorporate code from the auxiliary program that generates the table.

The modified programs should start by prompting for three small positive integers. Two of these define values for variables that replace `kSTART` and `kINCREMENT`. The third serves as the seed for the random number generator that is used to create the substitutions table.

3. Write a program that will lookup up details of distances and travel times for journeys between pairs of cities.

The program is to use a pair of initialized arrays to store the required data, one array to hold distances, the other travel times. For example, if you had just three cities, you might have a distances array like:

```
const int NCITIES = 3;
const double distances[NCITIES][NCITIES] = {
    { 0.0, 250.0, 710.0 },
    { 250.0, 0.0, 525.0 },
    { 710.0, 525.0, 0.0 }
};
```

with a similar array for travel times.

The program is to prompt the user to enter numbers identifying the cities of interest, e.g.

```
Select starting city:
1) London,
2) Paris,
3) Venice
City Number :
```

Given a pair of city identification numbers, the program prints the details of distance and travel time. (Remember, users generally prefer numbering schemes that start at 1 rather

than 0; convert from the users' numbering scheme to an internal numbering suitable for zero based arrays.)

You should plan to use several functions. The city names should also be stored in an array.

4. The arrays of distance and travel time in exercise 3 are both symmetric about the diagonal. You can halve the data storage needed by using a single array. The entries above the diagonal represent distances, those below the diagonal represent travel times.

Rewrite a solution to assignment 3 so as to exploit a single travel-data array.

12 Programs with functions and arrays

This chapter has a few examples that show some of the things that you can do using functions and arrays. A couple are small demonstration programs. Some, like the first example are slightly different. These examples— "curses", "menus", and "keywords" — entail the development of useful group of functions. These groups of functions can be used a bit like little private libraries in future examples.

12.1 CURSES

Curses?

Yes, well programs and curses are strongly associated but this is something different. On Unix, there is a library called "curses". It allows programs to produce crude "graphical" outputs using just low "cost cursor addressable" screens. These graphics are the same quality as the character based function plotting illustrated in 10.11.2. Unix's curses library is actually quite elaborate. It even allows you to fake a "multi-windowing" environment on a terminal screen. Our curses are less vehement, they provide just a package of useful output functions.

These functions depend on our ability to treat the computer screen (or just a single window on the screen) as a "cursor addressable terminal screen". Basically, this means that this screen (or window) can be treated as a two-dimensional array of individually selectable positions for displaying characters (the array dimension will be up to 25 rows by 80 columns, usually a bit smaller). The run-time support library must provide a `gotoxy()` function and a `putcharacter()` function. Special character oriented input functions must also be used.

Such facilities are available through the run time support libraries provided with Symantec C++ for PowerPC, and Borland's IDE. In the Borland system, if you want cursor addressing facilities you have to create your project as either a "DOS Standard"

*"Cursor
addressable" screens
and windows*

*Run-time support in
common IDEs*

project, or an "EasyWin" project (there are slight restrictions on EasyWin and cursor graphics).

Naturally, the run-time support libraries in Symantec and Borland differ. So here we have to develop code that will contain conditional compilation directives so that appropriate code is produced for each different environment.

Separate header and implementation files

This code is to be in a separate file so that it can be used in different programs that require simple graphics output. Actually, there will be two files. Like the standard libraries, our curses "package" will consist of an implementation file with the definitions of the functions and a header file. The header file contains just the function prototypes and some constants. This header will be #included into the programs that need to use the curses display facilities.

We will need a small program to test the library. A "drawing program" would do. This could start by displaying a blank window with a "pen position" shown by a character such as a '*'. Keyed commands would allow the user to move the pen up, down, left and right; other commands could let the "ink" to be changed from a drawing pattern ('#') to the background ('.'). This would allow creation of pictures like that shown in Figure 12.1.

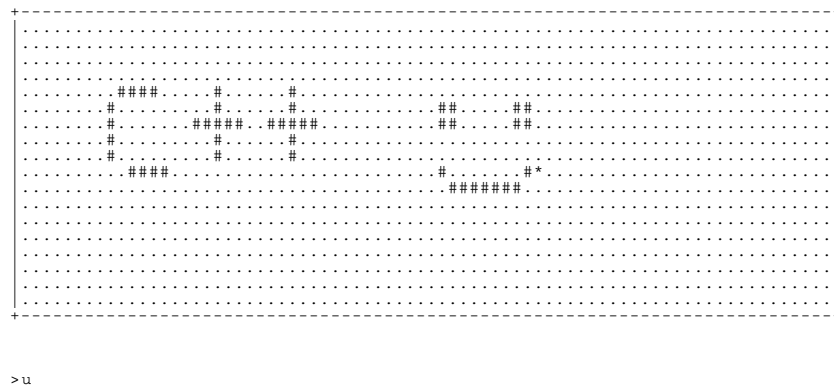


Figure 12.1 Picture produced using the cursor graphics package.

The IDE projects for programs built using the curses functions will have to identify the CG.cp ("Curses Graphics") file as one of the constituent files along with the normal main.cp. However, the header file, CG.h, will not be explicitly listed among the project files.

As noted earlier, the curses functions depend on a system's library that contains primitives like a `gotoxy()` function. IDEs normally have two groups of libraries – those that always get checked when linking a program, and those that are only checked when explicitly identified to the linker. In both the Symantec and Borland IDEs, the special run-time routines for cursor addressing are in files separate from the more common libraries. However, in both Symantec 8 and Borland, this extra library file is

among those checked automatically. In Symantec 7, the library with the cursor routines has to be explicitly made part of the project. If you do get linking errors when you try to build these programs, you will have to check the detailed documentation on your IDE to find how to add non-standard libraries to the set used by the linking loader.

Figure 12.2 illustrates how these programs will be constructed. A program using the curses routines will have at least two source files specified in its project; in the figure they are `test.cp` (which will contain `main()` and other functions) and `CG.cp`. Both these source files `#include` the `CG.h` routine. The `CG.cp` file also contains `#include` statements that get the header files defining the run-time support routines that it uses (indicated by the file `console.h` in the figure). The compiler will produce the linkable files `test.o` and `CG.o`. The linking loader combines these, and then adds all necessary routines from the normally scanned libraries and from any specifically identified additional libraries.

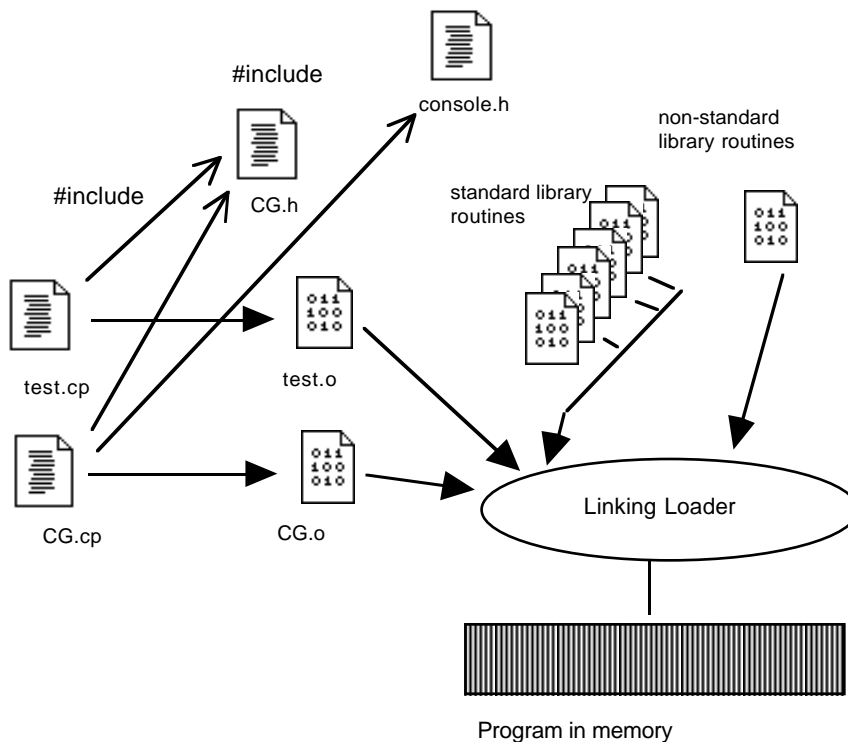


Figure 12.2 Creating a program from multiple files

Curses functions

The curses functions present the applications programmer with a simple window environment as shown in Figure 12.3. Characters are displayed by moving a "cursor" to a specific row and column position and then outputting a single character. If there are no intervening cursor movements, successive output characters will go in successive columns of the current row. (Output to the last column or last row of the screen should be avoided as it may cause unwanted scrolling of the display.)

As shown in Figure 12.3, the screen is normally divided into a work area (which may be marked out by some perimeter border characters), and a few rows at the bottom of the screen. These last few rows are used for things such as prompt line and data input area.

Input is usually restricted to reading single characters. Normally, an operating system will collect all characters input until a newline is entered; only then can the data be read by a program. This is not usually what one wants in an interactive program. So, instead, the operating system is told to return individual characters as they are entered.

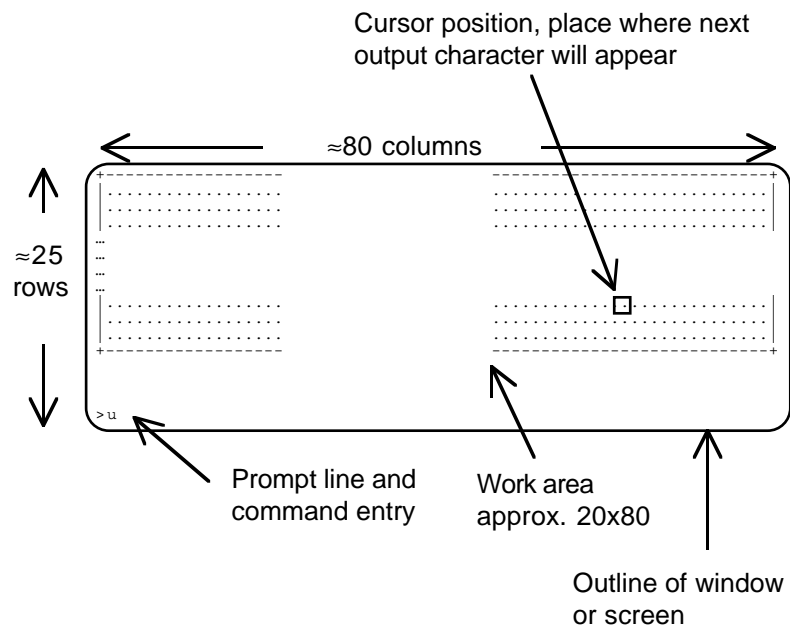


Figure 12.3 Curses model for an output window.

Often interactive programs need to display data that change. A program may need to arrange to pause for a bit so that the user can interpret one lot of data on the screen

before the display gets changed to show something else. Consequently, curses packages generally include a `delay()` function.

A curses package will have to provide at least the following:

*Minimal
functionality required
in a curses package*

- 1 Details of the width and height of the effective display area.
- 2 An initialize function.

In many environments, special requests have to be made to the operating system to change the way in which the terminal is controlled. These requests should go in an initialize function.

- 3 A reset function.

This would make the equivalent request to the operating system to put the terminal back into normal mode.

- 4 A clear window function.

This will clear all rows and columns of the display area, setting them to a chosen "background" character.

- 5 A frame window function.

Clears the window, then fills in the perimeter (as in the example shown in Figure 12.1).

- 6 A move cursor function.

This will use the IDE's `gotoxy()` function to select the screen position for the next character drawn ("positioning the cursor").

- 7 A put character function

Output a chosen character at the cursor's current position.

- 8 A prompt function that outputs a string on the prompt line.

- 9 A get character routine that returns the next character entered.

- 10 A delay function that can cause a delay of a specified number of seconds.

Prototypes of these functions are defined in the `CG.h` header file:

```
#ifndef __CGSTUFF__
#define __CGSTUFF__
```

CG.h

```
/*
Routines for a "cursor graphics" package.

These routines provide a consistent interface, there are
differing implementations for Intel (Borland) and PPC
(Symantec)
platforms.

This cursor graphics package is minimalist, it doesn't attempt
any optimizations.

The windows available for cursor graphics are usually 25 lines
by 80 columns. Since output to the last line or last column
can sometimes cause undesired scrolling, a smaller work area is
defined.
*/

const int CG_WIDTH = 78;
const int CG_HEIGHT = 20;
const int CG_PROMPTLINE = 24;

/*
Initialize --- call before using cursor screen.
Reset --- call when finished.
*/

void CG_Initialize();
void CG_Reset();

/*
Movement and output
*/
void CG_MoveCursor(int x, int y);
void CG_PutCharacter(char ch);
void CG_PutCharacter(char ch, int x, int y);
void CG_ClearWindow(char background = ' ');
void CG_FrameWindow(char background = ' ');

/*
Delay for graphics effects
*/
void CG_Delay(int seconds);

/*
Prompting and getting input
*/
void CG_Prompt(const char prompt[]);
char CG_GetChar();

#endif
```

This file organization illustrates "good style" for a package of related routines. Firstly, the entire contents are bracketed by a conditional compilation directive:

```
#ifndef __CGSTUFF__
#define __CGSTUFF__

...

...
#endif
```

*Directives to protect
against "multiple
include" errors*

This mechanism is used to avoid compilation errors if the same header file gets #included more than once. Before the file is read, the compile time "variable" `__CGSTUFF__` will be undefined, so `#ifndef` ("if not defined") is true and the rest of the file is processed. The first operation defines the "compile time variable" `__CGSTUFF__`; if the file is read again, the contents are skipped. If you don't have such checks, then multiple inclusions of the file lead to compilation errors relating to redefinition of constants like `CG_WIDTH`. (In larger programs, it is easy to get headers multiply included because header files may themselves have `#include` statements and so a particular file may get referenced in several places.)

A second stylistic feature that you should follow is illustrated by the naming of constants and functions. All constants and functions start with the character sequence `CG_`; this identifies them as belonging to this package. Naming conventions such as this make it much easier for people working with large programs that are built from many separate source files. When they see a function called in some code, the function name indicates the "package" where that it is defined.

Naming conventions

The functions provided by this curses package are grouped so as to make it easier for a prospective user to get an idea of what the package provides. Where appropriate, default argument values are defined.

Run time support from IDEs

The different development environments provide slightly different versions of the same low-level support functions. Although the functions are provided, there is little in the IDEs' documentation to indicate how they should be used. The Borland Help system and manuals do at least provide full details of the individual functions.

*Run-time support
functions of the IDEs*

In the Borland environment, the header file `conio.h` contains prototypes for a number of functions including:

```
int getche(void);           // get and echo character
void gotoxy(int x,int y);   // position cursor
int putch(int ch);          // put character
```

These specialized calls work directly with the DOS operating system and do not require any special changes to "modes" for keyboards or terminals. Borland "DOS/Standard" applications can use a `sleep()` function defined in `dos.h`.

In the Symantec environment, the header file `console.h` contains a prototype for

```
void cgotoxy(int x, int y, FILE *);    // position cursor
```

(the `FILE*` argument actually is set to a system provided variable that identifies the window used for output by a Symantec program). The prototype for the `sleep()` function is in `unix.h`, and the file `stdio.h` contains prototypes for some other functions for getting (`fgetc()`) and putting (`fputc()` and `fflush()`) individual characters. The handling of input requires special initializing calls to switch the input mode so that characters can be read immediately.

Delays in programs

The best way to "pause" a program to allow a user to see some output is to make a call to a system provided function. Following Unix, most operating systems provide a `sleep()` call that suspends a program for a specified number of seconds; while the program is suspended, the operating system runs other programs or deals with background housekeeping tasks. Sometimes, such a system call is not available; the Borland "EasyWin" environment is one such case. EasyWin programs are not permitted to use the DOS `sleep()` function.

If you need a delay in a program and don't have a `sleep()` function, or you need a delay shorter than one second, then you have a couple of alternatives. Both keep the program busy using the CPU for a specified time.

The more general approach is to use a compute loop:

```
void delay(int seconds)
{
    const long fudgefactor = 5000;
    double x;
    long lim = seconds*fudgefactor;
    while(lim>0) {
        x = 1.0/lim;
        lim--;
    }
}
```

The idea is that the calculation involving floating point division will take the CPU some microseconds; so a loop with thousands of divisions will take a measurable time. The `fudgefactor` is then adjusted empirically until appropriate delays are obtained.

There are a couple of problems. You have to change the `fudgefactor` when you move to a machine with a different CPU speed. You may get caught by an optimising compiler. A good optimising compiler will note that the value of `x` in the above code is never used; so it will eliminate the assignment to `x`. Then it will note that the loop has essentially no body. So it eliminates the loop. The assignment to `lim` can then be omitted; allowing the optimising compiler to reduce the function to `void delay(int)`

{ } which doesn't have quite the same properties. The compilers you usually use are much less aggressive about optimizing code so computational loops often work.

An alternative is to use system functions like `TickCount()` (see 10.10). If your system has `TickCount()` function that returns "seconds since machine switched on", you can achieve a delay using code like:

```
void delay(int seconds)
{
    long lim = TickCounter() + seconds;
    while(TickCounter() < lim)
        ;
}
```

(The function call in the loop will inhibit an optimising compiler, it won't try to change this code). Most compilers will give a warning about the empty body in the while loop; but it is exactly what we would need here.

Implementation of curses functions

The code to handle the cursor graphics functions is simple; it is largely comprised of calls to the run-time support functions. This code makes fairly heavy use of conditional compilation directives that select the specific statements required. The choice amongst alternatives is made by #defining one of the compiler time constants SYMANTEC, DOS, or EASYWIN.

The first part of the file #includes the appropriate system header files:

```
/*
Implementation of Cursor Graphics for Symantec 8
and Borland (either DOS-standard or EasyWin)

Versions compiled are controlled by compile time #defines
    SYMANTEC      for Mac/PC
    DOS
    EASYWIN
*/

#define SYMANTEC

#if defined(SYMANTEC)
/*
stdio is needed for fputc etc;
console is Symantec's set of functions like gotoxy
unix for sleep function
*/
#include <stdio.h>
#include <console.h>
#include <unix.h>
```

*First part of CG.cp,
#including selected
headers*

```

#endif

#if defined(DOS)
/*
conio has Borland's cursor graphics primitives
dos needed for sleep function
*/
#include <conio.h>
#include <dos.h>
#endif

#if defined(EASYWIN)
/*
Still need conio, but can't use sleep, achieve delay by
alternative mechanism
*/
#include <conio.h>
#endif

#include "CG.h"

```

All version need to include CG.h with its definitions of the constants that determine the allowed width and height of the display area.

The various functions in the package are then defined. In some versions, the body of a function may be empty.

***File CG.cp,
Initialize() and
Reset() functions***

```

void CG_Initialize()
{
#if defined(SYMANTEC)
/*
Have to change the "mode" for the 'console' screen.
Putting it in C_CBREAK allows characters to be read one by one
as they are typed
*/
    csetmode(C_CBREAK, stdin);
#else
/*
No special initializations are needed for Borland environments
*/
#endif
}

void CG_Reset()
{
#if defined(SYMANTEC)
    csetmode(C_ECHO, stdin);
#endif
}

```

The `MoveCursor()` function makes the call to the appropriate run-time support routine. Note that you cannot predict the behaviour of the run-time routine if you try to move the cursor outside of the screen area! The run-time routine might well crash the system. Consequently, this `MoveCursor()` function has to constrain the arguments to fit within the allowed range:

```
void CG_MoveCursor(int x, int y)
{
    x = (x < 1) ? 1 : x;
    x = (x > CG_WIDTH) ? CG_WIDTH : x;

    y = (y < 1) ? 1 : y;
    y = (y > CG_HEIGHT) ? CG_HEIGHT : y;
    #if defined(SYMANTEC)
        cgotoxy(x,y,stdout);
    #else
        gotoxy(x,y);
    #endif
}
```

*File CG.cp:
MoveCursor()
function*

Function `PutCharacter()` is "overloaded". Two versions are provided with slightly different argument lists. The first outputs a character at the current position. The second does a move to an explicit position before outputting the character. Note that the second function calls the more primitive `MoveCursor()` and `PutCharacter(char)` routines and does not duplicate any of their code. This is deliberate. If it is necessary to change some detail of movement or character output, the change need be made at only one place in the code.

```
void CG_PutCharacter(char ch)
{
    #if defined(SYMANTEC)
        fputc(ch, stdout);
        fflush(stdout);
    #elif
        putchar(ch);
    #endif
}

void CG_PutCharacter(char ch, int x, int y)
{
    CG_MoveCursor(x,y);
    CG_PutCharacter(ch);
}
```

*File CG.cp,
PutCharacter()
functions*

The `ClearWindow()` function uses a double loop to fill put "background" characters" at each position. (The run-time support routines provided by the IDEs may include additional "clear to end of screen" and "clear to end of line" functions that might be quicker if the background character is ' ').

The `FrameWindow()` function uses `ClearWindow()` and then has loops to draw top, bottom, left and right edges and individual output statements to place the four corner points.

*File CG.cp
ClearWindow() and
FrameWindow()
functions*

```
void CG_ClearWindow(char background)
{
    for(int y=1;y<=CG_HEIGHT;y++)
        for(int x=1; x<=CG_WIDTH;x++)
            CG_PutCharacter(background,x,y);
}

void CG_FrameWindow(char background)
{
    CG_ClearWindow(background);
    for(int x=2; x<CG_WIDTH; x++) {
        CG_PutCharacter('-',x,1);
        CG_PutCharacter('-',x,CG_HEIGHT);
    }
    for(int y=2; y < CG_HEIGHT; y++) {
        CG_PutCharacter('|',1,y);
        CG_PutCharacter('|',CG_WIDTH,y);
    }
    CG_PutCharacter('+',1,1);
    CG_PutCharacter('+',1,CG_HEIGHT);
    CG_PutCharacter('+',CG_WIDTH,1);
    CG_PutCharacter('+',CG_WIDTH,CG_HEIGHT);
}
```

The `Delay()` function can use the system `sleep()` call if available; otherwise it must use something like a computational loop:

*File CG.cp, Delay()
function*

```
void CG_Delay(int seconds)
{
    #if defined(EASYWIN)
    /*
    The EasyWin environment does not allow use of dos's sleep()
    function.
    So here do a "computational delay"
    The value 5000 will have to be adjusted to suit machine
    */
    const long fudgefactor = 5000;
    double x;
    long lim = seconds*fudgefactor;
    while(lim>0) {
        x = 1.0/lim;
        lim--;
    }
    #else
```



```
        sleep(seconds);  
    #endif  
}
```

Function `Prompt()` outputs a string at the defined prompt position. This code uses a loop to print successive characters; your IDE's run time routines may include a "put string" function that might be slightly more efficient.

The `GetChar()` routine uses a run time support routine to read a single character from the keyboard.

```
void CG_Prompt(const char prompt[])  
{  
    #if defined(SYMANTEC)  
        cgotoxy(1, CG_PROMPTLINE, stdout);  
    #else  
        gotoxy(1, CG_PROMPTLINE);  
    #endif  
    for(int i=0; prompt[i] != '\0'; i++)  
        CG_PutCharacter(prompt[i]);  
}  
  
char CG_GetChar()  
{  
    #if defined(SYMANTEC)  
        return fgetc(stdin);  
    #elif  
        return getche();  
    #endif  
}
```

*File CG.cp, Prompt()
and GetChar()
functions*

Example test program

Specification

The program to test the curses package will:

- 1 Display a window with a "pen" that the user can move by keyed commands.
- 2 The initial display is to show a "framed window" with '.' as a background character and the pen (indicated by a '*') located at point 10, 10 in the window. A '>' prompt symbol should be displayed on the promptline.
- 3 The pen can either be in "draw mode" or "erase mode". In "draw mode" it is to leave a trail of '#' characters as it is moved. In "erase mode" it leaves background '.' characters. Initially, the pen is in "draw mode".

- 4 The program is to loop reading single character commands entered at the keyboard. The commands are:

q or Q	terminate the loop and exit from the program
u or U	move the pen up one row
d or D	move the pen down one row
l or L	move the pen left one column
r or R	move the pen right one column
e or E	switch pen to erase mode
i or I	switch pen to ink mode

Any other character input is to be ignored.

- 5 The pen movement commands are to be restricted so that the pen does not move onto the window border.

Design

First iteration There is nothing much to this one. The program structure will be something like:

```

move pen
    output ink or background symbol at current pen position
    update pen position, subject to restrictions
    output pen character, '*', at new pen position

main
    initialize
    loop until quit command
        get command character
        switch to select
            change of pen mode
            movements
    reset

```

The different movements all require similar processing. Rather than repeat the code for each, a function should be used. This function needs to update the x, y coordinate of the pen position according to delta-x and delta-y values specified as arguments. The various movement cases in the switch will call the "move pen" function with different delta arguments.

Second iteration The pen mode can be handled by defining the "ink" that the pen uses. If it is in "erase mode", the ink will be a '.' character; in "draw mode" the character will be '#'.

The variables that define the x, y position and the ink have to be used in both `main()` and "move pen"; the x and y coordinates are updated by both routines. The x, y values could be made filescope – and therefore accessible to both routines. Alternatively, they

could be local to `main()` but passed by reference in the call to "move pen"; the pass by reference would allow them to be updated.

The only filescope data needed will be constants defining limits on movement and the various characters to be used for background, pen etc. The drawing limit constants will be defined in terms of the window-limit constants in the `#included CG.h` file.

File `CG.h` would be the only header file that would need to be included.

The loop structure and switch statement in `main()` would need a little more planning before coding. The loop could be a `while` (or a `for`) with a termination test that checks an integer (really a `boolean` but we don't have those yet in most C++ implementations) variable. This variable would initially be false (0), and would get set to true (1) when a 'quit' command was entered.

Third iteration

The switch should be straightforward. The branches for the movement commands would all contain just calls to the move pen function, but with differing delta x and delta y arguments.

Function `MovePen()` will have the prototype:

```
void MovePen(int& x, int& y, int dx, int dy, char ink)
```

The x, y coordinates are "input/output" ("value/result") arguments because the current values are used and then updated. These must therefore be passed by reference. The other arguments are "input" only, and are therefore passed by value.

Implementation

The file `test.cp` starts with its one `#include` (no need for the usual `#include <iostream.h>` etc). The CG files (`CG.h` and `CG.cp`) will have to be in the same directory as the test program. This doesn't cause any problems in the Borland environment as you can have a project folder with several different target programs that share files. In the Symantec environment, you may find it necessary to copy the CG files into the separate folders associated with each project.

After the `#include`, the constants can be defined:

```
#include "CG.h"

const int XMIN = 2;
const int XMAX = CG_WIDTH-1;
const int YMIN = 2;
const int YMAX = CG_HEIGHT-1;

const char pensym = '*';
const char background = '.';
const char drawsym = '#';
```

Function `MovePen()` is straightforward:

```

void MovePen(int& x, int& y, int dx, int dy, char ink)
{
    CG_PutCharacter(ink, x, y);

    x += dx;
    x = (x >= XMIN) ? x : XMIN;
    x = (x <= XMAX) ? x : XMAX;

    y += dy;
    y = (y >= YMIN) ? y : YMIN;
    y = (y <= YMAX) ? y : YMAX;

    CG_PutCharacter(pensym,x,y);
}

```

Function `main()` begins with declarations and the initialization steps:

```

int main()
{
    char ink = drawsym;

    int x = 10;
    int y = 10;

    int done = 0;

    CG_Initialize();
    CG_FrameWindow('.');
    CG_PutCharacter(pensym,x,y);
}

```

Here, the loop is done using `for(; !done;);`; a while loop might be more natural:

```

for( ; !done; ) {
    CG_Prompt(">");
    int ch;
    ch = CG_GetChar();
}

```

The switch handles the various possible commands as per specification:

```

switch(ch) {
case 'i':
case 'I':
    // put pen in drawing mode, the default
    ink = drawsym;
    break;
case 'e':
case 'E':
    // put pen in erase mode
}

```

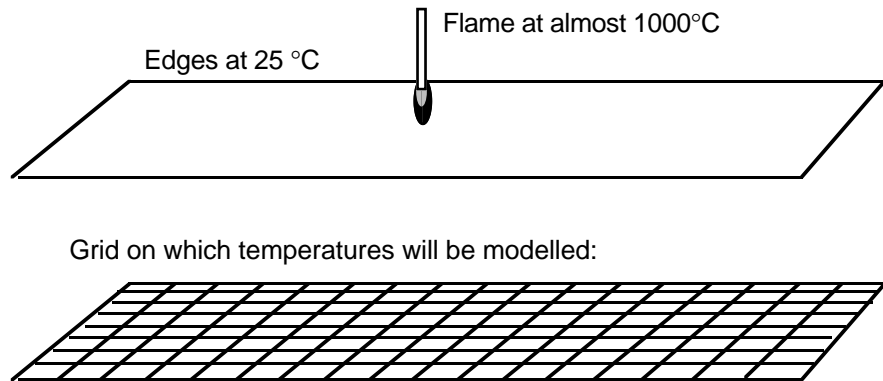
```
        ink = background;
        break;
    case 'u':
    case 'U':
        MovePen(x, y, 0, -1, ink);
        break;
    case 'd':
    case 'D':
        MovePen(x, y, 0, 1, ink);
        break;
    case 'l':
    case 'L':
        MovePen(x, y, -1, 0, ink);
        break;
    case 'r':
    case 'R':
        MovePen(x, y, 1, 0, ink);
        break;
    case 'q':
    case 'Q':
        done = 1;
        break;
    default:
        break;
    }
    CG_Reset();
    return 0;
}
```

12.2 HEAT DIFFUSION

Back around section 4.2.2, we left an engineer wanting to model heat diffusion in a beam. We can now do it, or at least fake it. The engineer didn't specify the heat diffusion formula so we'll just have to invent something plausible, get the program to work, and then give it to him to code up the correct formula.

Figure 12.4 illustrates the system the engineer wanted to study. A flame is to heat the mid point of a steel beam (of undefined thickness), the edges of which are held at ambient temperature. The heat diffusion is to be studied as a two dimensional system. The study uses a grid of rectangles representing areas of the beam, the temperatures in each of these rectangles are to be estimated. The grid can obviously be represented in the program by two dimensional array.

The system model assumes that the flame is turned on and immediately raises the temperature of the centre point to flame temperature. This centre point remains at this temperature for the duration of the experiment.



```
typedef double Grid[51][15];
```

Figure 12.4 The heat diffusion experiment.

Thermal conduction soon raises the temperature of the eight immediately neighboring grid square. In turn they warm their neighbors. Gradually the centre point gets to be surrounded by a hot ring, which in turn is surrounded by a warm ring. Eventually, the system comes to equilibrium.

The program is to display these changes in temperature as the system evolves. The temperatures at the grid squares are to be plotted using different characters to represent different temperatures ranges. Thus, '#' could represent temperatures in excess of 900°C, '@' for temperatures from 800 to 899°C and so forth. Figure 12.5 illustrates such plots.

The model for the heat diffusion is simple. (But it doesn't represent the real physics!) An iterative process models the changes of temperatures in unit time intervals. The temperature at each grid point increases or decreases so that is closer to the average of the surrounding eight grid points. For example, consider a point adjacent to the spot that is being heated, just after the flame is turned on:

25	25	25	25	25	...
25	<u>25</u>	25	25	25	...
25	25	1000	25	...	
25	25	25	25	...	

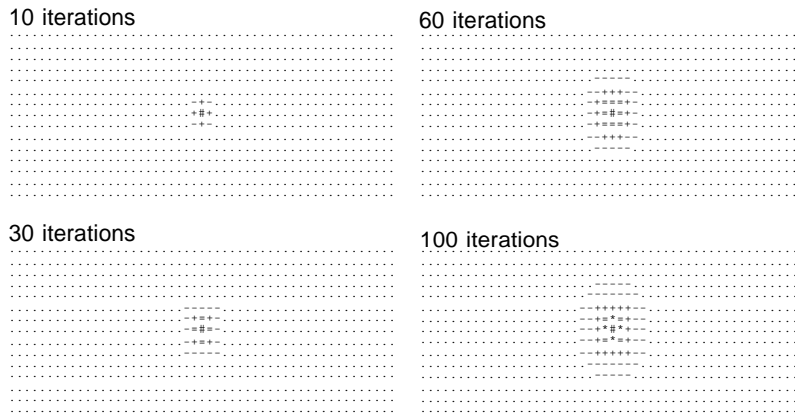


Figure 12.5 Temperature plots for the heat diffusion example.

The average of its eight neighbors is 146.8 $((1000 + 7 \cdot 125)/8)$. So, this point's temperature should increase toward this average. The rate at which it approaches the average depends on the thermal conductivity of the material; it can be defined by a formula like:

$$\text{new_temp} = \text{conduct} * \text{average} + (1 - \text{conduct}) * \text{current}$$

with the value of `conduct` being a fraction, e.g. 0.33. If the formula is defined like this, the new temperature for the marked point would be 65.1. This results in the temperature distribution

```
...
 25  25  25  25  25  ...
 25  65 65  65  25  ...
 25  65 1000 65 ...
 25  65  65  65 ...
...
```

(The centre point is held at 1000°C by the flame.) On the next iteration, the average temperature of this point's neighbors would be 156.8, and its temperature would increase to 95°C.

This process has to be done for all points on the grid. Note that this will require two copies of the grid data. One will hold the existing temperatures so that averages can be computed; the second is filled in with the new values. You can't do this on the same grid because if you did then when you updated one point you would change inappropriately the environment seen by the next point.

Once the values have been completed, they can be displayed. Changes in single iterations are small, so normally plots are needed after a number iterative cycles have

been completed. The "plotting" could use the cursor graphics package just implemented or standard stream output. The plot simply requires a double loop to print the values in each grid point. The values can be converted into characters by dividing the range (1000 - 25) into ten intervals, finding the interval containing a given temperature and using a plotting character selected to represent that temperature interval.

Specification

The program to model a two dimensional heat diffusion experiment:

- 1 The program will model the process using a two dimensional array whose bounds are defined by constants. Other constants in the program will specify the ambient and flame temperatures, and a factor that determines the "thermal conductivity" of the material.
- 2 The program will prompt the user for the number of iterations to be performed and the frequency of displays.
- 3 Once modelling starts, the temperature of the centre point is held at the "flame temperature", while the perimeter of the surface is held at ambient temperature.
- 4 In each iteration, the temperatures at each grid point (i.e. each array element) will be updated using the formulae described in the problem introduction.
- 5 Displays will show the temperature at the grid points using different characters to represent each of ten possible temperature ranges.

Design

First iteration The program structure will be something like:

```
main
  get iteration limit
  get print frequency
  initialize grid
  loop
    update values in grid to reflect heat diffusion
    if time to print, show current values
```

Second iteration Naturally, this breaks down into subroutines. Function `Initialize()` is obvious, as are `HeatDiffuse()` and `Show()`.

```
Initialize
```



```

double loop sets all temperatures to "ambient"
temperature of centre point set to "flame"

Show
double loop
  for each row do
    for each column do
      get temperature code character
      for this grid point
        output character
    newline

HeatDiffuse
  get copy of grid values
  double loop
  for each row do
    for each col do
      using copy work out average temp.
        of environment of grid pt.
      calculate new temp based on current
        and environment
      store in grid
  reset centre point to flame temperature

```

While Initialize() is codeable, both Show() and HeatDiffuse() require additional auxiliary functions. Operations like "get temperature code character" are too complex to expand in line.

Additional functions have to be defined to handle these operations:

Third iteration

```

CodeTemperature
// find range that includes temperature to be coded
set value to represent ambient + 0.1 * range
index = 0;
while value < temp
  index++, value += 0.1*range
use index to access array of characters representing
the different temperature ranges

Copy grid
double loop filling copy from original

New temperature
formula given conductivity*environment + (1- cond)*current

Average of neighbors
find average of temperatures in three neighbors in row
above, three neighbors in row below, and left, right
neighbors

```

While most of these functions are fairly straightforward, the "average of neighbors" does present problems. For grid point r, c , you want:

$$\begin{aligned} &g[r-1][c-1] + g[r-1][c] + g[r-1][c+1] + \\ &g[r][c-1] + g[r][c+1] + \\ &g[r+1][c-1] + g[r+1][c] + g[r+1][c+1] \end{aligned}$$

But what do you do if $r == 0$? There is no -1 row.

The code must deal with the various special cases when the point is on the edge and there is either no adjacent row or no adjacent column. Obviously, some moderately complex conditional tests must be applied to eliminate these cases.

Anything that is "moderately complex" should be promoted into a separate function! An approach that can be used here is to have a `TemperatureAt()` function that is given a row, column position. If the position is within the grid, it returns that temperature. Otherwise it returns "ambient temperature". The "average of neighbors" function can use this `TemperatureAt()` auxiliary function rather than accessing the grid directly.

```
TemperatureAt(row, col)
    if row outside grid
        return ambient
    if col outside grid
        return ambient
    return grid[r][c]

Average of neighbors(row, col)
    ave = - grid[row][col]
    for r=row-1 to row+1
        for col = col-1 to col+1
            ave += TemperatureAt(r,c)
    return ave/8
```

The sketch for the revised "Average of Neighbors" uses a double loop to run through nine squares on the grid. Obviously that includes the temperature at the centre point when just want to consider its neighbors; but this value can be subtracted away.

Final stages

All that remains is to decide on function prototypes, shared data, and so forth. As in earlier examples, the function prototypes are simplified if we use a typedef to define "Grid" to mean an array of doubles:

```
typedef double Grid[kWIDTH][kLENGTH];
```

The prototypes are:

```
char CodeTemperature(double temp);

void Show(const Grid g);

void CopyGrid(Grid dest, const Grid src);
```

```

double TemperatureAt(const Grid g, int row, int col);

double AverageOfNeighbors(const Grid g, int row, int col);

double NewTemperature(double currenttemp, double envtemp);

void HeatDiffuse(Grid g);

void Initialize(Grid g);

int main();

```

The main `Grid` variable could have been made filescope and shareable by all the functions but it seemed more appropriate to define it as local to main and pass it to the other functions. The `const` qualifier is used on the `Grid` argument to distinguish those functions that treat it as read only from those that modify it.

There should be no filescope variables, just a few constants. This version of the program uses stream output to display the results so `#includes <iostream.h>`.

Implementation

The file starts with the `#includes` and constants. The array `TemperatureSymbols[]` has increasingly 'dense' characters to represent increasingly high temperatures.

```

#include <iostream.h>
#include <assert.h>

const int kWIDTH = 15;
const int kLENGTH = 51;

const double kAMBIENT = 25;
const double kFLAME = 1000;
const double kRANGE = kFLAME - kAMBIENT;
const double kSTEP = kRANGE / 10.0;

const double kCONDUCT = 0.33;

const int kMIDX = kLENGTH / 2;
const int kMIDY = kWIDTH / 2;

const char TemperatureSymbols[] = {
    '.', '-', '+', '=', '*', 'H', '&', '$', '@', '#'
};

typedef double Grid[kWIDTH][kLENGTH];

```

Function `CodeTemperature()` sets the initial range to (25, 120) and keeps incrementing until the range includes the specified temperature. This should result in an index in range 0 .. 9 which can be used to access the `TemperatureSymbols[]` array.

```
char CodeTemperature(double temp)
{
    int i = 0;
    double t = kAMBIENT + kSTEP;
    while(t < temp) { i++; t += kSTEP; }

    assert(i < (sizeof(TemperatureSymbols)/sizeof(char)));

    return TemperatureSymbols[i];
}
```

Both `Show()` and `CopyGrid()` use similar double loops to work through all elements of the array, in one case printing characters and in the other copying values. `Show()` outputs a `\f` or "form feed" character. This should clear the window (screen) and cause subsequent output to appear at the top of the window. This is more convenient for this form of display than normal scrolling.

The curses library could be used. The routine would then use `CG_PutCharacter(char, int, int)` to output characters at selected positions.

```
void Show(const Grid g)
{
    cout << '\f';
    for(int r=0; r< kWIDTH; r++) {
        for(int c = 0; c < kLENGTH; c++)
            cout << CodeTemperature(g[r][c]);
        cout << endl;
    }
    cout << endl;
}

void CopyGrid(Grid dest, const Grid src)
{
    for(int r=0;r<kWIDTH;r++)
        for(int c=0;c<kLENGTH; c++) dest[r][c] = src[r][c];
}
```

The `TemperatureAt()` and `AverageOfNeighbors()` functions are straightforward codings of the approaches described in the design:

```
double TemperatureAt(const Grid g, int row, int col)
{
    if(row < 0)
        return kAMBIENT;
    if(row >= kWIDTH)
        return kAMBIENT;
```

```

    if(col < 0)
        return kAMBIENT;

    if(col >= kLENGTH)
        return kAMBIENT;

    return g[row][col];
}

double AverageOfNeighbors(const Grid g, int row, int col)
{
    double Average = -g[row][col];

    for(int r = row - 1; r <= row + 1; r++)
        for(int c = col - 1; c <= col + 1; c++)
            Average += TemperatureAt(g, r, c);

    return Average / 8.0;
}

```

As explained previously, there is nothing wrong with one line functions if they help clarify what is going on in a program. So function `NewTemperature()` is justified (you could define it as `inline double NewTemperature()` if you didn't like paying the cost of a function call):

```

double NewTemperature(double currenttemp, double envtemp)
{
    return (kCONDUCT*envtemp + (1.0 - kCONDUCT)*currenttemp);
}

```

Function `HeatDiffuse()` has to have a local `Grid` that holds the current values while we update the main `Grid`. This copy is passed to `CopyGrid()` to be filled in with current values. Then get the double loop where fill in the entries of the main `Grid` with new computed values. Finally, reset the centre point to flame temperature.

```

void HeatDiffuse(Grid g)
{
    Grid temp;
    CopyGrid(temp, g);
    for(int r = 0; r < kWIDTH; r++)
        for(int c = 0; c < kLENGTH; c++) {

            double environment = AverageOfNeighbors(
                                temp, r, c);

            double current = temp[r][c];
            g[r][c] = NewTemperature(
                                current, environment);
        }
    g[kMIDY][kMIDX] = kFLAME;
}

```

```

    }

void Initialize(Grid g)
{
    for(int r=0;r<kWIDTH;r++)
        for(int c=0;c<kLENGTH; c++) g[r][c] = kAMBIENT;
    g[kMIDY][kMIDX] = kFLAME;
}

```

The main program is again simple, with all the complex processing delegated to other functions.

```

int main()
{
    int nsteps;
    int nprint;
    cout << "Number of time steps to be modelled : ";
    cin >> nsteps;
    cout << "Number of steps between printouts : ";
    cin >> nprint;

    Grid g;
    Initialize(g);

    for(int i = 0, j = 0; i < nsteps; i++) {
        HeatDiffuse(g);
        j++;
        if(j==nprint) { j = 0; Show(g); }
    }
    return 0;
}

```

If you have a very fast computer, you may need to put a delay after the call to `Show()`. However, there are enough floating point calculations being done in the loops to slow down the average personal computer and you should get sufficient time to view one display before the next is generated. A suitable input data values are `nsteps == 100` and `nprint == 10`. The patterns get more interesting if the flame temperature is increased to 5000°C.

12.3 MENU SELECTION

Design of a single utility routine

Many programs need to present their users with a menu of choices. This kind of commonly required functionality can be separated out into a utility routine:

```

int MenuSelect(...);

```

Normal requirements include printing a prompt, possibly printing the list of choices, asking for an integer that represents the choice, validating the choice, printing error message, handling a ? request to get the options relisted, and so forth. This function can be packaged in a little separately compiled file that can be linked to code that requires menu selection facilities.

The function has to be given:

- 1 an initial prompt ("Choose option for ...");
- 2 an indication as to whether the list of choices should be printed before waiting for input;
- 3 an array with the strings defining the choices;
- 4 an integer specifying the number of choices.

The code will be along the following lines:

```
print prompt
if need to list options
    list them
repeat
    ask for integer in specified range
    deal with error inputs
    deal with out of range
until valid entry
```

Dealing with out of range values will be simple, just requires an error message reminding the user of the option range. Dealing with errors is more involved.

Errors like end of file or "unrecoverable input" error are probably best dealt with by returning a failure indicator to the calling program. We could simply terminate the program; but that isn't a good choice for a utility routine. It might happen that the calling program could have a "sensible default processing" option that it could use if there is no valid input from the user. Decisions about terminating the program are best left to the caller.

Design, second iteration

There will also be recoverable input errors. These will result from a user typing alphabetic or punctuation characters when a digit was expected. These extraneous characters should be discarded. If the first character is a '?', then the options should be displayed.

Sorting out unrecoverable and recoverable errors is too elaborate to be expanded in line, we need an auxiliary function. These choices lead to a refined design:

```
handle error
    check input status,
    if end of file or bad then
        return "give up" indicator (?0)
```

```

        read next character
        if ? then
            list options
        remove all input until newline
        return "retry" indicator (?1)

menuselect
    print prompt
    if need to list options
        list them
    repeat
        ask for integer in specified range
        read input

        if not input good
            if handle_error != retry
                return failure
            continue

        check input against limits
        if out of range
            print details of range, and
                "? to list options"
    until valid entry
    return choice

```

Design, third iteration

The activity "list options" turns up twice, so even though this will probably be a one or two line routine it is worth abstracting out:

```

list options
    for each option in array
        print index number, print option text

```

The `MenuSelect()` function has to be given the prompt and the options. These will be arrays of characters. Again it will be best if a typedef is used to introduce a name that will allow simplification of argument lists and so forth:

```

const int UT_TXTLENGTH = 60;

typedef char UT_Text[UT_TXTLENGTH];

```

The `UT_` prefix used in these names identifies these as belonging to a "utility" group of functions.

Conventions have to be defined for the values returned from the `MenuSelect()` function. A value -1 can indicate failure; the calling routine should terminate the program or find some way of managing without a user selection. Otherwise, the value can be in the range 0 ... N-1 if there are N entries in the options table. When

communicating with the user, the options should be listed as 1 ... N; but a zero based representation is likely to be more convenient for the calling routine.

Hiding implementation only functions

Although there appear to be three functions, only one is of importance to other programmers who might want to use this menu package. The "list options" and "handle error" routines are details of the implementation and should be hidden. This can be done in C and C++. These functions can be defined as having filescope – their names are not known to any other part of the program.

The function prototypes can now be defined:

Finalising the design

```
static void ListOptions(const UT_Text options[], int numopts);

static int HandleError(const UT_Text options[], int numopts);

int UT_MenuSelect(const UT_Text prompt, const UT_Text
options[],
    int numopts, int listopts);
```

The two static functions are completely private to the implementation. Other programmers never see these functions, so their names don't have to identify them as belonging to this package of utility functions. Function `MenuSelect()` and the `Text` (character array) data type are seen by other programmers so their finalised names make clear that they belong to this package.

Implementation

Two files are needed. The "header" file describes the functions and types to other parts of the program (and to other programmers). The implementation file contains the function definitions.

The header file has the usual structure with the conditional compilation bracketing to protect against being multiply included:

```
#ifndef __UT__
#define __UT__

const int UT_TXTLENGTH = 60;

typedef char UT_Text[UT_TXTLENGTH];

int UT_MenuSelect(const UT_Text prompt, const UT_Text
options[],
    int numopts, int listopts);

#endif
```

UT.h "header file"

The implementation file will start with #includes on necessary system files and on UT.h. The functions need the stream i/o facilities and the iomanip formatting extras:

```
UT.cp      #include <iostream.h>
implementation #include <iomanip.h>

#include "UT.h"
```

Function ListOptions() simply loops through the option set, printing them with numbering (starting at 1 as required for the benefit of users).

```
static void ListOptions(const UT_Text options[], int numopts)
{
    cout << "Choose from the following options:" << endl;
    for(int i = 1; i <= numopts; i++) {
        cout << setw(4) << i << ": " << options[i-1]
            << endl;
    }
}
```

Function HandleError() is only called if something failed on input. It checks for unrecoverable errors like bad input; returning a 0 "give up" result. Its next step is to "clear" the failure flag. If this is not done, none of the later operations would work.

Once the failure flag has been cleared, characters can be read from the input. These will be the buffered characters containing whatever was typed by the user (it can't start with a digit). If the first character is '?', then the options must be listed.

The cin.get(ch) input style is used instead of cin >> ch because we need to pick up the "whitespace" character '\n' that marks the end of the line. We could only use cin >> ch if we changed the "skip whitespace mode" (as explained in Chapter 9); but the mode would have to be reset afterwards. So, here it is easier to use the get() function.

```
static int HandleError(const UT_Text options[], int numopts)
{
    if(cin.eof() || cin.bad())
        return 0;

    cin.clear();
    char ch;
    cin.get(ch);
    if(ch == '?')
        ListOptions(options, numopts);
    else
        cout << "Illegal input, discarded" << endl;

    while(ch != '\n')
        cin.get(ch);
}
```

```

    return 1;
}

```

Function `MenuSelect()` prints the prompt and possibly gets the options listed. It then uses a `do ... while` loop construct to get data. This type of loop is appropriate here because we must get an input, so a loop that must be traversed at least once is appropriate. If we do get bad input, `choice` will be zero at the end of loop body; but zero is not a valid option so that doesn't cause problems.

The value of `choice` is converted back to a 0 ... N-1 range in the `return` statement.

```

int UT_MenuSelect(const UT_Text prompt, const UT_Text
options[],
    int numopts, int listopts)
{
    cout <<      prompt << endl;
    if(listopts)
        ListOptions(options, numopts);

    int choice = -1;
    do {
        cout << "Enter option number in range 1 to "
            << numopts
            << ", or ? for help" << endl;
        cin >> choice;

        if(!cin.good()) {
            if(!HandleError(options, numopts))
                return -1;
            else continue;
        }

        if(! ((choice >= 1) && (choice <= numopts))) {
            cout << choice <<
                " is not a valid option number,"
                " type ? to get option list" << endl;
        }
    }
    while (! ((choice >= 1) && (choice <= numopts)));
    return choice - 1;
}

```

A simple test program

Naturally, the code has to be checked out. A test program has to be constructed and linked with the compiled `UT.o` code. The test program will `#include` the `UT.h` header file and will define an array of `UT_Text` data elements that are initialized to the possible menu choices:

```

#include <stdlib.h>
#include <iostream.h>
#include "UT.h"

static UT_Text choices[] = {
    "Sex",
    "Drugs",
    "Rock and Roll"
};

int main()
{
    int choice = UT_MenuSelect("Lifestyle?", choices, 3, 0);

    switch(choice) {
case 0:
        cout << "Yes, our most popular line" << endl;
        break;
case 1:
        cout << "Not the best choice" << endl;
        break;
case 2:
        cout << "Are you sure you wouldn't prefer "
                "the first option?" << endl;
        break;
    }

    return EXIT_SUCCESS;
}

```

When the code has been tested, the UT files can be put aside for use in other programs that need menus.

12.4 PICK THE KEYWORD

Although not quite as common as menu selection, many programs require users to enter keyword commands. The program will have an array of keywords for which it has associated action routines. The user enters a word, the program searches the table to find the word and then uses the index to select the action. If you haven't encountered this style of input elsewhere, you will probably have met it in one of the text oriented adventure games where you enter commands like "Go North" or "Pick up the box".

These systems generally allow the user to abbreviate commands. If the user simply types a couple of letters then this is acceptable provided that these form a the start of a unique keyword. If the user entry does not uniquely identify a keyword, the program can either reject the input and reprompt or, better, can list the choices that start with the string entered by the user.

The function `PickKeyWord()` will need to be given:

- 1 an initial prompt;
- 2 an array with the keywords;
- 3 an integer specifying the number of keyword choices.

The code will be along the following lines:

```

print prompt
loop
    read input
    search array for an exactly matched keyword
    if find match
        return index of keyword
    search array for partial matches
    if none,
        (maybe) warn user that there is no match
        start loop again!
    if single partial match,
        (maybe) identify matched keyword to user
        return index of keyword
    list all partially matching keywords

```

The "search array for ..." steps are obvious candidates for being promoted into separate auxiliary functions. The search for an exact match can use the `strcmp()` function from the string library to compare the data entered with each of the possible keywords. A partial match can be found using `strncmp()`, the version of the function that only checks a specified number of characters; `strncmp()` can be called asking it to compare just the number of characters in the partial word entered by the user.

Design, second iteration

Partial matching gets used twice. First, a search is made to find the number of keywords that start with the characters entered by the user. A second search may then get made to print these partial matches. The code could be organized so that a single function could fulfil both roles (an argument would specify whether the partially matched keywords were to be printed). However, it is slightly clearer to have two functions.

The searches through the array of keywords will have to be "linear". The search will start at element 0 and proceed through to element N-1.

Thus, we get to the second more detailed design:

```

FindExactMatch
    for each keyword in array do
        if strcmp() matches keyword & input
            return index
    return -1 failure indicator

CountPartialMatches
    count = 0

```

```

        for each keyword in array do
            if strncmp() matches input & start of keyword
                increment count
        return count

PrintPartialMatches
    for each keyword in array do
        if strncmp() matches input & start of keyword
            print keyword

PickKeyWord
    print prompt
    loop
        read input

        mm = FindExactMatch(...)
        if(mm>=0)
            return mm

        partial_count = CountPartialMatches(...)

        if partial_count == 0,
            (maybe) warn user that there is no match
            start loop again!

        if partial_count == 1,
            (maybe) identify matched keyword to user
            return index of keyword ???

    PrintPartialMatches(...)

```

The sketch outline for the main `PickKeyWord()` routine reveals a problem with the existing `CountPartialMatches()` function. We really need more than just a count. We need a count and an index. Maybe `CountPartialMatches()` could take an extra output argument (`int&` – integer reference) that would be set to contain the index of the (last) of the matching keywords.

Design, third iteration

We have to decide what "keywords" are, and what the prompt argument should be. The prompt could be made a `UT_Text`, the same "array of ≈60 characters" used in the `MenuSelect()` example. The keywords could also be `UT_Texts` but given that most words are less than 12 characters long, an allocation of 60 is overly generous. Maybe a new character array type should be used

```

const int UT_WRDLENGTH = 15;

typedef char UT_WORD[UT_WRDLENGTH ];

```

The `PickKeyWord()` function might as well become another "utility" function and so can share the `UT_` prefix.

As in the case of `MenuSelect()`, although there four functions, only one is of importance to other programmers who might want to use this keyword matching package. The auxiliary `FindExactMatch()` and related functions are details of the implementation and should be hidden. As before, these functions can be defined as having file scope – their names are not known to any other part of the program.

The function prototypes can now be defined:

Finalising the design

```
static int FindExactMatch(const UT_Word keywords[], int nkeys,
                        const UT_Word input);

static int CountPartialMatches(const UT_Word keywords[], int
                             nkeys,
                             const UT_Word input, int& lastmatch);

static void PrintPartialMatches(const UT_Word keywords[],
                               int nkeys, const UT_Word input);

int UT_PickKeyWord(const UT_Text prompt,
                  const UT_Word keywords[], int nkeys);
```

These functions can go in the same `UT.cp` file as the `MenuSelect()` group of functions.

Implementation

The header file `UT.h` has now to include additional declarations:

```
const int UT_TXTLENGTH = 60;
const int UT_WRDLENGTH = 15;

typedef char UT_Word[UT_WRDLENGTH];
typedef char UT_Text[UT_TXTLENGTH];

int UT_MenuSelect(const UT_Text prompt, const UT_Text
                 options[],
                 int numopts, int listopts);
int UT_PickKeyWord(const UT_Text prompt,
                  const UT_Word keywords[], int nkeys);
```

The implementation in `UT.cp` is straightforward. Function `FindExactMatch()` has a simple loop with the call to `strcmp()`. Function `strcmp()` returns 0 if the two strings that it is given are equal.

```
static int FindExactMatch(const UT_Word keywords[], int nkeys,
                        const UT_Word input)
{
    for(int i = 0; i < nkeys; i++)
```

```

        if(0 == strcmp(input, keywords[i]))
            return i;
    return -1;
}

```

Functions `CountPartialMatches()` and `PrintPartialMatches()` both use `strlen()` to get the number of characters in the user input and `strncmp()` to compare substrings of the specified length. The count routine has its reference argument `lastmatch` that it can use to return the index of the matched string.

```

int CountPartialMatches(const UT_Word keywords[], int nkeys,
    const UT_Word input, int& lastmatch)
{
    int count = 0;
    int len = strlen(input);
    lastmatch = -1;
    for(int i = 0; i < nkeys; i++)
        if(0 == strncmp(input, keywords[i], len)) {
            count++;
            lastmatch = i;
        }
    return count;
}

void PrintPartialMatches(const UT_Word keywords[], int nkeys,
    const UT_Word input)
{
    cout << "Possible matching keywords are:" << endl;
    int len = strlen(input);
    for(int i = 0; i < nkeys; i++)
        if(0 == strncmp(input, keywords[i], len))
            cout << keywords[i] << endl;
}

```

The only point of note in the `PickKeyWord()` function is the "forever" loop – `for(;;) { ... }`. We don't want the function to return until the user has entered a valid keyword.

```

int UT_PickKeyWord(const UT_Text prompt,
    const UT_Word keywords[], int nkeys)
{
    cout << prompt;
    for(;;) {
        UT_Word input;
        cin >> input;

        int mm = FindExactMatch(keywords, nkeys, input);
        if(mm >= 0)
            return mm;
    }
}

```



```

        int match;
        int partial_count =
            CountPartialMatches(keywords,
                                nkeys, input, match);

        if(partial_count == 0) {
            cout << "There are no keywords like "
                << input << endl;
            continue;
        }

        if(partial_count == 1) {
            cout << "i.e. " << keywords[match] << endl;
            return match;
        }

        PrintPartialMatches(keywords, nkeys, input);
    }
}

```

A simple test program

As usual, we need a simple test program to exercise the functions just coded:

```

#include <stdlib.h>
#include <iostream.h>
#include "UT.h"

static UT_Word commands[] = {
    "Quit",
    "North",
    "South",
    "East",
    "West",
    "NW",
    "NE",
    "SE",
    "SW"
};

const int numkeys = sizeof(commands)/sizeof(UT_Word);

int main()
{
    cout << "You have dropped into a maze" << endl;

    int quit = 0;

    while(!quit) {
        cout << "From this room, passages lead "

```

```

                                "in all directions" << endl;
    int choice = UT_PickKeyWord(">", commands, numkeys);
    switch(choice) {
case 1:
case 3:                        cout << "You struggle along a steep and"
                                "narrow passage" << endl;
                                break;
case 2:
case 5:                        cout << "You move quickly through a rocky"
                                " passage" << endl;
                                break;
default:
                                cout << "You walk down a slippery, dark,"
                                "damp, passage" << endl;
                                break;
case 0:
                                quit = 1;
                                break;
                                }
    }
    cout << "Chicken, you haven't explored the full maze"
                                << endl;
    return EXIT_SUCCESS;
}

```

12.5 HANGMAN

You must know the rules of this one. One player selects a word and indicates the number of letters, the other player guesses letters. When the second player guesses a letter that is in the word, the first player indicates all occurrences of the letter. If the second player guesses wrongly, the first player adds a little more to a cartoon of a corpse hung from a gibbet. The second player wins if all letters in the word are guessed. The first player wins if the cartoon is completed while some letters are still not matched.

In this version, the program takes the role of player one. The program contains a large word list from which it selects a word for the current round. It generates a display showing the number of letters, then loops processing letters entered by the user until either the word is guessed or the cartoon is complete. This version of the program is to use the curses functions, and to produce a display like that shown in Figure 12.6.

Specification

Implement the hangman program; use the curses package for display and interaction.

Thus the program is going to be built of a number of parts:

- Hangm.cp
This file will contain the main() function and all the other functions defined for the program.
- vocab.cp
The file with the words.
- UT.h
The header file with the typedef defining a "word" as a character array.
- CG.cp and CG.h
These files contain the curses cursor graphics package.

*First iteration
through design of
code*

The program will be something like:

```

initialize random number generator and cursor graphics
loop
    pick a word
    show word as "*****" giving indication of length
    loop
        get user to guess character
        check word, where character matches; change
            displayed word to show matches e.g. "***a*"
        if didn't get any match
            draw one more part of cartoon
        check for loop termination

    report final win/loss status

    ask if another game,

tidy up

```

Given the relative complexity of the code, this is going to require breaking down into many separate functions. Further, the implementation should be phased with parts of the code made to work while before other parts get implemented.

The main line shown above is too complex. It should be simplified by "abstracting out" all details of the inner loop that plays the game. This reduces it to:

```

main()
    Initialize()
    do
        PlayGame()
    while AnotherGame()
    TidyUp()

```

This structure should be elaborated with a "reduced" version of `PlayGame()`. The reduced version would just pick a random word, show it as "****", wait for a few second, and then show the actual word. This reduced version allows the basic framework of the program to be completed and tested.

Following these decisions, the preliminary design for the code becomes:

*Second iteration,
design of a simplified
program*

```
AnotherGame()
    prompt user for a yes/no reply
    return true if user enters 'y'

PlayGame()
    get curses window displayed
    pick random word
    display word as "****" at some point in window

    !!!delay
    !!!display actual word
```

The two steps marked as !!! are only for this partial implementation and will be replaced later.

Function `AnotherGame()` can be coded using the CG functions like `CG_Prompt()` and `CG_GetChar()`, so it doesn't require further decomposition into simpler functions. However, a number of additional functions will be required to implement even the partial `PlayGame()`.

These functions obviously include a "display word" function and a "pick random word" function. There is also the issue of how the program should record the details of the word to be guessed and the current state of the users guess.

One approach would be to use two "words" – `gGuess` and `gWord`, and a couple of integer counters. The "pick random word" function could select a word from the vocabulary, record its length in one of the integer counters, copy the chose word into `gWord` and fill `gGuess` with the right number of '*'s. The second integer counter would record the number of characters matched. This could be used later when implementing code to check whether the word has been guessed completely.

So, we seem to have:

```
PickRandomWord
    pick random number i in range 0 ... N-1
        where N is number of word in vocab
    record length of vocab[i] in length
    copy vocab[i] into gWord
    fill gGuess with '*'s

ShowGuess
    move cursor to suitable point on screen
    copy characters from gGuess to screen
```

```
ShowWord
    move cursor to suitable point on screen
    copy characters from gWord to screen
```

***Final design iteration
for simplified
program***

The functions identified at this point are sufficiently simple that coding should be straightforward. So, the design of this part can be finished off by resolving outstanding issues of data organization and deciding on function prototypes.

The data consist of the four variables (two words and two integer counters) that can be made globals. If these data are global, then the prototypes for the functions identified so far are:

```
int AnotherGame(void);
void Initialize(void);
void PickWord(void);
void PlayGame(void);
void ShowGuess(void);
void ShowWord(void);

int main()
```

External declarations

The code in the main file Hangm.cp needs to reference the vocabulary array and word count that are defined in the separate vocab.cp file. This is achieved by having "external declarations" in Hangm.cp. An external declaration specifies the type and name of a variable; it is basically a statement to the compiler *"This variable is defined in some other file. Generate code using it. Leave it to the linking loader to find the variables and fill in the correct addresses in the generated code."*

Implementation of simplified program

The project has include the files Hangm.cp (main program etc), vocab.cp (array with words), and CG.cp (the curses functions); the header files CG.h and UT.h are also needed in the same directory (UT.h is only being used for the definition of the UT_Word type).

The file Hangm.cp will start with the #includes of the standard header files and definitions of global variables. The program needs to use random numbers; stdlib provides the rand() and srand() functions. As explained in 10.10, a sensible way of "seeding" the random number generator is to use a value from the system's clock. The clock functions vary between IDE's. On Symantec, the easiest function to use is TickCount() whose prototype is in events.h; in the Borland system, either the function time() or clock() might be used, their prototypes are in time.h. The header ctype is #included although it isn't required in the simplified program.

```
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
// change events.h to time.h for Borland
```

```
#include <events.h>
#include "CG.h"
#include "UT.h"
```

Here are the extern declarations naming the variables defined in the separate vocab.cp file:

```
extern UT_Word vocab[];
extern int numwords;
```

and these are the global variables:

```
UT_Word gGuess;
UT_Word gWord;
int gLength;
int gMatched;
```

The functions Initialize(), AnotherGame(), ShowGuess() and ShowWord() are all simple:

```
void Initialize(void)
{
    CG_Initialize();
    // change to srand(time(NULL)); on Borland
    srand(TickCount());
}

int AnotherGame(void)
{
    CG_Prompt("Another Game?");
    char ch = CG_GetChar();
    CG_Prompt(" ");
    return ((ch == 'y') || (ch == 'Y'));
}

void ShowGuess(void)
{
    const xGuess = 6;
    const yGuess = 4;

    CG_MoveCursor(xGuess, yGuess);
    for(int i=0; i<gLength; i++)
        CG_PutCharacter(gGuess[i]);
}

void ShowWord(void)
{
    const xGuess = 6;
    const yGuess = 4;
```

```

        CG_MoveCursor(xGuess, yGuess);
        for(int i=0; i<gLength; i++)
            CG_PutCharacter(gWord[i]);
    }

```

Function `PickWord()` uses the random number generator to pick a number that is converted to an appropriate range by taking its value modulo the number of possible words. The chosen word is then copied and the guess word is filled with '*'s as required.

```

void PickWord(void)
{
    int choice = rand() % numwords;
    gLength = strlen(vocab[choice]);
    for(int i = 0; i < gLength; i++) {
        gWord[i] = vocab[choice][i];
        gGuess[i] = '*';
    }
    gWord[gLength] = gGuess[gLength] = '\0';
    gMatched = 0;
}

```

Function `PlayGame()` is supposed to look after a complete game. Of course in this simplified version, it just refreshes the curses window, picks a word, shows it as stars, waits, and shows the word:

```

void PlayGame(void)
{
    CG_FrameWindow();
    PickWord();
    ShowGuess();

    CG_Delay(2);

    ShowWord();

    return;
}

```

Function `main()` is complete, it shouldn't need to be changed later. The "Tidy()" routine postulated in the original design has collapsed into the call to reset the curses system, there didn't seem to be any other tidying to do so no need for a separate function.

```

int main()
{
    Initialize();
    do {

```



```
        PlayGame();
    }
    while( AnotherGame());

    CG_Reset();
    return 0;
}
```

The other source file is `vocab.cp`. This should have definitions of the words and the number of words. This file needs to `#include UT.h` to get the declaration of type `UT_Word`. The array `vocab[]` should contain a reasonable number of common words from a standard dictionary such as the Oxford English Dictionary:

```
#include "UT.h"

UT_Word vocab[] = {
    "vireo",
    "inoculum",
    "ossuary",
    "...",
    "thurible",
    "jellaba",
    "whimbrel",
    "gavotte",
    "clearcole",
    "theandric"
};

int numwords = sizeof(vocab)/ sizeof(UT_Word);
```

Although the vocabulary and number of words are effectively constant, they should not be defined as `const`. In C++, `const` carries the implication of filescope and the linking loader might not be able to match these names with the external declarations in the `Hangm.cp` file.

The code as shown should be executable. It allows testing of the basic structure and verifies that words are being picked randomly.

Designing the code to handle the user's guesses

The next step of a phased implementation would be to handle the user's guesses (without exacting any penalties for erroneous guesses).

The `PlayGame()` function needs to be expanded to contain a loop in which the user enters a character, the character is checked to determine whether any additional letters have been matched, and counts and displays are updated.

This loop should terminate when all letters have been guessed. After the loop finishes a "You won" message can be displayed.

The '*'s in the gGuess word can be changed to appropriate characters as they are matched. This will make it easy to display a partially matched word as the ShowGuess() function can be called after each change is complete.

Some additional functions are needed. Function GetGuessedCharacter() should prompt the user for a character and get input. If the character is a letter, it should be converted to lower case and returned. Otherwise GetGuessedCharacter() should just loop, repeating the prompt. Function CheckCharacter() should compare the character with each letter in the gWord word; if they match, the character should overwrite the '*' in gGuess and a count of matches should be incremented. There will also be a ShowResult() function that can display a message and cause a short pause.

```

GetGuessed charater
loop
    prompt for character
    get input
    if letter
        convert to lower case and return

CheckCharacter ch
count = 0;
for each letter in gWord
    if letter == ch
        count++
    set character in gGuess

Show Result
display message
delay a couple of seconds

PlayGame()
CG_FrameWindow()
PickWord()
ShowGuess()

gameover = false

while not gameover
    Get guessed character
    matched = CheckCharacter
    if(matched > 0)
        ShowGuess
        gMatched += matched
    gameover = (gMatched == gLength)

Show Result "You win"

```

The prototypes for the additional functions are:

```
int CheckChar(char ch);
```

```
char GetGuessedChar();  
void ShowResult(const char msg[]);
```

Implementation of code for user input

The code implementing these additional and modified function functions is straightforward. Function `GetGuessedCharacter()` uses a `do ... while` loop to get an alphabetic character. Function `CheckChar()` uses a `for` loop to check the match of letters.

```
char GetGuessedChar()  
{  
    CG_Prompt(" ");  
    CG_Prompt(">");  
    char ch;  
    do  
        ch = CG_GetChar();  
    while (!isalpha(ch));  
    ch = tolower(ch);  
    return ch;  
}  
  
int CheckChar(char ch)  
{  
    int count = 0;  
    for(int i = 0; i < gLength; i++)  
        if((gWord[i] == ch) && (gGuess[i] == '*')) {  
            gGuess[i] = ch;  
            count++;  
        }  
    return count;  
}  
  
void ShowResult(const char msg[])  
{  
    CG_Prompt(msg);  
    CG_Delay(5);  
}
```

The `while` loop in `PlayGame()` terminates when `gameOverMan` is true. Currently, there is only one way that the game will be over – all the letters will have been guessed. But we know that the implementation of the next phase will add other conditions that could mean that the game was over.

```
void PlayGame(void)  
{  
    CG_FrameWindow();  
    PickWord();
```

```

        ShowGuess();

        int count = 0;
        int gameOverMan = 0;

        while(!gameOverMan) {
            char ch = GetGuessedChar();
            int matched = CheckChar(ch);
            if(matched > 0) {
                ShowGuess();
                gMatched += matched;
            }

            gameOverMan = (gMatched == gLength);
        }

        ShowResult("You won");

        return;
    }
}

```

Again, this code is executable and a slightly larger part of the program can be tested.

Designing the code to handle incorrect guesses

The final phase of the implementation would deal with the programs handling of incorrect guesses. Successive incorrect guesses result in display of successive components of the cartoon of the hung man. The game ends if the cartoon is completed.

The cartoon is made up out of parts: the frame, the horizontal beam, a support, the rope, a head, a body, left and right arms, and left and right legs. Each of these ten parts is to be drawn in turn.

The `PlayGame()` function can keep track of the number of incorrect guesses; the same information identifies the next part to be drawn. Selection of the parts can be left to an auxiliary routine – "show cartoon part".

Each part of the cartoon is made up of a number of squares that must be filled in with a particular character. The simplest approach would appear to be to have a little "fill squares" function that gets passed arrays with x, y coordinates, the number of points and the character. This routine could loop using `CG_PutCharacter()` to display a character at each of the positions defined in the arrays given as arguments.

Details of the individual parts are best looked after by separate routines that have their own local arrays defining coordinate data etc.

Thus, this phase of the development will have to deal with the following functions:

```

Fill_squares
loop

```

```

        move cursor to position of next point defined
            by x, y argument arrays
        output character

Show Frame
    call Fill squares passing that function arrays
        defining the points that make up the frame

Show Head
    call Fill squares passing that function arrays
        defining the points that make up the head

...

Show Cartoon part (partnumber)
    switch on partnumber
        call show frame, or show beam, or ...
        as appropriate

```

Function `PlayGame()` requires some further extension. If a character didn't match, the next cartoon part should be drawn and the count of errors should be increased. There is an additional loop termination condition – error count exceeding limit. The final display of the result should distinguish wins from losses.

```

PlayGame()
    CG_FrameWindow()
    PickWord()
    ShowGuess()

    gameover = false
    count = 0;
    while not gameover
        Get guessed character
        matched = CheckCharacter
        if(matched > 0)
            ShowGuess
            gMatched += matched
        else
            ShowCartoonPart
            count++

    gameover = (gMatched == gLength) || (count > limit)

    if (gMatched== gLength)Show Result "You win"
    else
        ShowWord
        Show Result "You lost"

```

Implementation of final part of code

The extra routines that get the cartoon parts drawn are all simple, only a couple of representatives are shown:

```
void FillSquares(char fill, int num, int x[], int y[])
{
    for(int i = 0; i < num; i++) {
        CG_MoveCursor(x[i],y[i]);
        CG_PutCharacter(fill);
    }
}

void ShowBeam(void)
{
    int x[] = { 51, 52, 53, 54, 55, 56, 57, 58 };
    int y[] = { 5, 5, 5, 5, 5, 5, 5, 5 };
    int n = sizeof(x) / sizeof(int);
    FillSquares('=', n, x, y);
}

void ShowHead(void)
{
    int x[] = { 59 };
    int y[] = { 8 };
    int n = sizeof(x) / sizeof(int);
    FillSquares('@', n, x, y);
}

void ShowCartoonPart(int partnum)
{
    switch(partnum) {
case 0:
        ShowFrame();
        break;
case 1:
        ShowBeam();
        break;
case 2:
        ShowSupport();
        break;
...
...
case 9:
        ShowRightLeg();
        break;
    }
}
```

The final version of `PlayGame()` is:

```

void PlayGame(void)
{
    CG_FrameWindow();
    PickWord();
    ShowGuess();

    int count = 0;
    int gameOverMan = 0;

    while(!gameOverMan) {
        char ch = GetGuessedChar();
        int matched = CheckChar(ch);
        if(matched > 0) {
            ShowGuess();
            gMatched += matched;
        }
        else {
            ShowCartoonPart(count);
            count++;
        }
        gameOverMan = (count >= kNMOVES) ||
                      (gMatched == gLength);
    }

    if(gMatched==gLength) ShowResult("You won");
    else {
        ShowWord();
        ShowResult("You lost");
    }
    return;
}

```

(The constant kNMOVES = 10 gets added to the other constants at the start of the file.)

You will find that most of the larger programs that you must write will need to be implemented in phases as was done in this example.

*Phased
implementation
strategy*

12.6 LIFE

Have you been told "You are a nerd – spending too much time on your computer."?

Have you been told "Go out and get a life."?

OK. Please everyone. Get a Life in your computer – Conway's Life.

Conway's Life is not really a computer game. When it was first described in Scientific American it was introduced as a "computer recreation" (Sci. Am. Oct 1970, Feb 1971). The program models a "system" that evolves in accord with specified rules. The idea of the "recreation" part is that you can set up different initial states for this "system" and watch what happens.

The system consists of a two-dimensional array of cells. This array is, in principle, infinite in its dimensions; of course for a computer program you must choose a finite size. Cells contain either nothing, or a "live organism". Each cell (array element) has eight neighbors – three in the row above, three in the row below, a left neighbor and a right neighbor.

Evolution proceeds one generation at a time. The rules of the recreation specify what happens to the contents of each cell at a change of generation. The standard rules are:

- 1 An organism in a cell "dies of loneliness" if it has less than two neighbors.
- 2 An organism "dies of overcrowding" if it has four or more neighbors.
- 3 An organism survives for another generation if it has two or three neighbors.
- 4 An empty cell is filled with a new live organism if it had exactly three neighbors.

These deaths and births are coordinated. They happen simultaneously (this affects how the simulation may be programmed).

The recreation is started by assigning an initial population of organisms to selected cells. Most distributions are uninteresting. The majority of the population quickly die out leaving little groups of survivors that cling tenaciously to life. But some patterns have interesting behaviours. Figure 12.7 shows three examples – "Cheshire Cat", "Glider", and "Tumbler"; there are others, like "glider guns" and "glider eaters" that are illustrated in the second of the two Scientific American articles. A "Glider" moves slowly down and rightwards; a "Cheshire Cat fades away to leave just a grin".

Cheshire Cat	Tumbler
.....
....*.*.....**.*.....
....****.....**.*.....
....*.....**.*.....
....*.*.*.....*.*.*.....
....*.....**.*.*.....
....****.....**.*.....
.....
.....*.*.....
.....**.....
.....*.....
.....
.....
.....

Glider

Figure 12.7 Configurations for Conway's Life

Specification

Implement a version of Conway's Life recreation. The program should use the curses package for display. It should start with some standard patterns present in the array. The program should model a small number of generational turns, then ask the user whether to continue or quit. The modelling and prompting process continues until the user enters a quit command.

Design

This program will again have multiple source files – Life.cp and CG.cp (plus the CG.h header). All the new code goes in the Life.cp file. The code that scans the array and sorts out which cells are "live" at the next generation will have some points in common with the "heat diffusion" example. *First Iteration*

The overall structure of the program will be something like:

```
Initialize curses display and "life arrays"
Place a few standard patterns in cells in the array
display starting configuration
loop
    inner-loop repeated a small number of times (≈5)
        work out state for next generation
        display new state
    ask user whether to continue or quit
tidy up
```

As always, some steps are obvious candidates for being promoted into separate functions. These "obvious" functions include an initialization function, a to set an initial configuration, a display function, a "run" function that looks after the loops, and a function to step forward one generation.

The "life arrays" will be similar to the arrays used in the heat diffusion example. As in that example, there have to be two arrays (at least while executing the function that computes the new state). One array contains the current state data; these data have to be analyzed to allow new data to be stored in the second array. Here the arrays would be character arrays (really, one could reduce them to "bit arrays" but that is a lot harder!). A space character could represent an empty cell, a '*' or '#' could mark a live cell.

This example can illustrate the use of a "three dimensional array"! We need two nxm arrays (current generation, next generation). We can define these as a single data aggregate:

```
const int kROWS = 20;
const int kCOLS = 50;

char gWorlds[2][kROWS][kCOLS];
```

We can use the two [kROWS][kCOLS] subarrays alternately. We can start with the initial life generation in subarray gWorlds[0], and fill in gWorlds[1] with details for the next generation. Subarray gWorlds[1] becomes the current generation that gets displayed. At the next step, subarray gWorlds[0] is filled in using the current gWorlds[1] data. Then gWorlds[0] is the current generation and is displayed. As each step is performed, the roles of the two subarrays switch. All the separate functions would share the global gWorlds data aggregate.

*Second iteration
through design
process*

Each function has to be considered in more detail, possibly additional auxiliary functions will be identified. The initial set of functions become:

```

Initialize
    fill both gWorld[] subarrays with spaces
        (representing empty cells)
    initialize curses routines

Set starting configuration
    place some standard patterns
        Not specified, so what patterns?
        maybe a glider at one point in array, cheshire cat
        somewhere else.

Display state
    double loop through "current gWorld" using
        curses character display routines to plot spaces
        and stars

Run
    Display current state
    loop
        ask user whether to quit,
            and break loop if quit command entered
    loop ≈ 5 times
        step one generation forward
        display new state

Step
    ?

main()
    Initialize
    Set Starting Configuration
    Run
    reset curses
  
```

This second pass through the program design revealed a minor problem. The specification requires "some standard patterns present in the array"; but doesn't say what patterns! It is not unusual for specifications to turn out to be incomplete. Usually, the designers have to go back to the people who commissioned the program to get more

details specified. In a simple case, like this, the designers can make the decisions – so here "a few standard patterns" means a glider and something else.

There are still major gaps in the design, so it is still too early to start coding.

The "standard patterns" can be placed in much the same way as the "cartoon components" were added in the Hangman example. A routine, e.g. "set glider", can have a set of x, y points that it sets to add a glider. As more than one glider might be required, this routine should take a pair of x, y values as an origin and set cells relative to the origin.

Some of the patterns are large, and if the origin is poorly chosen it would be easy to try to set points outside the bounds of the array. Just as in the heat diffusion example where we needed a `TemperatureAt(row, column)` function that checked the row and column values, we will here need a `SetPoint(row, column)` that sets a cell to live provided that the row and column are valid.

These additions make the "set starting configuration" function into the following group of functions:

```

set point (col, row)
    if col and row valid
        add live cell to gWorld[0][row][col]

set glider (xorigin, yorigin)
    have arrays of x, y values
    for each entry in array
        set point(x+xorigin, y + yorigin)

Set starting configuration
    set glider ..., ...
    set cc ..., ...
    ...

```

(Note, x corresponds to column, y corresponds to row so if set point is to be called with an x, y pair then it must define the order of arguments as column, row.)

The other main gap in the second design outline was function `Step()`. Its basic structure can be obtained from the problem description. The entire new generation array must be filled, so we will need a double loop (all the rows by all the columns). The number of neighbors of each cell in current `gWorld` must be calculated and used in accord with the rules to determine whether the corresponding cell is 'live' in the next `gWorld`. The rules can actually be simplified. If the neighbor count is not 2 or 3, the cell is empty in the next generation (0, 1 imply loneliness, 4 or more imply overcrowding). A cell with 3 neighbors will be live at next generation (either 3 neighbors and survive rule, or 3 neighbors and birth rule). A cell with 2 neighbors is only live if the current cell is live. Using this reformulated version of the rules, the `Step()` function becomes:

```

Step
    identify which gWorld is to be filled in

```

```

    for each row
        for each col
            nbrs = Count live Neighbors of cell[row][col]
                in current gWorld
            switch(nbrs)
                2 new cell state = current cell state
                3 new cell state = live
                default new cell state = empty
    note gWorld just filled in as "current gWorld"

```

The `Step()` function and the display function both need to know which of the `gWorld`s is current (i.e. is it the subarray `gWorld[0]` or `gWorld[1]`). This information should be held in another global variable `gCurrent`. The `Step()` function has to use the value of this variable to determine which array to fill in for the next generation, and needs to update the value of `gCurrent` once the new generation has been recorded.

We also need a "count live neighbors" function. This will be a bit like the "Average of neighbors" function in the Heat Diffusion example. It will use a double loop to run through nine elements (a cell and its eight neighbors) and correct for the cell's own state:

```

Count Neighbors (row, col)
    count = -Live(row,col)
    for r = row - 1 to row + 1
        for c = col - 1 to col + 1
            cout += Live(r,c)

```

Just as the Heat Diffusion example needed a `TemperatureAt(row, col)` function that checked for out of range row and col, here we need a `Live(row, col)` function. What happens at the boundaries in a finite version of Life? One possibility is to treat the non-existent cells (e.g. cells in `row == -1`) as being always empty; this is the version that will be coded here. An alternative implementation of Life "wraps" the boundaries. If you want a cell in `row == kROWS` (i.e. you have gone off the bottom of the array), you use `row 0`. If you want a cell in `row == -1`, (i.e. you have gone off the top of the array), you use `row == kROWS-1` (the bottommost row). If you "wrap" the world like this, then gliders moving off the bottom of the array reappear at the top.

The code for functions like `Live()` will be simple. It can use a `Get()` function to access an element in the current `gWorld` (the code of `Get()` determines whether we are "wrapping" the `gWorld` or just treating outside areas as empty). `Live()` will simply check whether the character returned by `Get()` signifies a 'live cell' or an 'empty cell'. So, no further decomposition is required.

*Third iteration
through design
process*

The function prototypes and shared data can now be specified. The program needs to have as globals an integer `gCurrent` that identifies which `gWorld` subarray is current, and, as previously mentioned, the `gWorld` data aggregate itself.

The functions are:

```

char Get(int row, int col);

```

```
int CountNbrs(int row, int col);

void DisplayState(void);

void Initialize(void);

int Live(int row, int col);

void Run(void);

void SetPoint(int col, int row);

void SetGlider(int x0, int y0);

void SetStartConfig(void);

void Step(void);

int main();
```

Implementation

For the most part, the implementation code is straightforward. The file starts with the `#includes`; in this case just `ctype.h` (need `tolower()` function when checking whether user enters "quit command") and the CG header for the curses functions. After the `#includes`, we get the global variables (the three dimensional `gWorld` aggregate etc) and the constants.

```
#include <ctype.h>
#include "CG.h"

const int kROWS = 20;
const int kCOLS = 50;
const int kSTEPSPERCYCLE = 5;

char gWorlds[2][kROWS][kCOLS];

int gCurrent = 0;

const char EMPTY = ' ';
const char LIVE = '*';
```

Functions `Initialize()` and `DisplayState()` both use similar double loops to work through all array elements. Note the `c+1`, `r+1` in the call to `CG_PutCharacter()`; the arrays are zero based but the display system is one based (top left corner is square 1, 1 in display).

```

void Initialize(void)
{
    for(int w=0; w < 2; w++)
        for(int r = 0; r < kROWS; r++)
            for(int c = 0; c < kCOLS; c++)
                gWorlds[w][kROWS][kCOLS] =
                    EMPTY;

    CG_Initialize();
}

void DisplayState(void)
{
    for(int r = 0; r < kROWS; r++)
        for(int c = 0; c < kCOLS; c++)
            CG_PutCharacter(gWorlds[gCurrent][r][c],
                            c+1, r+1 );
}

```

Function `Get()` looks indexes into the current `gWorld` subarray. If "wrapping" was needed, a row value less than 0 would have `kROWS` added before accessing the array; similar modifications in the other cases.

Function `Live()` is another very small function; quite justified, its role is different from `Get()` even if it is the only function here that uses `Get()` they shouldn't be combined.

```

char Get(int row, int col)
{
    if(row < 0)
        return EMPTY;
    if(row >= kROWS)
        return EMPTY;
    if(col < 0)
        return EMPTY;
    if(col >= kCOLS)
        return EMPTY;

    return gWorlds[gCurrent][row][col];
}

int Live(int row, int col)
{
    return (LIVE == Get(row,col));
}

```

Function `CountNbrs()` is very similar to the `AverageOfNeighbors()` function shown earlier:

```

int CountNbrs(int row, int col)

```

```

{
    int count = -Live(row,col); // Don't count self!
    for(int r = row - 1; r <= row + 1; r++)
        for(int c = col - 1; c <= col + 1; c++)
            count += Live(r,c);
    return count;
}

```

Function `Step()` uses a "tricky" way of identifying which `gWorld` subarray should be updated. Logically, if the current `gWorld` is subarray `[0]`, then subarray `gWorld[1]` should be changed; but if the current `gWorld` is `[1]` then it is `gWorld[0]` that gets changed. Check the code and convince yourself that the funny modulo arithmetic achieves this:

```

void Step(void)
{
    int Other = (gCurrent + 1) % 2;
    for(int r = 0; r < kROWS; r++)
        for(int c = 0; c < kCOLS; c++) {
            int nbrs = CountNbrs(r,c);
            switch(nbrs) {
case 2:
                    gWorlds[Other][r][c] =
                        gWorlds[gCurrent][r][c];
                    break;
case 3:
                    gWorlds[Other][r][c] = LIVE;
                    break;
default:
                    gWorlds[Other][r][c] = EMPTY;
            }
        }
    gCurrent = Other;
}

```

Function `SetPoint()` makes certain that it only accesses valid array elements. This function might need to be changed if "wrapping" were required.

```

void SetPoint(int col, int row)
{
    if(row < 0)
        return;
    if(row >= kROWS)
        return;
    if(col < 0)
        return;
    if(col >= kCOLS)
        return;

    gWorlds[gCurrent][row][col] = LIVE;
}

```

```
}
```

Function `SetGlider()` uses the same approach as the functions for drawing cartoon parts in the Hangman example:

```
void SetGlider(int x0, int y0)
{
    int x[] = { 1, 2, 3, 1, 2 };
    int y[] = { 1, 2, 2, 3, 3 };
    int n = sizeof(x) / sizeof(int);
    for(int i = 0; i < n; i++)
        SetPoint(x0 + x[i], y0 + y[i]);
}

void SetStartConfig(void)
{
    SetGlider(4,4);
    ...
}
```

Function `Run()` handles the user controlled repetition and the inner loop advancing a few generation steps:

```
void Run(void)
{
    char ch;
    DisplayState();
    int quit = 0;
    for(;;) {
        CG_Prompt("Cycle (C) or Quit (Q)?");
        ch = CG_GetChar();
        ch = tolower(ch);

        if(ch == 'q')
            break;
        CG_Prompt(" ");

        for(int i = 0; i < kSTEPSPERCYCLE; i++) {
            Step();
            DisplayState();
        }
    }
}
```

As usual, `main()` should be basically a sequence of function calls:

```
int main()
{
    Initialize();
    SetStartConfig();
}
```



```
Run();
CG_Reset();
return 0;
}
```

EXERCISES

- 1 Use the `curses` package to implement a (fast) clock:

The "minutes" hand should advance every second, and the "hours" hand every minute.

- 2 In his novel "The Day of the Triffids", John Wyndham invented triffids – hybrid creatures that are part plant, part carnivore. Triffids can't see, but they are sensitive to sound and vibrations. They can't move fast, but they can move. They tend to move toward sources of sounds – sources such as animals. If a triffid gets close enough to an animal, it can use a flexible vine- like appendage to first kill the animal with a fast acting poison, and then draw nutrients from the body of the animal

At one point in the story, there is a blind man in a field with a triffid. The man is lost stumbling around randomly. The triffid moves (at less than one fifth of the speed of the man), but it moves purposively toward the source of vibrations – the triffid is hunting its lunch.

Provide a graphical simulation using the curses library.

A diagram showing a rectangular region. The top boundary is a dashed line, and the bottom boundary is a solid line. Inside the region, the letter 'T' is located to the left of the letter 'm'.

- 3 Somewhere, on one of the old floppies lost in your room is a copy of a 1980s computer game "Doctor Who and the Daleks". Don't bother to search it out. You can now write your own version.

The game consist of a series of rounds with the human player controlling the action of Dr. Who, and the program controlling some number of Daleks. A round is comprised on many turns. The human player and the program take alternate turns; on its turn the program

will move all the Daleks. Things (Daleks and Dr) can move one square vertically, horizontally, or diagonally. Nothing can move out of the playing area. The display is redrawn after each turn. Dr. Who's position is marked by a W, the Daleks are Ds, and wrecked Daleks are #s.

```

+-----+
|               |
|               |
|      D        |
|               |
|      W  #      |
|               |
|               |
|               |
|               |
|               |
+-----+
Command>

```

The human player can enter commands that move Dr. Who (these are one character commands, common choices of letter and directions are 'q' NW, 'w' N, 'e' NE, 'a' W, ... with North being up the screen).

If a Dalek moves onto the square where Dr. Who is located, the game ends. If a Dalek moves onto a square that is occupied, it is wrecked and does not participate further in the round. The wreckage remains for the rest of that round; other Daleks may crash into wreckage and destroy themselves. Dr. Who can not move onto a square with wreckage.

The Daleks' moves are determined algorithmically. Each Dalek will move toward the current position of Dr. Who. The move will be vertical, horizontal, or diagonal; it is chosen to minimize the remaining distance from the Dalek's new position to that of Dr. Who.

A round of the game starts with Dr. Who and some number of Daleks being randomly placed on a clear playing area. (There are no constraints, Dr. Who may discover that he is totally surrounded by Daleks; he may even start dead if a Dalek gets placed on his square.)

The human player's winning strategy is to manoeuvre Dr. Who so as to cause the maximum number of collisions among Daleks.

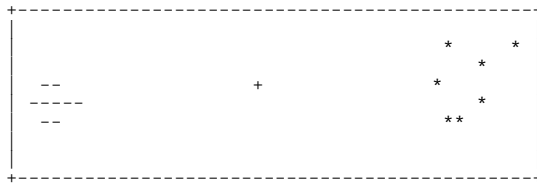
The player has two other options. Dr. Who may be teleported to some random location (command 't') so allowing escape from converging Daleks. There are no guarantees of safe arrival; he may rematerialise on wreckage (and be killed), on a Dalek (and again be killed), or next to a Dalek (and be killed when the Dalek moves). Once per round, Dr. Who may use a "sonic screwdriver" that destroys Daleks in the immediate vicinity (immediate? radius 2, radius 3, depends how generous you feel).

If Dr. Who survives a round in which all Daleks are destroyed, a new round starts. Each new round increases the number of Daleks up to a maximum. Initially, there are three Daleks, the maximum depends on the size of the playing area but should be around 20.

4. The professor who commissioned the π -cannon is disappointed at the rate of convergence and believes that the cannoneers are doing something wrong.

Combine the code from the π -cannon example with the curses package to produce a visual display version that the professor may watch:

The program should show the cannon ball moving from gun barrel to point of impact (getting the successive x positions is easy, you need to do a bit of trigonometry to get the y positions).



5. Modify Conway's Life so that the playing area is "wrapped" (i.e. gliders moving off the top of the play area reappear at the bottom etc.)
6. Implement a version of the "Mastermind" game using the curses package.

In "Mastermind", the program picks a four digit (or four letter code), e.g. EAHM. The human player tries to identify the code in the minimum number of guesses. The player enters a guess as four digits or letters. The program indicates 1) how many characters are both correct and in the correct place, and 2) how many characters are correct but in the wrong place (these reports do not identify which characters are the correct ones). The player is to use these clues to help make subsequent guesses.

A display shows the history of guesses and resulting clues:

Guess#	Correct	Guess	Misplaced letter
1		ABCD	*
2		WXYZ	
3	*	EFGH	*
4	**	EAIJ	

Rules vary as to whether you can have repeated characters. Make up your own rules.

7. Build a better curses.

The "refresh" rate when drawing to a window is poor in the curses package given in the text.

You can make things faster by arranging that the curses package make use of a character array `screen[kHEIGHT][kWIDTH]`. This is initialized to contain null characters. When characters are drawn, a check is made against this array. If the array element corresponding to the character position has a value that differs from the required character, then the required character is drawn and stored in the array. However, if the array indicates that the character on the screen is already the required character then no drawing is necessary.

8. Combine the Life program with the initial curses drawing program to produce a system where a user can sketch an initial Life configuration and then watch it evolve.

13 Standard algorithms

Each new program has its own challenges, its own unique features. But the same old subproblems keep recurring. So, very often at some point in your program you will have to search for a key value in an ordered collection, or sort some data. These subproblems have been beaten to death by generations of computer theoreticians and programmers. The best approaches are now known and defined by standard algorithms.

This chapter presents four of these. Section 13.1 looks at searching for key values. The next three sections explore various "sorting" algorithms.

The study of algorithms once formed a major part of sophomore/junior level studies in Computer Science. Fashions change. You will study algorithms in some subsequent subjects but probably only to a limited extent, and mainly from a theoretical perspective while learning to analyse the complexity of programs.

13.1 FINDING THE RIGHT ELEMENT: BINARY SEARCH

The code in the "keyword picker" (the example in section 12.4) had to search linearly through the table of keywords because the words were unordered. However, one often has data that are ordered. For instance you might have a list of names:

```
"Aavik",      "Abbas",      "Abbot",      "Abet",
...
"Dyer",       "Dzieran",    "Eady",       "Eames",
...
"O'Keefe",    "O'Leary",    ...
...
"Tierney",    "Tomlinson",  "Torrecilla", ...
...
"Yusuf",      "Zitnik",     "Zuzic",      "Zylstra"
```

If you are searching for a particular value, e.g. Sreckovic, you should be able to take advantage of the fact that the data are ordered. Think how you might look up the name

in a phone book. You would open the book in the middle of the "subscriber" pages to find names like "Meadows". The name you want is later in the alphabet so you pick a point half way between Meadows and Zylstra, and find a name like Terry. Too far, so split again between Meadows and Terry to get to Romanows. Eventually you will get to Sreckovic. May take a bit of time but it is faster than checking of Aavik, Abbas, Abbot,

Subject to a few restrictions, the same general approach can be used in computer programs. The restrictions are that all the data must be in main memory, and that it be easy to work out where each individual data item is located. Such conditions are easily satisfied if you have something like an ordered array of names. The array will be in memory; indexing makes it easy to find a particular element. There are other situations where these search techniques are applicable; we will meet some in Part IV when we look at some "tree structures".

The computer version of the approach is known as "binary search". It is "binary" because at each step, the portion of the array that you must search is cut in half ("*bi*-sected"). Thus, in the names example, the first name examined ("Meadows") established that the name sought must be in the second half of the phone book. The second test (on the name "Terry") limited the search to the third quarter of the entire array.

13.1.1 An iterative binary search routine

The binary search algorithm for finding a key value in an array uses two indices that determine the array range wherein the desired element must be located. (Of course, it is possible that the element sought is not present! A binary search routine has to be able to deal with this eventuality as well.) If the array has N elements, then these low and high indices are set to 0 and $N-1$.

The iterative part involves picking an index midway between the current low and high limits. The data value located at this midway point is compared with the desired key. If they are equal, the data has been found and the routine can return the midway value as the index of the data.

If the value at the midway point is lower than that sought, the low limit should be increased. The desired data value must be located somewhere above the current midway point. So the low limit can be set one greater than the calculate midway point. Similarly, if the value at the midway point is too large, the high limit can be lowered so that it is one less than the midway point.

Each iteration halves the range where the data may be located. Eventually, the required data are located (if present in the array). If the sought key does not occur in the array, the low/high limits will cross (low will exceed high). This condition can be used to terminate the loop.

A version of this binary search for character strings is:

```

typedef char Name[32];

int BinarySearch(const Name key, const Name theNames[], int
num)
{
    int low = 0;
    int high = num - 1;

    while(low<=high) {
        int midpoint = (low + high) / 2;
        int result = strcmp(key, theNames[midpoint]);

        if(result == 0)
            return midpoint;

        if(result < 0)
            high = midpoint - 1;
        else
            low = midpoint + 1;
    }
    return -1;
}

```

The function returns the index of where the given key is located in the array argument. The value -1 is returned if the key is not present. The strings are compared using `strcmp()` from the string library. Versions of the algorithm can be implemented for arrays of doubles (the comparison test would then be expressed in terms of the normal greater than (>), equals (==) and less than (<) operators).

Binary search is such a commonly used algorithm that it is normally packaged in `stdlib` as `bsearch()`. The `stdlib` function is actually a general purpose version that can be used with arrays of different types of data. This generality depends on more advanced features of C/C++.

13.1.2 Isn't it a recursive problem?

The binary search problem fits well with the "recursive" model of problem solving (section 4.8). You can imagine the binary search problem being solved by a bureaucracy of clerks. Each clerk follows the rules:

```

    look at the array you've been given, if it has no elements
        report failure
    else
        look at the midpoint of your array,
        if this contains the sought value then report success
        else
            if the midpoint is greater than the sought value
                pass the bottom half of your array to the next clerk

```

```

        asking him to report whether the data are
            present
        report whatever the next clerk said
    else
        pass the top half of your array to the next clerk
        asking him to report whether the data are
            present
        report whatever the next clerk said

```

This version is also easy to implement. Typically, a recursive solution uses a "setting up function" and a second auxiliary function that does the actual recursion. The recursive routine starts with the termination test (array with no elements) and then has the code with the recursive calls. This typical style is used in the implementation:

```

int AuxRecBinSearch(const Name key, const Name theNames[],
    int low, int high)
{
    if(low > high)
        return -1;

    int midpoint = (low + high) / 2;

    int result = strcmp(key, theNames[midpoint]);

    if(result == 0)
        return midpoint;
    else
        if(result < 0)
            return AuxRecBinSearch(key, theNames,
                low, midpoint-1);
        else
            return AuxRecBinSearch(key, theNames,
                midpoint + 1, high);
}

int RecBinarySearch(const Name key, const Name theNames[],
    int num, )
{
    return AuxRecBinSearch(key, theNames, 0, num-1);
}

```

Test for "empty" array

Success, have found item

Recursive call to search bottom half of array

Recursive call to search top half of array

The "setting up" function

In this case, the recursive solution has no particular merit. On older computer architectures (with expensive function calls), the iterative version would be much faster. On a modern RISC architecture, the difference in performance wouldn't be as marked but the iterative version would still have the edge.

Essentially, binary search is too simple to justify the use of recursion. Recursive styles are better suited to situations when there is reasonable amount of work to do as the recursion is unwound (when you have to do something to the result returned by the

clerk to whom you gave a subproblem) or where the problem gets broken into separate parts that both need to be solved by recursive calls.

13.1.3 What was the cost?

The introduction to this section argued that binary search was going to be better than linear search. But how much better is it?

You can get an idea by changing the search function to print out the names that get examined. You might get output something like:

```
Meadows
Terry
Romanows
Smith
Stone
Sreckovic
```

In this case, six names were examined. That is obviously better than checking "Aavik", "Abbas", "Abbot", "Abet", ...; because the linear search would involve examination of hundreds (thousands maybe) of names.

Of course, if you wanted the name Terry, you would get it in two goes. Other names might take more than six steps. To get an idea of the cost of an algorithm, you really have to consider the worst case.

The binary search algorithm can be slightly simplified by having the additional requirement that the key has to be present. If the array has only two elements, then a test against element [0] determines the position of the key; either the test returned "equals" in which case the key is at [0], or the key is at position [1]. So one test picks from an array size of 2. Two tests suffice for an array size of 4; if the key is greater than element [1] then a second test resolves between [2] and [3]. Similarly, three tests pick from among 8 (or less elements).

Tests	Number of items can pick from
1	2
2	4
3	8
4	16
...	...
...	...
?	N

In general, k tests are needed to pick from 2^k items. So how many tests are needed to pick from an array of size N ?

You need:

$$2^k \geq N$$

This equation can be solved for k . The equation is simplified by taking the logarithm of both sides:

$$k \log(2) = \log(N)$$

You can have logarithms to the base 10, or logs to any base you want. You can have logarithms to the base 2. Now $\log_2(2)$ is 1. This simplifies the equation further:

$$\text{number of tests needed} = k = \log_2(N) = \lg(N)$$

The cost of the search is proportional to the number of tests, so the cost of binary search is going to increase as $O(\lg(N))$. Now a $\lg(N)$ rate of growth is a lot nicer than linear growth :

10	3.3
50	5.6
100	6.6
250	7.9
500	8.9
1000	9.9
5000	12.2
10000	13.2
50000	15.6
100000	16.6
500000	18.9
1000000	19.9

A linear search for a key in a set of a million items could (in the worst case) involve you in testing all one million. A binary search will do it in twenty tests.

Of course, there is one slight catch. The data have to be in order before you can do binary search. You are going to have to do some work to get the data into order.

13.2 ESTABLISHING ORDER

Sometimes, you get data that are relatively easy to put into order. For example, consider a class roll ordered by name that identifies pupils and term marks (0 ... 100):

Armstrong, Alice S.	57
Azur, Hassan M.	64
Bates, Peter	33
...	
Yeung, Chi Wu	81
Young, Paula	81

A common need would be a second copy of these data arranged in increasing order by mark rather than by name.

You can see that a pupil with a mark of 100 would be the last in the second reordered array, while any getting 0 would be first. In fact, you can work out the position in the second array appropriate for each possible mark.

This is done by first checking the distribution of marks. For example, the lowest recorded mark might be 5 with two pupils getting this mark; the next mark might be 7, then 8, each with one pupil. Thus the first two slots in the reordered copy of the array would be for pupils with a mark of 5, the next for the pupil with mark 7, and so on. Once this distribution has been worked out and recorded, you can run through the original list, looking at each pupil's mark and using the recorded details of the distribution to determine where to place that pupil in the reordered array. The following algorithm implements this scheme:

1. Count the number of pupils with each mark (or, more generally, the number of "records" with each "key").

These counts have to be stored in an array whose size is determined by the range of key values. In this case, the array has 101 elements (`mark_count[0] ... mark_count[100]`). The data might be something like

```
Mark      0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
mark_count 0, 0, 0, 0, 0, 2, 0, 1, 1, 0, ...
```

2. Change the contents of this `mark_count` array so that its elements record the number of pupils with marks less than or equal to the given mark.

i.e. code something like:

```
for(i=1; i<101;i++)
    mark_count[i] += mark_count[i-1];
```

```
Mark      0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
mark_count 0, 0, 0, 0, 0, 2, 2, 3, 4, 4, ...
```

3. Run through the array of pupils, look up their mark in the `mark_count` array. This gives their relative position in the reordered array (when implementing, you have to remember C's zero-based arrays where position 2 is in `[1]`). The data should be copied to the other array. Then the `mark_count` entry should be reduced by one.

For example, when the first pupil with mark 5 is encountered, the `mark_count` array identifies the appropriate position as being the 2nd in the reordered array (i.e. element `[1]`). The `mark_count` array gets changed to:

```
mark_count 0, 0, 0, 0, 0, 1, 2, 3, 4, 4, ...
```

so, when the next pupil with mark 5 is found his/her position can be seen to be the first in the array (element [0]).

An implementation of this algorithm is:

```
typedef char Name[40];
...

void MakeOrderedCopy(const Name names[], const int marks[],
    int num,
    Name ord_names[], int ord_marks[])
{
    int mark_count[101];

    Initialize counts for
    keys (marks)
    for(int i = 0; i < 101; i++)
        mark_count[i] = 0;

    Get frequencies of
    each key (mark)
    // Count number of occurrences of each mark
    for(i = 0; i < num; i++) {
        int mark = marks[i];
        mark_count[mark]++;
    }

    Turn counts into
    "positions"
    // Make that count of number of pupils with marks less than
    // or equal to given mark
    for(i=1; i<101; i++)
        mark_count[i] += mark_count[i-1];

    Copy data across into
    appropriate positions
    for(i=0; i < num; i++) {
        int mark = marks[i];
        int position = mark_count[mark];
        position--; // correct to zero based array
        // copy data
        strcpy(ord_names[position], names[i]);
        ord_marks[position] = marks[i];
        // update mark_count array
        mark_count[mark]--;
    }
}
```

Note how both data elements, name and mark, have to be copied; we have to keep names and marks together. Since arrays can not be assigned, the `strcpy()` function must be used to copy a Name.

The function can be tested with the following code:

```
#include <iostream.h>
#include <iomanip.h>
```

```

#include <string.h>

typedef char Name[40];

Name pupils[] = {
    "Armstrong, Alice S.",
    "Azur, Hassan M.",
    "Bates, Peter",
    ...
    "Ward, Koren",
    "Yeung, Chi Wu",
    "Young, Paula",
    "Zarra, Daniela"
};

int marks[] = {
    57, 64, 33, 15, 69, 61, 58,
    ...
    45, 81, 81, 78
};

int num = sizeof(marks) / sizeof(int);
int num2 = sizeof(pupils) / sizeof(Name);

int main()
{
    Name ordnms[500];
    int ordmks[500];
    MakeOrderedCopy(pupils, marks, num, ordnms, ordmks);
    cout << "Class ordered by names: " << endl;
    for(int i = 0; i < num; i++)
        cout << setw(40) << pupils[i] << setw(8)
            << marks[i] << endl;
    cout << "\n\nOrdered by marks:" << endl;
    for( i = 0; i < num; i++)
        cout << setw(40) << ordnms[i] << setw(8)
            << ordmks[i] << endl;
    return 0;
}

```

which produces output like:

```

Class ordered by names:
    Armstrong, Alice S.      57
    Azur, Hassan M.         64
    Bates, Peter            33
...
...
    Yeung, Chi Wu           81
    Young, Paula            81

```

	Zarra, Daniela	78
Ordered by marks:		
	McArdie, Hamish	5
	Hallinan, Jack	5
...		
...		
	Young, Paula	81
	Yeung, Chi Wu	81
	Horneman, Sue	87
	Goodman, Jerry	91

These example data illustrate another minor point. The original list had Yeung and Young in alphabetic order, both with mark 81. Since their marks were equal, there was no need to change them from being in alphabetic order. However the code as given above puts pupils with the same mark in *reverse* alphabetic order.

"Stable sorts" Sometimes, there is an extra requirement: the sorting process has to be "stable". This means that records that have the same key values (e.g. pupils with the same mark) should be left in the same order as they originally were.

It is easy to change the example code to achieve stability. All you have to do is make the final loop run backwards down through the array:

```
for(i=num-1; i >= 0; i--) {
    int mark = marks[i];
    int position = mark_count[mark];
    position--; // correct to zero based array
    // copy data
    strcpy(ord_names[position], names[i]);
    ord_marks[position] = marks[i];
    // update mark_count array
    mark_count[mark]--;
}
```

Implementation limitations This implementation is of course very specific to the given problem (e.g. the `mark_count` array is explicitly given the dimension 101, it does not depend on a data argument). The function can be made more general but this requires more advanced features of the C/C++ language. Nevertheless, the code is simple and you should have no difficulty in adapting it to similar problems where the items to be sorted have a narrow range of keys and you want two copies of the data.

What is the cost?

To get an idea of the cost of this sorting mechanism, you have to look at the number of times each statement gets executed. Suppose that `num`, the number of data elements, was 350, then the number of times that each statement was executed would be as shown below:

```

void MakeOrderedCopy(const Name names[], const int marks[],
    int num,
    Name ord_names[], int ord_marks[])
{
    int mark_count[101];

    for(int i = 0; i < 101; i++)
101        mark_count[i] = 0;

    // Count number of occurrences of each mark
    for(i = 0; i < num; i++) {
350        int mark = marks[i];
350        mark_count[mark]++;
    }
    // Make that count of number of pupils with marks
    // less than or equal to given mark
    for(i=1; i<101; i++)
101        mark_count[i] += mark_count[i-1];

    for(i= 0; i < num; i++) {
350        int mark = marks[i];
350        int position = mark_count[mark];
350        position--; // correct to zero based array
        // copy data
350        strcpy(ord_names[position], names[i]);
350        ord_marks[position] = marks[i];
350        mark_count[mark]--;
    }
}

```

The costs of these loops is directly proportional to the number of times they are executed. So the cost of this code is going to be proportional to the number of entries in the array (or the range of the key if this was larger, the '101' loops would represent the dominant cost if you had less than 100 records to sort).

This algorithm is $O(N)$ with N the number of items to sort. That makes it a relatively cheap algorithm.

13.3 A SIMPLE SORT OF SORT

The last example showed that in some special cases, you could take data and get them sorted into order in "linear time". However, the approach used is limited.

There are two limitations. Firstly, it made a sorted copy of the data so that you had to sufficient room in the computer's memory for two sets of data (the initial set with the pupils arranged in alphabetic order, and the sorted set with them arranged by mark). This is usually not acceptable. In most cases where it is necessary to sort data, you can't have two copies of the entire data set. You may have one or two extra items (records)

*Normally require
sorting "in situ"*

but you can't duplicate the whole set. Data have to be sorted "in situ" (meaning "in place"); the original entries in the data arrays have to be rearranged to achieve the required order.

Small range of key values

The other simplifying factor that does not usually apply was the limited range of the keys. The keys were known to be in the range 0...100. This made it possible to have the `mark_count[101]` array. Usually the keys span a wider range, e.g. bank accounts can (I suppose) have any value from -20000000000 to +20000000000. You can't allocate an array equivalent to `mark_count[101]` if you need four thousand million entries.

When you don't have the special advantages of data duplication and limited key range, you must use a general purpose sorting function and you will end up doing a lot more work.

The late 1950s and early 1960s was a happy time for inventors of sorting algorithms. They invented so many that the topic has gone quite out of fashion. There is a whole variety of algorithms from simple (but relatively inefficient) to complex but optimized. There are also algorithms that define how to sort data collections that are far too large to fit into the memory of the computer. These "external sorts" were important back when a large computer had a total of 32768 words of memory. Now that memory sizes are much larger, external sorts are of less importance; but if you ever do get a very large set of data you can dig out those old algorithms, you don't have to try to invent your own.

There is a fairly intuitive method for sorting the data of an array into ascending order. The first step is to search through the array so as to select the smallest data value; this is then exchanged for the data element that was in `element [0]`. The second step repeats the same process, this time searching the remaining unsorted subarray `[1]...[N-1]` so as to select its smallest element, which gets swapped for the data value originally in `[1]`. The selection and swapping processes are repeated until the entire array has been processed.

A version of this code, specialized to handle an array of characters is:

Code to find index of minimum value in rest of array

Code to swap data value with smallest

```
void SelectionSort(char data[], int n)
{
    for(int i = 0; i < n-1; i++) {
        int min = i;
        for(int j=i+1; j<n; j++)
            if(data[j] < data[min]) min = j;

        char temp = data[min];
        data[min] = data[i];
        data[i] = temp;
    }
}
```

The old cliché states "A picture is worth a thousand words"; so how about a few pictures to illustrate how this sort process works. The following program uses the `curses` package to draw pictures illustrating the progress of the sort:


```
#include "CG.h"

char testdata[] = {
    'a', 'r', 'g', 'b', 'd',
    'i', 'q', 'j', 'm', 'o',
    't', 'p', 'l', 's', 'n',
    'f', 'c', 'k', 'e', 'h'
};

int n = sizeof(testdata) / sizeof(char);

void ShowArray()
{
    CG_ClearWindow();
    for(int i=0; i < n; i++) {
        int x = i + 1;
        int y = 1 + testdata[i] - 'a';
        CG_PutCharacter(testdata[i], x, y);
    }
    CG_Prompt("continue>");
    char ch = CG_GetChar();
}

void SelectionSort(char data[], int n)
{
    ShowArray();
    for(int i = 0; i < n-1; i++) {
        int min = i;
        for(int j=i+1; j<n; j++)
            if(data[j] < data[min]) min = j;
        char temp = data[min];
        data[min] = data[i];
        data[i] = temp;
        ShowArray();
    }
}

int main()
{
    CG_Initialize();
    SelectionSort(testdata, n);
    CG_Reset();
    return 0;
}
```

Figure 13.1 illustrates the output of this program.

What is the cost?

The cost is determined by the number of data comparisons and data exchanges that must be done in that inner loop. If there are N elements in the array, then:

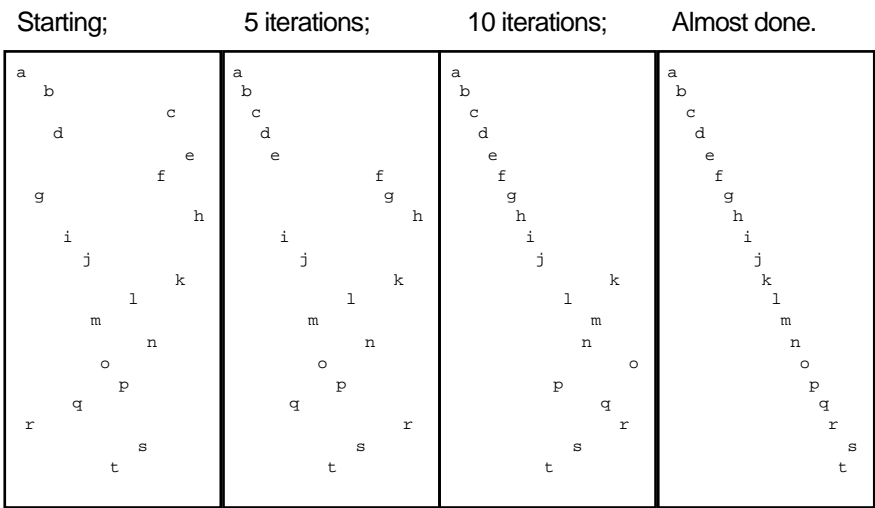


Figure 13.1 Illustrating selection sort.

i	iterations of j loop
0	N-1
1	N-2
2	N-3
...	...
N-2	1

Each iteration of the *j* loop involves a comparison (and, if you are unlucky, an assignment). The cost will be proportional to the sum of the work of all the *j* loops:

cost ∝ $\sum_{i=1}^{i=N-1} (N-i)$

It is an arithmetic progression and its sum is "first term plus last term times half the number of terms:

$(N-1 + 1) (N-1)/2$
 $0.5N^2 - 0.5N$

An *N*² cost The cost of this sorting process increases as the square of the number of elements.

13.4 QUICKSORT: A BETTER SORT OF SORT

An N^2 sorting process is not that wildly attractive; doubling the number of elements increases the run time by a factor of four. Since you know that a data element can be found in an ordered array in time proportional to $\lg(N)$, you might reasonably expect a sort process to be cheaper than N^2 . After all, you can think of "sorting" as a matter of adding elements to an ordered set; you have N elements to add, finding the right place for each one should be $\lg(N)$, so an overall cost of $N\lg(N)$ seems reasonable.

That argument is a little spurious, but the conclusion is correct. Data can be sorted in $N\lg(N)$ time. There are some algorithms that guarantee an $N\lg(N)$ behaviour. The most popular of the sophisticated sorting algorithms does not guarantee $N\lg(N)$ behaviour; in some cases, e.g. when the data are already more or less sorted, its performance deteriorates to N^2 . However, this "Quicksort" algorithm usually runs well.

13.4.1 The algorithm

Quicksort is recursive. It is one of those algorithms that (conceptually!) employs an bureaucracy of clerks each of whom does a little of the work and passes the rest on to its colleagues. The basic rules that the clerks follow are:

- 1 If you are given an array with one element, say that you have sorted it.
- 2 If you are given an array with several data elements, "shuffle them" into two groups – one group containing all the "large" values and the other group with all the "small" values. Pass these groups to two colleagues; one colleague sorts the "large" values, the other sorts the "small" values.

Ideally, each clerk breaks the problem down into approximately equal sized parts so that the next two clerks get similar amounts of work. So, if the first clerk in a group is given the data:

37, 21, 12, 103, 82, 97, 64, 62, 62, 28, 44, 50, 33, 91

then (s)he splits this into:

37, 21, 12, 33, 50, 44, 28,

and

62, 62, 64, 97, 82, 103, 91

It is up to the next two clerks to sort out these subarrays.

In the example just shown, the clerk knew "intuitively" that the best partitioning value would be around 60 and moved values less than this into the "small" group with the others in the "large" group. Usually, the best partitioning value is not known. The scheme works even if a non-optimal value is used. Suppose that the first clerk picked 37, then the partitions could be:

33, 21, 12, 28

and

82, 97, 64, 62, 62, 103, 44, 50, 37, 91

The first partition contains the values less than 37, the second contains all values greater than or equal to 37. The partitions are unequal in size, but they can still be passed on to other clerks to process.

The entire scheme for dealing with the data is shown in Figure 13.2. The pattern of partitions shown reflects the non-optimal choice of partitioning values. In this example, it is quite common for the partitioning to separate just one value from a group of several others rather than arranging an even split. This necessitates extra partitioning steps later on. It is these deviations from perfect splitting that reduce the efficiency of Quicksort from the theoretical $N \lg(N)$ optimum.

How should the data be shuffled to get the required groups?

You start by picking the value to be used to partition the data, and for lack of better information you might as well take the first value in your array, in this case 37. Then you initialize two "pointers" – one off the left of the array, and one off the right end of the array:

Left ↑ 37, 21, 12, 103, 82, 97, 64, 62, 62, 28, 44, 50, 33, 91 ↑ Right

Move the "right" pointer to the left until its pointing to a data value less than or equal to the chosen partitioning value:

Left ↑ 37, 21, 12, 103, 82, 97, 64, 62, 62, 28, 44, 50, 33, 91 ↑ Right

Next, move the "left" pointer to the right until its pointing to a data value greater than or equal to the partitioning value:

Left ↑ 37, 21, 12, 103, 82, 97, 64, 62, 62, 28, 44, 50, 33, 91 ↑ Right

You have now identified a "small" value in the right hand part of the array, and a "large" value in the left hand part of the array. These are in the wrong parts; so exchange them:

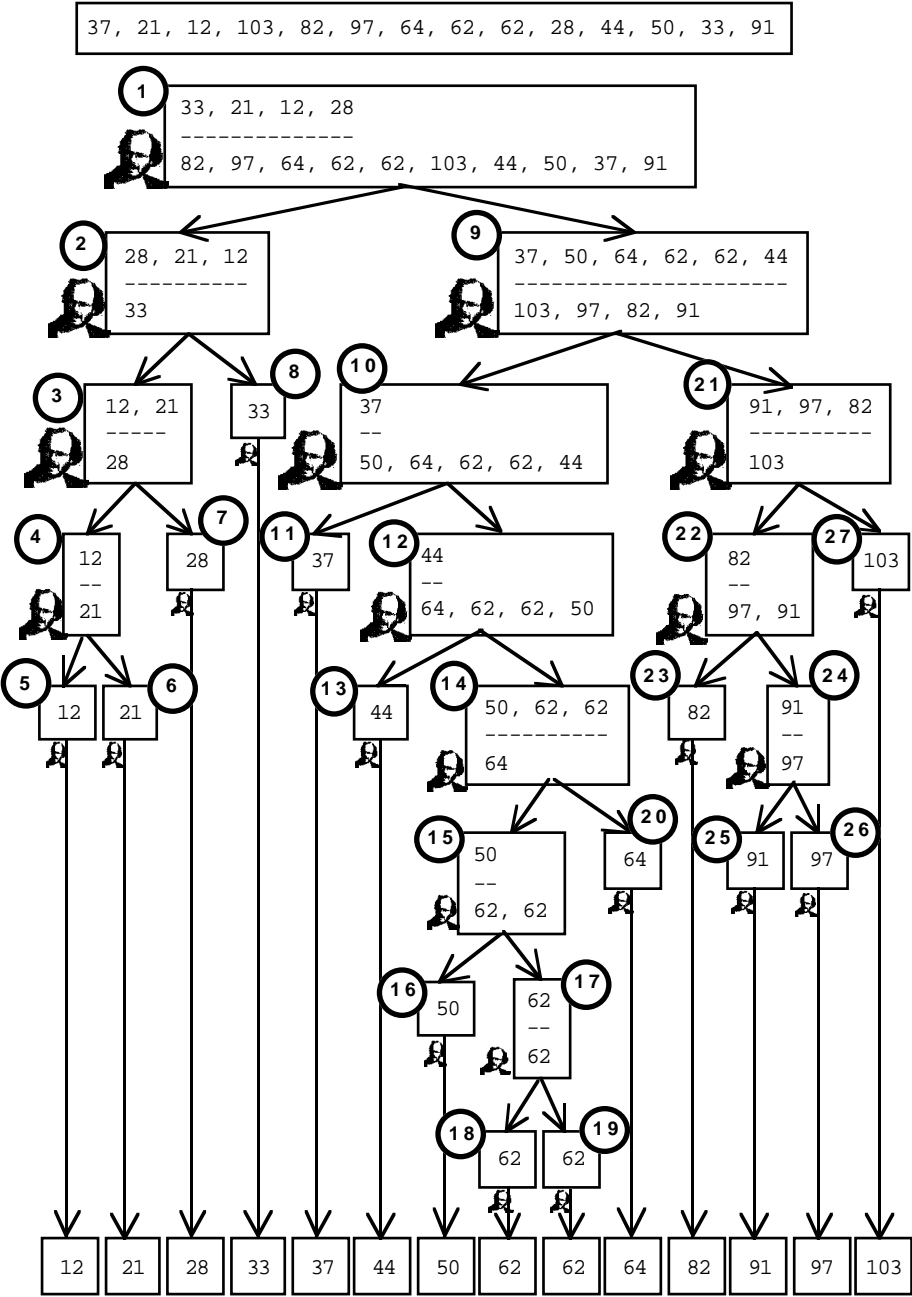


Figure 13.2 A bureaucracy of clerks "Quicksorts" some data. The circled numbers identify the order in which groups of data elements are processed.

```

      33, 21, 12, 103, 82, 97, 64, 62, 62, 28, 44, 50, 37, 91
Left  ↑                               ↑Right

```

Do it again. First move the right pointer down to a value less than or equal to 37; then move the left pointer up to a value greater than or equal to 37:

```

      33, 21, 12, 103, 82, 97, 64, 62, 62, 28, 44, 50, 37, 91
      Left  ↑                               ↑Right

```

Once again, these data values are out of place, so they should be exchanged:

```

      33, 21, 12, 28, 82, 97, 64, 62, 62, 103, 44, 50, 37, 91
      Left  ↑                               ↑Right

```

Again move the right pointer down to a value less than 37, and move the left pointer up to a greater value:

```

      33, 21, 12, 28, 82, 97, 64, 62, 62, 103, 44, 50, 37, 91
                        ↑Right
                        Left  ↑

```

In this case, the pointers have crossed; the "right" pointer is to the left of the "left" pointer. When this happens, it means that there weren't any more data values that could be exchanged.

The "clerk" working on this part of the problem has finished; the rest of the work should be passed on to others. The data in the array up to the position of the "right" pointer are those values less than the value that the clerk chose to use to do the partitioning; the other part of the array holds the larger values. These two parts of the array should be passed separately to the next two clerks.

The scheme works. The rules are easy enough to be understood by a bureaucracy of clerks. So they are easy enough to program.

13.4.2 An implementation

The recursive Quicksort function takes as arguments an array, and details of the range that is to be processed:

```
void Quicksort( int d[], int left, int right);
```

The details of the range will be given as the indices of the leftmost (low) and rightmost (high) data elements. The initial call will specify the entire range, so if for example you had an array `data[100]`, the calling program would invoke the `Quicksort()` function as:

```
Quicksort(data, 0, 99);
```

Subsequent recursive calls to `Quicksort()` will specify subranges e.g. `Quicksort(data, 0, 45)` and `Quicksort(data, 46, 99)`.

The partitioning step that splits an array into two parts (and shuffles data so one part contains the "low" values) is sufficiently complex to merit being a separate function. Like `Quicksort()` itself, this `Partition()` function needs to be given the array and "left and right" index values identifying the range that is to be processed:

```
int Partition( int d[], int left, int right);
```

The `Partition()` function should return details of where the partition point is located. This "split point" will be the index of the array element such that `d[left] ... d[split_point]` all contain values less than the value chosen to partition the data values. Function `Partition()` has to have some way of picking a partitioning value; it can use the value in the leftmost element of the subarray that it is to process.

The code for `Quicksort()` itself is nice and simple:

```
void Quicksort( int d[], int left, int right)
{
    if(left < right) {
        int split_pt = Partition(d, left, right);
        Quicksort(d, left, split_pt);
        Quicksort(d, split_pt+1, right);
    }
}
```

*Shuffle the data
Give the two
subarrays to two
assistants*

If the array length is 1 (`left == right`), then nothing need be done; an array of one element is sorted. Otherwise, use `Partition()` to shuffle the data and find a split point and pass the two parts of the split array on to other incarnations of `Quicksort()`.

The `Partition()` function performs the "pointer movements" described in the algorithm. It actually uses two integer variables (`lm` = left margin pointer, and `rm` = right margin pointer). These are initialized "off left" and "off right" of the subarray to be processed. The partitioning value, `val`, is taken as the leftmost element of the subarray.

```
int Partition( int d[], int left, int right)
{
    int val = d[left];
    int lm = left-1;
    int rm = right+1;
```

*Initialize val and
"left and right
pointers"*

The movement of the pointers is handled in a "forever" loop. The loop terminates with a return to the caller when the left and right pointers cross.

Inside the for loop, there are two `do ... while` loops. The first `do ... while` moves the right margin pointer leftwards until it finds a value less than or equal to `val`. The

second do ... while loop moves the left margin pointer rightwards, stopping when it finds a value greater than or equal to val.

If the left and right margin pointers have crossed, function Partition() should return the position of the right margin pointer. Otherwise, a pair of misplaced values have been found; these should be exchanged to get low values on the left and high values on the right. Then the search for misplaced values can resume.

<p><i>Move right margin pointer leftwards</i></p> <p><i>Move left margin pointer rightwards</i></p> <p><i>Exchange misplaced data</i></p> <p><i>Return if all done</i></p>	<pre> for(;;) { do rm--; while (d[rm] > val); do lm++; while(d[lm] < val); if(lm<rm) { int tempr = d[rm]; d[rm] = d[lm]; d[lm] = tempr; } else return rm; } </pre>
---	---

What does it cost?

If you really want to know what it costs, you have to read several pages of a theoretical text book. But you can get a good idea much more simply.

The partitioning step is breaking up the array. If it worked ideally, it would keep splitting the array into halves. This process has to be repeated until the subarrays have only one element. So, in this respect, it is identical to binary search. The number of splitting steps to get to a subarray of size one will be proportional to $\lg(N)$.

In the binary search situation, we only needed to visit one of these subarrays of size one – the one where the desired key was located. Here, we have to visit all of them. There are N of them. It costs at least $\lg(N)$ to visit each one. So, the cost is $N\lg(N)$. You can see from Figure 13.2 that the partitioning step frequently fails to split a subarray into halves; instead it produces a big part and a little part. Splitting up the big part then requires more partitioning steps. So, the number of steps needed to get to subarrays of size one is often going to be greater than $\lg(N)$; consequently, the cost of the sort is greater than $N\lg(N)$.

In the worst case, every partition step just peels off a single low value leaving all the others in the same subarray. In that case you will have to do one partitioning step for

the first element, two to get the second, three for the third, ... up to N-1 for the last. The cost will then be approximately

$$1 + 2 + 3 + \dots + (N-2) + (N-1)$$

or $0.5N^2$.

13.4.3 An enhanced implementation

For most data, a basic Quicksort() works well. But there are ways of tweaking the algorithm. Many focus on tricks to pick a better value for partitioning the subarrays. It is a matter of trade offs. If you pick a better partitioning value you split the subarrays more evenly and so require fewer steps to get down to the subarrays of size one. But if it costs you a lot to find a better partitioning value then you may not gain that much from doing less partitions.

There is one tweak that is usually worthwhile when you have big arrays (tens of thousands). You combine two sort algorithms. Quicksort() is used as the main algorithm, but when the subarrays get small enough you switch to an alternative like selection sort. What is small enough? Probably, a value between 5 and 20 will do.

The following program illustrates this composite sort (Borland users may have to reduce the array sizes or change the "memory model" being used for the project):

```
#include <iostream.h>
#include <stdlib.h>

const int kSMALL_ENOUGH = 15;
const int kBIG = 50000; // Relying on int = long
int data[kBIG];

void SelectionSort(int data[], int left, int right)
{
    for(int i = left; i < right; i++) {
        int min = i;
        for(int j=i+1; j<= right; j++)
            if(data[j] < data[min]) min = j;
        int temp = data[min];
        data[min] = data[i];
        data[i] = temp;
    }
}

int Partition( int d[], int left, int right)
{
    int val =d[left];
    int lm = left-1;
    int rm = right+1;
```

*SelectionSort of a
subarray*

Partition code

```

        for(;;) {
            do
                rm--;
                while (d[rm] > val);

            do
                lm++;
                while( d[lm] < val);

            if(lm<rm) {
                int tempr = d[rm];
                d[rm] = d[lm];
                d[lm] = tempr;
            }
            else
                return rm;
        }
    }

Quicksort driver void Quicksort( int d[], int left, int right)
    {
        if(left < (right-kSMALL_ENOUGH)) {
            int split_pt = Partition(d,left, right);
            Quicksort(d, left, split_pt);
            Quicksort(d, split_pt+1, right);
        }
        else SelectionSort(d, left, right);
    }

Call SelectionSort on  
smaller subarrays

int main()
{
    int i;
    long sum = 0;
Get some data, with  
lots of duplicates for(i=0;i <kBIG;i++)
        sum += data[i] = rand() % 15000;

    Quicksort(data, 0, kBIG-1);

Check the results!
    int last = -1;
    long sum2 = 0;
    for(i = 0; i < kBIG; i++)
        if(data[i] < last) {
            cout << "Oh ....; the data aren't in order"
                << endl;
            cout << "Noticed the problem at i = " << i
                << endl;
            exit(1);
        }
        else {
            sum2 += last = data[i];
        }
    if(sum != sum2) {

```

```

        cout << "Oh ...; we seem to have changed the data"
              << endl;
    }

    return 0;
}

```

13.5 ALGORITHMS, FUNCTIONS, AND SUBROUTINE LIBRARIES

Searching and sorting – they are just a beginning.

For one of your more advanced computing science subjects you will get a reference book like "Introduction to Algorithms" by Cormen, Leiserson, and Rivest (MIT press). There are dozens of other books with similar titles, but the Cormen book will do (its pseudo-code for the algorithms is correct which is more than can be said for the "code" in some of the alternatives).

The Cormen book is fairly typical in content. Like most of the others, it starts with a section on topics such as recurrence relations, probability, and complexity analysis. Then it has several chapters on sorting and searching (things like binary search). The sections on "searching" explore the use of the "tree structures" introduced in Chapter 21 in Part IV of this text. Hashing algorithms, touched on briefly in Chapter 18, may also be covered in detail.

*Searching and
sorting*

The algorithm books generally review "string processing". The most basic "string processing" task is "find a word in a block of text". Some "string search" algorithm has been implemented in the editor of your IDE; it does the work for the Edit/Search option. This kind of string searching is relatively simple, you just have to match a word (e.g. a misspelled keyword viod) against the text. Even so, there are a number of fairly complex algorithms for doing this; algorithms that implement tricks to minimize the work that must be done, and hence the time used to perform a search. In most IDE editors, there are more powerful search facilities (you will probably never use them). They may be called "grep" searches ("grep" was the name of an early program on Unix that provided very elaborate mechanisms for searching text in files). A "grep" search allows you to specify a pattern – "I want to find a word that starts with a capital letter, contains two adjacent o's, and does not end in 'l'". There are algorithms that define efficient ways of performing such searches.

"String processing"

In some texts, the string processing section also introduces algorithms for "compressing" data files.

Usually, these books contain some "Graph algorithms". These graphs are *not* x,y plots of functions. Rather, a graph represents a network – any kind of network; see Figure 13.3. A graph is made up of things (represented as "nodes" or "vertices") that are related (the "edges"). The nodes might represent cities, with the edges being connecting air-routes; or they might be atoms, with the edges being bonds; or they might be electrical components with the edges as wires. There are many properties of

Graph algorithms

graphs that have to be calculated, e.g. the distances between all pairs of cities, or the shortest path, or the existence of cycles (rings) in a graph that represents a molecule. Because graphs turn up in many different practical contexts, graph algorithms have been extensively studied and there are good algorithmic solutions for many graph related problems.

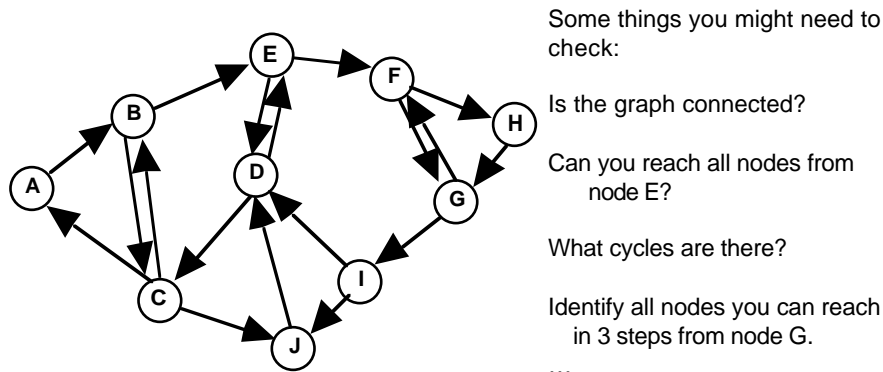


Figure 13.3 Graphs – networks of nodes and edges.

Searching, sorting, strings, and graphs; these are the main ingredients of the algorithm books. But, there are algorithms for all sorts of problems; and the books usually illustrate a variety of the more esoteric.

***Bipartite graph
matching***

Suppose you had the task of making up a list of partners for a high school dance. As illustrated in Figure 13.4 you would have groups of boys and girls. There would also be links between pairs, each link going from a boy to a girl. Each link would indicate sufficient mutual attraction that the pair might stay together all evening. Naturally, some individuals are more attractive than others and have more links. Your task is to pair off individuals to keep them all happy.

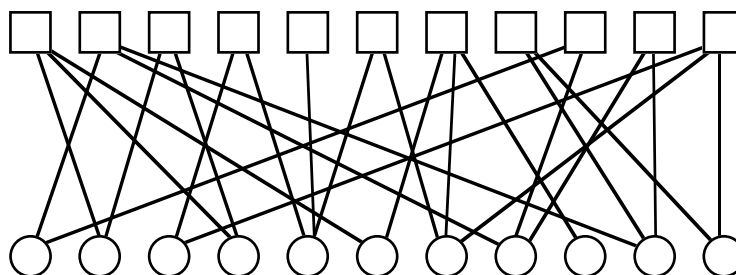


Figure 13.4 The bipartite matching (or "high school prom") problem. Find a way of matching elements from the two sets using the indicated links.

What would you do? Hack around and find your own ad hoc scheme? There is no need. This is a standard problem with a standard solution. Of course, theoreticians have given it an imposing name. This is the "bipartite graph matching" problem. There is an algorithm to solve it; but you would never know if you hadn't flicked through an algorithms book.

How about the problem in Figure 13.5? You have a network of pipes connecting a "source" (e.g. a water reservoir) to a "sink" (the sewers?). The pipes have different capacities (given in some units like cubic metres per minute). What is the maximum rate at which water might flow out of the reservoir? Again, don't roll your own. There is a standard algorithm for this one, Maximal Flow.

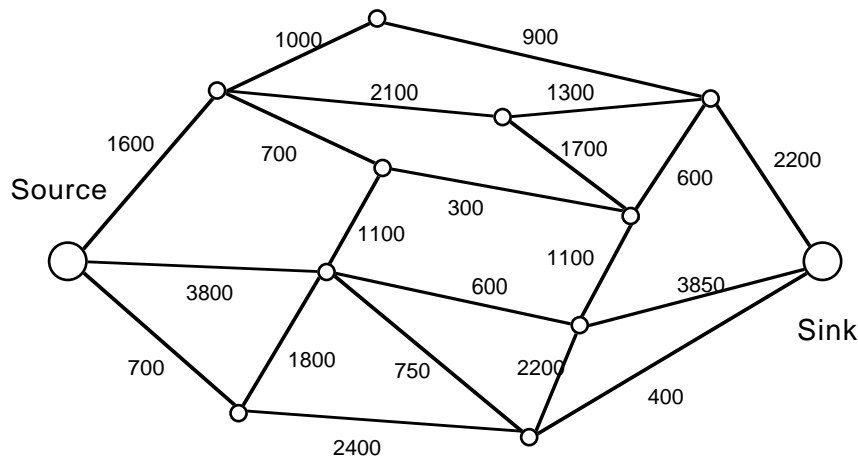


Figure 13.5 Maximal flow problem: what is the maximum flow from source to sink through the "pipes" indicated.

Figure 13.6 illustrates the "package wrapping problem" (what strange things academic computer scientists study!). In this problem, you have a large set of points (as x,y coordinate pairs). The collection is in some random order. You have to find the perimeter points.

Despite their five hundred to one thousand pages, the algorithms books are just beginning. There are thousands more algorithms, and implementations as C functions or FORTRAN subroutines out there waiting to be used. The "Association for Computing Machinery" started publishing algorithms in its journal "Communications of the ACM" back in the late 1950s. There is now a special ACM journal, ACM Transactions on Mathematical Software, that simply publishes details of new algorithms together with their implementation. In recent years, you could for example find how to code up "Wavelet Transform Algorithms for Finite Duration Discrete Time Signals" or a program to "Generate Niederreiter's Low Discrepancy Sequences".

The CACM libraries

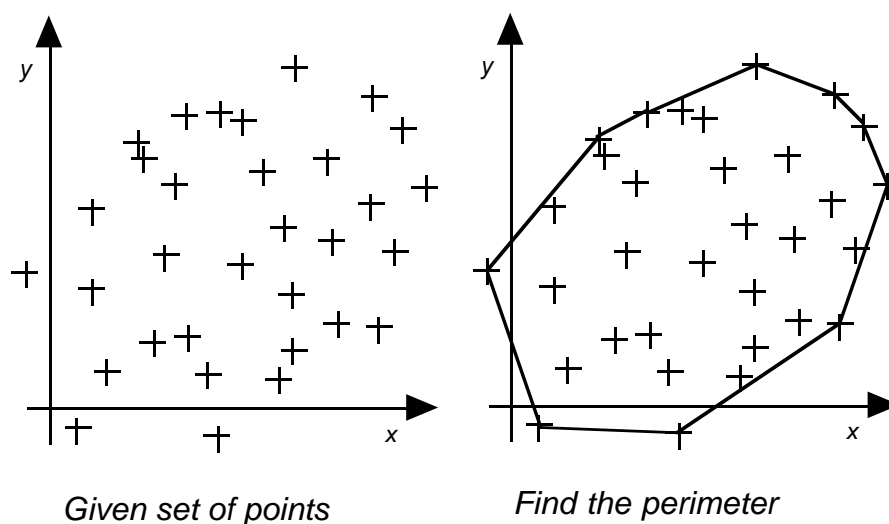


Figure 13.6 The "package wrapping" (convex hull) problem.

Numerical algorithms

Several groups, e.g. Numerical Algorithms Group, have spent years collecting algorithms and coded implementation for commonly needed things like "matrix inversion", "eigenvalues", "ordinary differential equations", "fast fourier transforms" and all the other mathematical processing requirements that turn up in scientific and engineering applications. These numerical routines are widely applicable. Some collections are commercially available; others are free. You may be able to get at the free collections via the "netlib" service at one of ATT's computers.

Discipline specific function libraries

Beyond these, each individual engineering and scientific group has established its own library of functions. Thus, "Quantum Chemists" have functions in their QCPE library to calculate "overlap integrals" and "exchange integrals" (whatever these may be).

Surf the net

These collections of algorithms and code implementations are a resource that you need to be aware of. In later programs that you get to write, you may be able to utilize code from these collections. Many of these code libraries and similar resources are available over the World Wide Web (though you may have to search diligently to locate them).

14 Tools

This chapter introduces two powerful support tools.

14.1 THE "CODE COVERAGE" TOOL

A "code coverage" tool helps you determine whether you have properly tested your program. Think about all the `if ... else ...` and `switch()` statements in your code. Each involves optionally executed code, with calls to other functions where there are further conditional constructs. These conditional constructs mean that there can be a very large number of different execution paths through your code.

Role of a code coverage tool

You can never be certain that you have tested all paths through a complex program; so there will always be chances that some paths have errors where data are combined incorrectly. But if you have never tested certain paths, there can be gross errors that will crash your program.

The simple programs that we have considered so far don't really justify the use of a code coverage tool; after all, things simulating the π -cannon don't involve that many choices and there is very little dependence on different inputs. But as you move to more complex things, like some of the "tree algorithms" in Part IV, code coverage tools become more and more necessary. The "tree algorithms" build up complicated data structures that depend on the input data. Rules implemented in the algorithms define how these structures are to change as new data elements get added. Some of these rules apply to relatively rare special cases that occur only for specific combinations of data values. The code for the "tree algorithms" has numerous paths, including those for these rare special cases.

How can you test all paths? Basically, you have to choose different input data so as to force all processing options to be executed. Choosing the data is sometimes hard; you may think that you have included examples that cover all cases, but it is difficult to be certain.

This is where code coverage tools get involved. The basic idea is that the compiler and run-time system should help you check that your tests have at least executed the

Compiler adds record keeping code

code on every conditional branch of your program. Unfortunately, a "code coverage" tool is not yet a standard part of the IDEs available for personal computers. The example in this section uses the "tcov" tool on Sun Unix.

Special compile time options have to be specified when preparing a program for this form of analysis. These options direct the compiler to add extra instructions to those that would be generated from the program's source code. As well as extra instructions embedded in the code, the compiler adds an extra data table, a startup function and a termination function. The extra instructions, and related components, arrange for counts to be kept of the number of times that each conditional block is executed, when the program terminates these counts are saved to a data file.

You can imagine the process as being one where the compiler converts the following simple program:

```
#include <iostream.h>

int main()
{
    cout << "Enter Number ";
    int num;
    cin >> num;
    if(num >= 0)
        cout << "that was positive" << endl;
    else
        cout << "that was negative" << endl;
    return 0;
}
```

into an "instrumented" version:

```
#include <iostream.h>

int __counters[3];
extern void __InitializeCounters(int d[], int n);
extern void __SaveCountersToFile(int d[], int n);

int main()
{
    __InitializeCounters(__counters, 3);
    __counters[0]++;
    cout << "Enter Number ";
    int num;
    cin >> num;
    if(num >= 0) {
        __counters[1]++;
        cout << "that was positive" << endl;
    }
    else {
        __counters[2]++;
        cout << "that was negative" << endl;
    }
}
```



```

    }
    __SaveCountersToFile(__counters, 3);
    return 0;
}

```

The "instrumented" program will run just like the original, except that it saves the counts to a file before terminating. In a properly implemented system, each time the program is run the new counts are added to those in any existing record file.

When you have run your program several times on different data, you get the *Analysis tool* analysis tool to interpret the contents of the file with the counts. This analysis tool combines the counts data with the original source code to produce a listing that illustrates the number of times each different conditional block of code has been executed. Sections of code that have never been executed are flagged, warning the tester that checks are incomplete. The following output from the Unix tcov tool run on a version of the Quicksort program from Chapter 13.

```

const int kSMALL_ENOUGH = 15;
const int kBIG = 50000;
int data[kBIG];

void SelectionSort(int data[], int left, int right)
6246 -> {
        for(int i = left; i < right; i++) {
43754 ->             int min = i;
                for(int j=i+1; j<= right; j++)
248487 ->                     if(data[j] < data[min]) min =
35251 ->                         int temp = data[min];
43754 ->                         data[min] = data[i];
                                data[i] = temp;
                }
6246 -> }

int Partition( int d[], int left, int right)
6245 -> {
        int val =d[left];
        int lm = left-1;
        int rm = right+1;
        for(;;) {
152418 ->             do
                    rm--;
518367 ->             while (d[rm] > val);
152418 ->             do
                    lm++;
412418 ->             while( d[lm] < val);
152418 ->             if(lm<rm) {
146173 ->                 int tempr = d[rm];
                    d[rm] = d[lm];
                    d[lm] = tempr;
            }
        }
    }

```

```

                                else
6245 ->                                return rm;
##### ->                                }
##### -> }

void Quicksort( int d[], int left, int right)
12491 -> {
                                if(left < (right-kSMALL_ENOUGH)) {
6245 ->                                int split_pt = Partition(d,left,
                                Quicksort(d, left, split_pt);
                                Quicksort(d, split_pt+1, right);
                                }
6246 ->                                else SelectionSort(d, left, right);
6246 -> }

int main()
1 -> {
                                int i;
                                long sum = 0;
                                for(i=0;i <kBIG;i++)
50000 ->                                sum += data[i] = rand() % 15000;

50000 ->                                Quicksort(data, 0, kBIG-1);

                                int last = -1;
                                long sum2 = 0;
                                for(i = 0; i < kBIG; i++)
50000 ->                                if(data[i] < last) {
##### ->                                cout << "Oh ....; the data
                                cout << "Noticed the problem at
                                exit(1);
                                }

50000 ->                                else {
                                sum2 += last = data[i];
                                }

50000 ->                                if(sum != sum2) {
##### ->                                cout << "Oh ....; we seem to have
                                }

1 ->                                return 0;

```

All the parts of the program that we want executed have indeed been executed.

For real programs, code coverage tools are an essential part of the test and development process.

14.2 THE PROFILER

A code coverage tool helps you check that you have tested your code, a "Profiler" helps you make your code faster.

First some general advice on speeding up programs:

General advice 1:

1. Make it work.
2. Make it fast.

Step 2 is optional.

General advice 2:

The slow bit isn't the bit you thought would be slow.

General advice 3:

Fiddling with "register" declarations, changing from arrays to pointers and general hacking usually makes not one measurable bit of difference.

General advice 4

The only way to speed something up significantly will involve a fundamental change of algorithm (and associated data structure).

General advice 5

Don't even think about making changes to algorithms until you have acquired some detailed timing statistics.

Most environments provide some form of "profiling" tool that can be used to get the timing statistics that show where a program really is spending its time. To use a profiling tool, a program has to be compiled with special compile-time flags set. When these flags are set, the compiler and link-loader set up auxiliary data structures that allow run time addresses to be mapped back to source code. They also arrange for some run-time component that can be thought of as regularly interrupting a running program and asking where its program counter (PC) is. This run-time component takes the PC value and works out which function it corresponds to and updates a counter for that function. When the program finishes, the counts (and mapping data) are saved to file. The higher the count associated with a particular bit of code, the greater the time spent in that code.

A separate utility program can take the file with counts, and the source files and produce a summary identifying which parts of a program use most time. The following data were obtained by using the Unix profiler to analyze the performance of the extended Quicksort program:

%Time	Seconds	Cumsecs	Name
44.7	0.42	0.42	__0FJPartitionPiiTC
23.4	0.22	0.64	__0FNSelectionSortPiiTC
11.7	0.11	0.75	main
6.4	0.06	0.81	.rem
6.4	0.06	0.87	.mul
1.1	0.01	0.94	rand

Both the Symantec and the Borland IDEs include profilers that can provide information to that illustrated.

The largest portion of this program's time is spent in the `Partition()` function, the `Quicksort()` function itself doesn't even register because it represents less than 1% of the time. (The odd names, like `__0FNSelectionSortPiiTC` are examples of "mangled" names produced by a C++ compiler.)

Once you know where the program spends its time, then you can start thinking about minor speed ups (register declarations etc) or major improvements (better algorithms).

15 Design and documentation : 1

The examples given in earlier chapters, particularly Chapter 12, have in a rather informal way illustrated a design style known as "top-down functional decomposition".

"Top down functional decomposition" is a limited approach to design. It works very well for simple scientific and engineering applications that have the basic structure:

```
initialize  
get the input data  
process the data  
print the results
```

and where the "data" are something simple and homogeneous, like the array of double precision numbers in the heat diffusion example. Top down functional decomposition is not a good strategy for the overall design of more complex programs, such as those that are considered in Parts IV and V. However, it does reappear there too, in a minor role, when defining the individual "behaviours of objects".

Although the approach is limited, it is simple and it does apply to a very large number of simple programs. So, it is worth learning this design approach at least as a beginning.

The first section in this chapter simply summarizes materials from earlier examples using them to illustrate a basic strategy for program development. The second section looks briefly at some of the ways that you can document design decisions. Here, simple textual documentation is favoured.

15.1 TOP DOWN FUNCTIONAL DECOMPOSITION

As it says, you start the top with "program", decompose it into functions, then you iterate along the list of functions going down one level into each to decompose into

auxiliary functions. The process is repeated until the auxiliary functions are so simple that they can be coded directly.

Note the focus on functions. You don't usually have to bother much about the data because the data will be simple – a shared array or something similar.

Beginning

You begin with a phrase or one sentence summary that defines the program:

- The program models a two-dimensional heat diffusion experiment. e.g..
- The program plays the game of "hangman".

and try to get a caricature sketch of the `main()` function. This should be some slight variation of the code at the start of this chapter.

Figure 15.1 illustrates initial decompositions for `main()` in the two example programs.

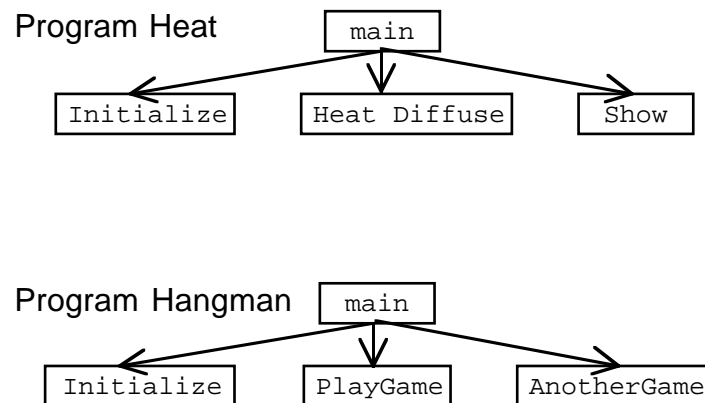


Figure 15.1 From program specification to initial functional decomposition.

The sketched functions are:

Program Heat's `main()`:

```

get iteration limit and print frequency
call Initialize() to initialize grid
loop
    call HeatDiffuse() to update values in grid
    if time to print, call Show()
  
```

Program Hangman's `main()`:

```

Initialize()
  
```

```
do
    PlayGame()
while AnotherGame()
```

You aim for this first stage is to get this initial sketch for `main()` and a list of "top level functions", each of which should be characterized by a phrase or sentence that summarizes what it does:

Products of beginning step

```
AnotherGame()
    Get and use a Yes/No input from user,
    return "true" if Yes was input

PlayGame()
    Organize the playing of one complete hangman game

HeatDiffuse()
    Recalculate temperature at each point on grid.
```

You now consider each of these top-level functions in isolation. It is all very well to say "Organize playing of game" or "Recalculate temperatures" but what do these specifications really mean.

Second step

It is essentially the same process as in the first step. You identify component operations involved in "organizing the playing of the game" or whatever, and you work out the sequence in which these operations occur. This gives you a caricature sketch for the code of the top-level function that you are considering and a list of second level functions. Thus, `HeatDiffuse()` gets expanded to:

```
HeatDiffuse
    get copy of grid values
    double loop
    for each row do
        for each col do
            using copy work out average temp.
                of environment of grid pt.
            calculate new temp based on current
                and environment
            store in grid
    reset centre point to flame temperature
```

with a number of additional functions identified: `CopyGrid()`, `AverageofNeighbors()`, and `NewTemperature()`.

The products of this design step are once again the sketches of functions and the lists of the additional auxiliary functions together with one sentence specifications for each:

```
CopyGrid()
    Copy contents of grid into separate data area.

AverageofNeighbors()
```

For a given grid point, this finds the average of the temperatures of the eight neighboring points.

N-steps This process is repeated until no more functions need be introduced. The relationships amongst the functions can be summarized in a "call graph" like that shown in Figure 15.2.

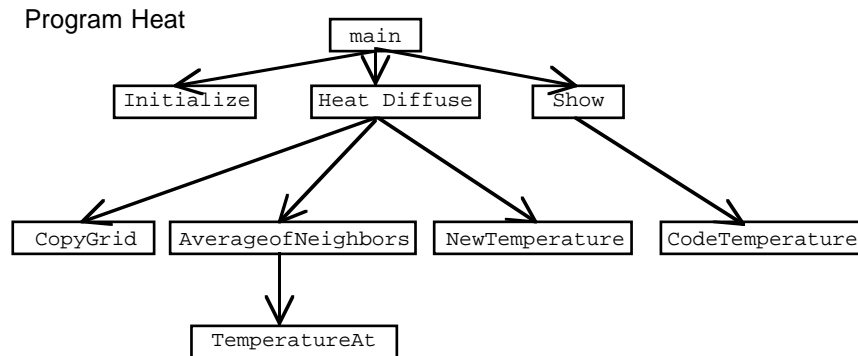


Figure 15.2 "Call graph" for a program.

Sorting out data and communications

The next stage in the design process really focuses on data. You have to resolve how the functions communicate and decide whether they share access to any global (or filescope) data. For example, you may have decided that function `TemperatureAt()` gets the temperature at a specified grid point, but you have still to determine how the function "knows" which point and how it accesses the grid.

The "input/output" argument lists of all the functions, together with their return types should now be defined. A table summarizing any filescope or global data should be made up.

Planning test data

Generally, you need to plan some tests. A program like Hangman does not require any elaborate preplanned tests; the simple processing performed can be easily checked through interaction with the program. The Heat program was checked by working out in advance the expected results for some simple cases (e.g. the heated point at 1000°C and eight neighbors at 25°C) and using the debugger to check that the calculated "new temperatures" were correct. Such methods suffice for simple programs but as you get to move elaborate programs, you need more elaborate preplanned tests. These should be thought about at this stage, and a suitable test plan composed.

Finalising the design

The final outputs that you want from the design process will include:

1. A sketch for `main()` summarising the overall processing sequence.
2. A list of function prototypes along with one sentence descriptions of what these functions do.


```
char CodeTemperature(double temp);  
    Converts temperature to a letter code; different  
    letters correspond to different temperature ranges.  
  
void Show(const Grid g);  
    Iterates over grid printing contents row by row,  
    uses CodeTemperature() to convert temp. to letter.
```

3. A list of any typedef types (e.g. `typedef double Grid[kROWS][kCOLS]`) and a list defining constants used by the program.
4. A table summarizing global and filescope data.
5. A more detailed listing of the functions giving the pseudo-code outlines for each.
6. A test plan.

It is only once you have this information that it becomes worth starting coding.

Although you will often complete the design for the entire program and then start on implementation, if the program is "large" like the Hangman example then it may be worth designing and implementing a simplified version that is then expanded to produce the final product.

15.2 DOCUMENTING A DESIGN

For most programs, the best documentation will be the design outlines just described. You essentially have two documents. The first contains items 1...4. It is a kind of executive summary and is what you would provide to a manager, to coworkers, or to the teaching assistant who will be marking your program. The second document contains the pseudo-code outlines for the functions. This you will use to guide your implementation. Pseudo-code is supposed to be easier to read than actual code, so you keep this document for use by future "maintenance programmers" who may have to implement extensions to your code (they should also be left details of how to retest the code to check that everything works after alterations).

This documentation is all textual. Sometimes, diagrams are required. Call graphs, like that in Figure 15.2, are often requested. While roughly sketched call graphs are useful while you are performing the design process and need to keep records as you move from stage to stage, they don't really help that much in documenting large programs. Figure 15.3 illustrates some of the problems with "call graphs".

The call graph for the sort program using Quicksort may be correct but it really fails to capture much of the behaviour of that recursive system. The call graph for the Hangman program is incomplete. But even the part shown is overwhelming in its complexity.

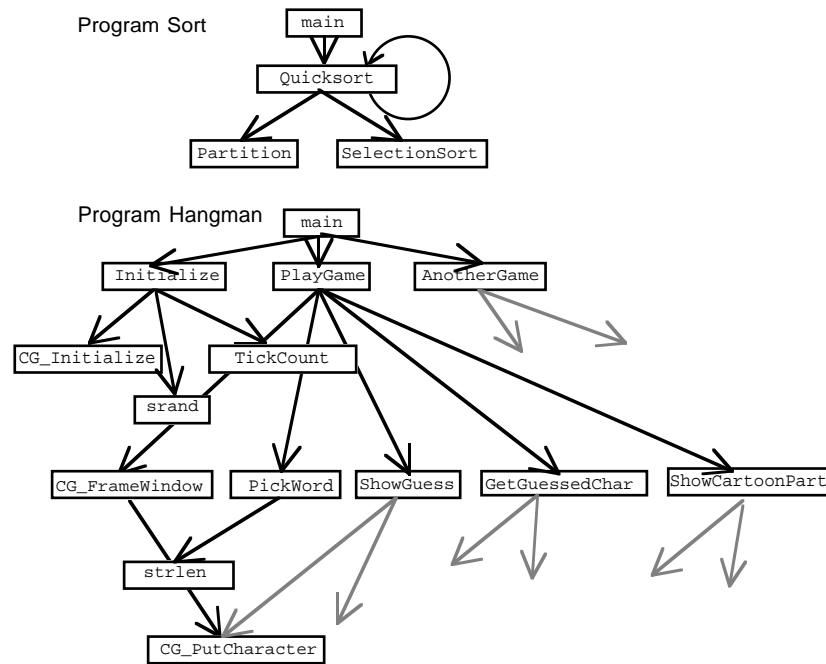


Figure 15.3 Further examples of "call graphs".

Psychologists estimate that the human brain can work with about seven items of information; if you attempt to handle more, you forget or confuse things. A complete call graph has too much information present and so does not provide a useful abstract overview.

You could break the graph down into subparts, separating `PlayGame`'s call graph from the overall program call graph. But such graphs still fail to convey any idea of the looping structure (the fact that some functions are called once, others many times).

Overall, call graph diagrams aren't a particularly useful form of documentation.

Sometimes, "flowcharts" are requested rather than pseudo-code outlines for functions. "Flowcharts" are old; they are contemporary with early versions of FORTRAN. Flowcharting symbols correspond pretty much to the basic programming constructs of that time, and lack convenient ways of representing multiway selections (`switch()` statements etc).

At one time, complete programs were diagrammed using flowcharts. Such program flowcharts aren't that helpful, for the same reason that full call graphs aren't that helpful. There is too much detail in the diagram, so it doesn't effectively convey a program's structure.

It is possible to "flowchart" individual functions, an example in flowcharting style is shown in Figure 15.4. Most people find pseudo-code outlines easier to understand.

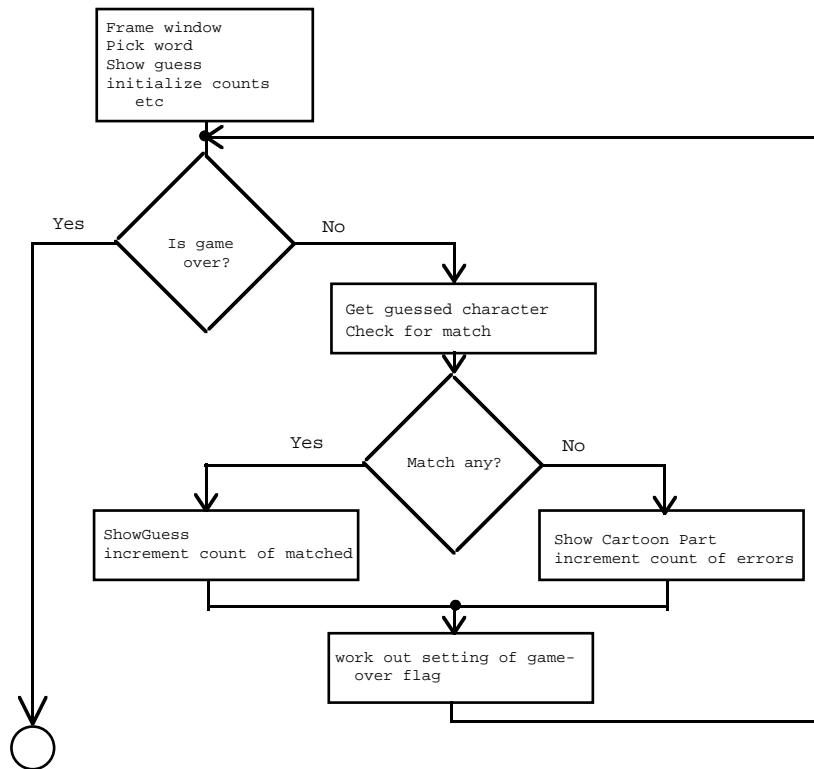


Figure 15.4 "Flowchart" representation, an alternative to a pseudo-code outline of a function.

On the whole, diagrams aren't that helpful. The lists of functions, the pseudo-code outlines etc are easier to understand and so it isn't worth trying to generate diagrammatic representations of the same information.

However, you may have access to a CASE ("Computer Assisted Software Engineering") program. These programs have a user interface somewhat similar to a drawing package. There is a palette of tools that you can use to place statement sequences, conditional tests, iterative constructs etc. Dialogs allow you to enter details of the operations involved.

The CASE tool can generate the final function prototypes, and the code for their implementation, from the information that you provide in these dialogs. If you use such a program, you automatically get a diagrammatic representation of your program along with the code.

16 Enum, Struct, and Union

This chapter introduces three simple kinds of programmer defined data types.

You aren't limited to the compiler provided char, int, double data types and their derivatives like arrays. You can define your own data types. Most of Part IV of this text is devoted to various forms of programmer defined type. Here we introduce just three relatively simple kinds. These are almost the same as the equivalents provided in C; in contrast, C has no equivalent to the more sophisticated forms of programmer defined type introduced in Part IV.

Enumerated types, treated in section 16.1, are things that you should find useful in that they can improve the readability of your programs and they allow the compiler to do a little extra type checking that can eliminate certain forms of error. Although useful, enumerated types are not that major a factor in C++ programming.

Structs are the most important of the three language elements introduced here, section 16.2. They do have wider roles, but here we are interested in their primary role which is the grouping of related data.

Unions: treat them as a "read only" feature of C++. You will sometimes see unions being employed in library code that you use, but it is unlikely that you will find any real application for unions in the programs that you will be writing.

16.1 ENUMERATED TYPES

You often need to use simple integer constants to represent domain specific data. For example, suppose you needed to represent the colour of an automobile. You could have the following:

```
const int  cRED    = 0;
const int  cBLUE   = 1;
...

int auto_colour;
```

```
auto_colour = cBLUE;
```

This is quite workable. But there are no checks. Since variable `auto_colour` is an integer, the following assignments are valid:

```
auto_colour = -1;
...
auto_colour = rand();
```

But of course, these statements lose the semantics; variable `auto_colour` doesn't any longer represent a colour, it is just an integer.

It is nice to be able to tell the compiler:

"Variable `auto_colour` is supposed to represent a colour, that is one of the defined set of choices RED, BLUE,"

and then have the compiler check, pretty thoroughly, that `auto_colour` is only used in this way throughout the code.

enums and type checking

This is the role of `enums` or enumerated types. If you think how they are actually implemented, they are just an alternative way of declaring a set of integer constants and defining some integer variables. But they also introduce new distinct types and allow the compiler to do type checking. It is this additional type checking that makes `enums` worthwhile.

The following is a simple example of an enum declaration:

```
enum Colour { eRED, eBLUE, eYELLOW, eGREEN, eSILVERGREY,
              eBURGUNDY };
```

Naming convention for enums

The entries in the enumeration list are just names (of constant values). The same rules apply as for any other C++ names: start with a letter, contain letters digits and underscores. However, by convention, the entries in the enum list should have names that start with 'e' and continue with a sequence of capital letters. This makes enum values stand out in your code.

With `Colour` now defined, we can have variables of type `Colour`:

```
Colour auto_colour;
...
auto_colour = eBURGUNDY;
```

The compiler will now reject things like `auto_colour = 4`. Depending on the compiler you are using you may get just "Warning – anachronism", or you may get an error (really, you should get an error).

enumerators

What about the "enumerators" `eRED`, `eBLUE` etc? What are they?

Really, they are integers of some form. The compiler may chose to represent them using shorts, or as unsigned chars. Really, it isn't any of your business how your compiler represents them.

The compiler chooses distinct values for each member of an enumeration. Normally, the first member has value 0, the second is 1, and so forth. So in this example, `eRED` would be a kind of integer constant 0, `eSILVERGREY` would be 4.

Note the effect of the type checking:

```
auto_colour = 4;    // Wrong, rejected or at least
                   //          warned by compiler
auto_colour = eSILVERGREY; // Fine, set auto_colour to
                   //          (probably) the value 4
```

It isn't the values that matter, it is the types. The value 4 is an integer and can't be directly assigned to a `Colour` variable. The constant `eSILVERGREY` is a `Colour` enumerator and can be assigned to a `Colour` variable.

You can select for yourself the integer values for the different members of the enumeration, provided of course that you keep them all distinct. You won't have cause to do this yourself; but you should be able to read code like:

```
enum MagicEnum { eMERLIN = 17, eGANDALF = 103, eFRED = 202 };
```

Treat this as one of those "read only" features of C++. It is only in rare circumstances that you want to define specific values for the members of the enumeration. (Defining specific values means that you aren't really using the enum consistently; at some places in your code you intend to treat enums as characters or integers.)

Output of enums

Enums are fine in the program, but how do you get them transferred using input and output statements?

Well, with some difficulty!

An enum is a form of integer. You can try:

```
cout << auto_colour;
```

and you might get a value like 0, or 3 printed. More likely, the compiler will give you an error message (probably something rather obscure like "ambiguous reference to overloaded operator function"). While the specific message may not be clear, the compiler's unhappiness is obvious. It doesn't really know how you want the `enum` printed.

You can tell the compiler that it is OK to print the enum as an integer:

```
cout << int(auto_colour);
```

The function like form `int(auto_colour)` tells the compiler to convert the data value `auto_colour` to an integer. The statement is then `cout << integer` which the compiler knows to convert into a call to a `PrintInteger()` routine. (The compiler doesn't need to generate any code to do the conversion from `enum` to integer. This conversion request is simply a way for the programmer to tell the compiler that here it is intended that the `enum` be regarded as just another integer).

Printing an `enum` as an integer is acceptable if the output is to a file that is going to be read back later by the program. Human users aren't going to be pleased to get output like:

```
Engine:    1.8 litre
Doors:     4
Colour:    5
```

If you are generating output that is to be read by a human user, you should convert the `enum` value into an appropriate character string. The easiest way is to use a `switch` statement:

```
switch(auto_colour) {
eRED:      cout << "Red"; break;
eBLUE:     cout << "Blue"; break;
...
eBURGUNDY: cout << "Burgundy"; break;
}
```

Input

If your program is reading a file, then this will contain integer values for `enums`; for example, the file could have the partial contents

```
1.8 4      5
```

for an entry describing a four door, burgundy coloured car with 1.8 litre engine. But you can't simply have code like:

```
double engine_size;
int num_doors;
Colour auto_colour;
...
input >> engine_size;
input >> num_doors ;
input >> auto_colour;
```


The compiler gets stuck when it reaches `input >> auto_colour;`. The compiler's translation tables let it recognize `input >> engine_size` as a form of "input gives to double" (so it puts in a call to a `ReadDouble()` routine), and similarly `input >> num_doors` can be converted to a call to `ReadInt()`. But the compiler's standard translation tables say nothing about `input >> Colour`.

The file contains an integer; so you had better read an integer:

```
int temp;
input >> temp;
```

Now you have an integer value that you hope represents one of the members of the `enum Colour`. Of course you must still set the `Colour` variable `auto_colour`. You can't simply have an assignment:

```
auto_colour = temp;
```

because the reason we started all of this fuss was to make such assignments illegal!

Here, you have to tell the compiler to suspend its type checking mechanisms and trust you. You can say *"I know this integer value will be a valid Colour, do the assignment."* The code is

```
auto_colour = Colour(temp);
```

Input from file will use integers, but what of input from a human user?

Normally, if you are working with enumerated types like this, you will be prompting the user to make a selection from a list of choices:

```
Colour GetColour()
{
    cout << "Please enter preferred colour, select from "
         << endl;
    cout << "1\tRed" << endl;
    cout << "2\tBlue" << endl;
    ...
    cout << "6\tBurgundy" << endl;

    for(;;) {
        int choice;
        cin >> choice;
        switch(choice) {
case 1:    return eRED;
case 2:    return eBLUE;
        ...
case 6:    return eBURGUNDY;
        }
        cout << "There is no choice #" << choice << endl;
    }
}
```

```

        cout << "Please select from 1 Red, 2 Blue ... "
              " 6, Burgundy" << endl;
    }
}

```

(Note, internally `eRED` may be 0, `eBLUE` may be 1 etc. But you will find users generally prefer option lists starting with 1 rather than 0. So list the choices starting from 1 and make any adjustments necessary when converting to internal form.)

Other uses of enums

You do get cases like `Colour auto_colour` where it is appropriate to use an enumerated type; but they aren't that common (except in text books). But there is another place where enums are very widely used.

Very often you need to call a routine specifying a processing option from a fixed set:

`DrawString` – this function needs to be given the string to be displayed and a style which should be one of

```

    Plain
    Bold
    Italic
    Outline

```

`AlignObjects` – this function needs to be given an array with the object identifiers, a count, and an alignment specification which should be one of

```

    LeftAligned
    Centred
    RightAligned

```

You can define integer constants:

```

const int cPLAIN = 0;
const int cBOLD = 1;
...

```

and have an integer argument in your argument list:

```

void DrawString(char txt[], int style);

```

but the compiler can't check that you only use valid styles from the set of defined constants and so erroneous code like

```

DrawString("silly", 12345);

```

gets through the compiler to cause problems at run time.

But, if you code using an enumerated type, you do get compile time checks:

```
enum TextStyles { ePLAIN, eBOLD, eITALIC, eOUTLINE };

void DrawString(char txt[], TextStyles style);
```

Now, calls like:

```
DrawString("Home run", eBOLD);
```

are fine, while erroneous calls like

```
DrawString("Say what?", 59);
```

are stomped on by the compiler (again, you may simply get a warning, but it is more likely that you will get an error message about a missing function).

16.2 STRUCTS

It is rare for programs to work with simple data values, or even arrays of data values, that are individually meaningful. Normally, you get groups of data values that belong together.

*The need for
"structs"*

Let's pick on those children again. This time suppose we want records of children's heights in cm, weights in kilos, age (years and months), and gender. We expect to have a collection of about one thousand children and need to do things like identify those with extreme heights or extreme weights.

We can simply use arrays:

```
const int  kMAXCHILDREN = 1000;

double     heights[kMAXCHILDREN];
double     weights[kMAXCHILDREN];
int        years[kMAXCHILDREN];
int        months[kMAXCHILDREN];
char       gender[kMAXCHILDREN];
...
// read file with data, file terminated by sentinel data
// value with zero height
count = 0;
infile >> h;
while(h > 0.0) {
    heights[count] = h;
    infile >> weights[count] >> years[count] >>
        months[count] >> gender[count];
```

```

        count++;
        infile >> h;
    }
    ...

```

Now `heights[5]`, `weights[5]`, ..., `gender[5]` all relate to the same child. But if we are using arrays, this relationship is at most implicit.

There is no guarantee that we can arrange that the data values that logically belong together actually stay together. After all, there is nothing to stop one from sorting the `heights` array so that these values are in ascending order, at the same time losing the relationship between the data value in `heights[5]` and those in `weights[5]` ... `gender[5]`.

***Have to be able to
group related data
elements
Declaring structs***

A programming language must provide a way for the programmer to identify groups of related data. In C++, you can use `struct`.

A C++ `struct` declaration allows you to specify a grouping of data variables:

```

struct Child {
    double height;
    double weight;
    int    years;
    int    months;
    char   gender;
};

```

After reading such a declaration, the compiler "knows what a `Child` is"; for the purposes of the rest of the program, the compiler knows that a `Child` is a data structure containing two doubles, two integers, and a character. The compiler works out the basic size of such a structure (it would probably be 25 bytes); it may round this size up to some larger size that it finds more convenient (e.g. 26 bytes or 28 bytes). It adds the name `Child` to its list of type names.

This grouping of data is the primary role of a `struct`. In C++, `structs` are simply a special case of classes and they can have perform roles other than this simple grouping of data elements. However it is useful to make a distinction. In the examples in this book, `structs` will only be used as a way of grouping related data elements.

***"record", "field",
"data member"***

The term "record" is quite often used instead of `struct` when describing programs. The individual data elements within a `struct` are said to be "fields", or "data members". The preferred C++ terminology is "data members".

***Defining variables of
struct types***

A declaration doesn't create any variables, it just lets the compiler know about a new type of data element that it should add to the standard `char`, `long`, `double` etc. But, once a `struct` declaration has been read, you can start to define variables of the new type:

***Definitions of some
variables of struct
type***

```

Child    cute;
Child    kid;
Child    brat;

```

and arrays of variables of this type:

```
Child    surveyset[kMAXCHILDREN];
```

Each of these `Child` variables has its own doubles recording height and weight, its own ints for age details, and a char gender flag.

The definition of a variable of struct type can include the data needed to initialize its data members:

Initialization of structs

```
Child example = {
    125.0, 32.4, 13, 2, 'F'
};
```

An instance of a struct can be defined along with the declaration:

```
struct Rectangle {
    int left, top;
    int width, height;
} r1, r2, r3;
```

This declares the form of a `Rectangle` and defines three instances. This style is widely use in C programs, but it is one that you should avoid. A declaration should introduce a new data type; you should make this step separate from any variable definitions. The construct is actually a source of some compile-time errors. If you forget the ';' that terminates a structure declaration, the compiler can get quite lost trying to interpret the next few program elements as being names of variables (e.g. the input `struct Rect { ... } struct Point {...} int main() { ... }` will make a compiler quite unhappy).

Programs have to be able to manipulate the values in the data members of a struct variable. Consequently, languages must provide a mechanism for referring to a particular data member of a given struct variable.

Most programming languages use the same approach. They use compound names made up of the variable name and a qualifying data member name. For example, if you wanted to check whether `brat`'s height exceeded a limit, then in C++ you would write:

Accessing data members of a variable of a struct type
Fields of a variable identified using compound names

```
if(brat.height > h_limit)
    ...
```

Similarly, if you want to set `cute`'s weight to 35.4 kilos, you would write:

```
cute.weight = 35.4;
```

Most statements and expressions will reference individual data members of a struct, but assignment of complete structures is permitted:

Assignment of structs

```
Child Tallest;
```

```
Tallest.height = 0;
for(int i = 0; i < NumChildren; i++)
    if(surveyset[i].height > Tallest.height)
        Tallest = surveyset[i];
```

Compilers generally handle such assignments by generating code using a "blockmove". A blockmove (which is often an actual instruction built into the machine hardware) copies a block of bytes from one location to another. The compiler knows the size of the structs so it codes a blockmove for the appropriate number of bytes.

*Struct and enum
declarations in C
libraries*

In C++, a struct declaration introduces a new type. Once you have declared:

```
struct Point {
    int x;
    int y;
};
```

You can define variables of type `Point`:

```
Point p1, p2;
```

In C, struct (and enum) declarations don't make the struct name (or enum name) a new type. You must explicitly tell the compiler that you want a new type name to be available. This is done using a `typedef`. There are a variety of styles. Two common styles are:

typedef

```
struct Point {
    int x;
    int y;
};

typedef struct Point Point;
```

or:

```
typedef struct _xpt {
    int x;
    int y;
} Point;
```

Similarly, enums require typedefs.

```
enum Colour { eRED, eBLUE, eGREEN };

typedef enum Colour Colour;
```

You will see such typedefs in many of the C libraries that you get to use from your C++ programs.

A function can have arguments of struct types. Like simple variables, structs can be passed by value or by reference. If a struct is passed by value, it is handled like an assignment – a blockmove is done to copy the bytes of the structure onto the stack. Generally, because of the cost of the copying and the need to use up stack space, you should avoid passing large structs by value. If a function uses a struct as an "input parameter", its prototype should specify the struct as const reference, e.g.:

Structs and functions

```
void      PrintChildRecord(const struct& theChild)
{
    cout << "Height " << theChild.height << ...
    ...
}
```

Functions can have structs as their return values. The following illustrates a function that gets an "car record" filled in:

```
struct car {
    double engine_size;
    int    num_doors;
    Colour auto_colour;
};

car GetAutoDetails()
{
    car    temp;
    cout << "Select model, GL (1), GLX (2), SX (3), TX2(4) : "
;
    int model;
    cin >> model;
    while((model < 1) || (model >4)) {
        cout << "Model value must be in range 1...4" << endl;
        cout << "Select model : "
        cin >> model;
    }
    switch(model) {
case 1:    temp.engine_size = 1.8; temp.num_doors = 3; break;
case 2: ...
    }
    temp.colour = GetColour();

    return temp;
}
```

*Function with a
struct as a returned
value
Local variable of
return type defined*

*Data entered into
fields of local struct
variable*

*return the local struct
as the value*

Although this is permitted, it should not be overused. Returning a struct result doesn't matter much with a small structure like struct car, but if your structures are large this style becomes expensive both in terms of space and data copying operations.

Code using the GetAutoDetails() function would have to be something like:

```
car purchasers_choice;
```

```
...
purchasers_choice = GetAutoDetails();
```

The instructions generated would normally be relatively clumsy. The stack frame setup for the function would have a "return value area" sufficient in size to hold a `car` record; there would be a separate area for the variable `temp` defined in the function. The return statement would copy the contents of `temp` into the "return value area" of the stack frame. The data would then again be copied in the assignment to `purchasers_choice`.

If you needed such a routine, you might be better to have a function that took a reference argument:

```
void GetAutoDetails(struct car& temp)
{
    cout << "Select model, GL (1), GLX (2), SX (3), TX2(4) : "
;
    ...
    temp.colour = GetColour();
    return ;
}
```

with calling code like:

```
car purchasers_choice;
...
GetAutoDetails(purchasers_choice);
```

16.3 UNIONS

Essentially, unions define a set of different interpretations that can be placed on the data content area of a struct. For you, "unions" should be a "read-only" feature of C++. It may be years before you get to write code where it might be appropriate for you to define a new union. However, you will be using libraries of C code, and some C++ libraries, where unions are employed and so you need to be able to read and understand code that utilizes unions.

Unions are most easily understood from real examples. The following examples are based on code from Xlib. This is a C library for computers running Unix (or variations like Mach or Linux). The Xlib library provides the code needed for a program running on a computer to communicate with an X-terminal. X-terminals are commonly used when you want a multi-window style of user interface to Unix.

An X-terminal is a graphics display device that incorporates a simple microprocessor and memory. The microprocessor in the X-terminal does part of the work of organizing the display, so reducing the computational load on the main computer.

When the user does something like move the mouse, type a character, or click an action button, the microprocessor packages this information and sends it in a message to the controlling program running on the main computer.

In order to keep things relatively simple, all such messages consist of a 96 byte block of data. Naturally, different actions require different data to be sent. A mouse movement needs a report of where the mouse is now located, a keystroke action needs to be reported in terms of the symbol entered.

Xlib-based programs use `XEvent` unions to represent these 96 byte blocks of data. The declaration for this union is

```
typedef union _XEvent {
    int type;
    XAnyEvent xany;
    XButtonEvent xbutton;
    XMotionEvent xmotion;
    XCreateWindowEvent xcreatewindow;
    ...
    ...
} XEvent;
```

union declaration

This declaration means that an `XEvent` may simply contain an integer (and 92 bytes of unspecified data), or it may contain an `XAnyEvent`, or it may contain an `XButtonEvent`, or There are about thirty different messages that an Xterminal can send, so there are thirty different alternative interpretations specified in the union declaration.

Each of these different messages has a struct declaration that specifies the data that that kind of message will contain. Two of these structs are:

```
typedef struct {
    int type;
    unsigned long serial;
    Bool send_event;
    Display *display;
    Window window;
    Window root;
    Window subwindow;
    Time time;
    int x, y;
    int x_root, y_root;
    unsigned int state;
    unsigned int button;
    Bool same_screen;
} XButtonEvent;
```

*Declaration of
alternative structs
that can be found in
an XEvent*

```
typedef struct {
    int type;
    unsigned long serial;
    Bool send_event;
```

```
Display *display;
Window window;
int x, y;
int width, height;
int border_width;
Bool override_redirect;
} XCreateWindowEvent;
```

As illustrated in Figure 16.1, the first part of any message is a type code. The way that the rest of the message bytes are used depends on the kind of message.

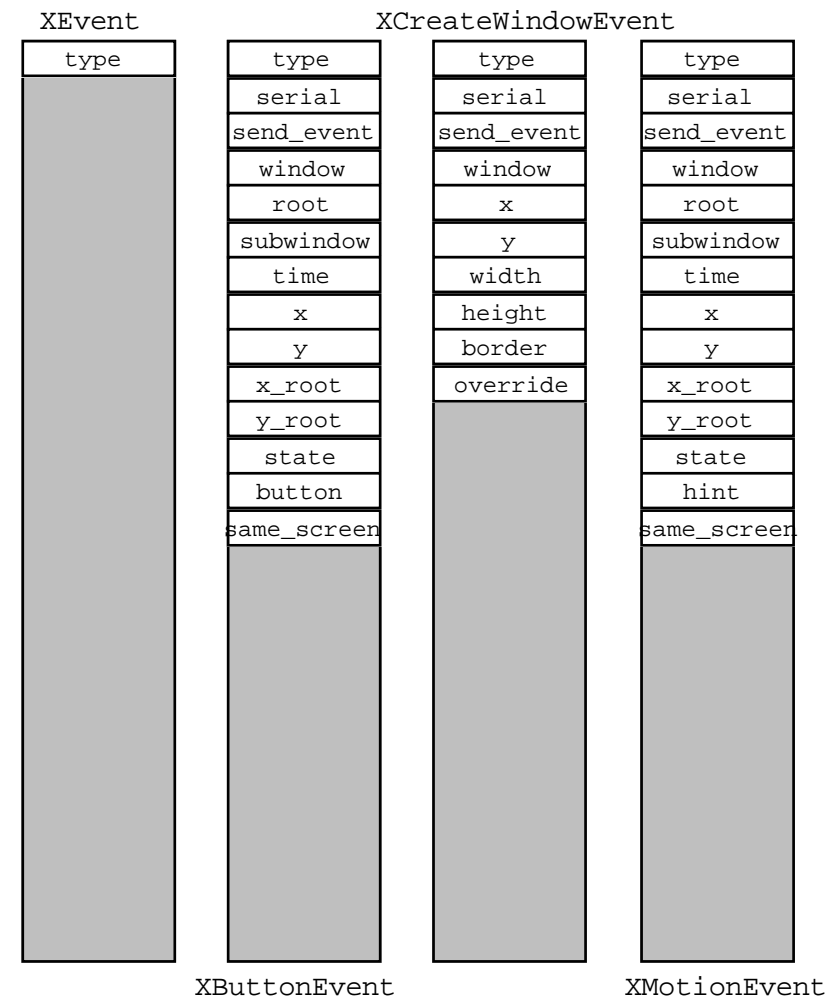


Figure 16.1 XEvents – an example of a union from the Xlib library.

If you have an `XEvent` variable `ev`, you can access its type field using the "." operator just like accessing a data field in a normal structure:

```
XEvent    ev;
...
switch(ev.type) {
    ...
}
```

If you know that `ev` really encodes an `XCreateWindowEvent` and you want to work out the area of the new window, you can use code like:

```
area = ev.xcreatewindow.width * eve.xcreatewindow.height;
```

*Doubly qualified
names to access data
members of variants
in union*

The appropriate data fields are identified using a doubly qualified name. The variable name is qualified by the name of the union type that is appropriate for the kind of record known to be present (so, for an `XCreateWindowEvent`, you start with `ev.xcreatewindow`). This name is then further qualified by the name of the data member that should be accessed (`ev.xcreatewindow.width`).

A programmer writing the code to deal with such messages knows that a message will start with a type code. The Xlib library has a series of #defined constants, e.g. `ButtonPress`, `DestroyNotify`, `MotionNotify`; the value in the type field of a message will correspond to one of these constants. This allows messages from an Xterminal to be handled as follows:

```
XEvent eV;
...
/* Code that gets calls the Unix OS and
gets details of the next message from the Xterminal
copied into eV */
...
switch(ev.type) {
case ButtonPress:
    /* code doing something depending on where the button was
    pressed, access using xbutton variant from union */

    if((ev.xbutton.x > x_low) && (ev.xbutton.x < x_high) && ...

        break;
case MotionNotify:
    /* user has moved pointer, get details of when this
    happened
    and decide what to do, access using xmotion variant from
    union */

    thetime = ev.xmotion.time;
    ...
```

```
        break;

case CreateNotify:
    /* We have a new window, code here that looks at where and
       what size, access using xcreatewindow variant of union */
    int hpos = ev.xcreatewindow.x;
    ...
    break;

...
}
```

17 Examples using structs

This chapter contains a few simple examples illustrating the use of structs in programs.

In future programs, you will use structs mainly when communicating with existing C libraries. For example, you will get to write code that uses the graphics primitives on your system. The "applications programmer interface" (API) for the graphics will be defined by a set of C functions (in some cases based on an earlier interface defined by a set of Pascal functions). If you are programming on Unix, you will probably be using the Xlib graphics package, on an Apple system you would be using Quickdraw, and on an Intel system you would use a Windows API. The functions in the API will make extensive use of simple structs. Thus, the Xlib library functions use instances of the following structs:

Using structs with existing libraries

```
typedef struct {
    short x, y;
} XPoint;

typedef struct {
    short x, y;
    unsigned short width, height;
} XRectangle;
```

and

```
typedef struct {
    short x, y;
    unsigned short width, height;
    short angle1, angle2;
} XArc;
```

So, if you are using the Xlib library, your programs will also define and use variables that are instances of these struct types.

For new application-specific code you will tend to use variables of class types more often than variables of struct types.

Your own structs?

The examples in this chapter are intended merely to illustrate how to declare struct types, use instances of these types, access fields and so forth. The example programs in section 17.3, introduce the idea of a "file of records". These programs have a single struct in memory and a whole set of others in a disk file. The struct in memory gets loaded with data from a requested record on disk. Record files are extremely common in simple business data processing applications, and represent a first step towards the more elaborate "databases" used in industry. You will learn more about "files of records" and "databases" in later subjects. While you can continue to use C/C++ when working with database systems, you are more likely to use specialized languages like COBOL and SQL.

17.1 REORDERING THE CLASS LIST AGAIN

The example in section 13.2 illustrated a "sorting" algorithm that was applied to reorder pupil records. The original used two arrays, `pupils` and `marks`:

```
typedef char Name[40];

Name pupils[] = {
    "Armstrong, Alice S.",
    ...
    "Zarra, Daniela"
};

int marks[] = {
    57,
    ...
    78
};
```

But the data in the two arrays "belong together" – Miss Armstrong's mark is 57 and she shouldn't be separated from it.

This is a typical case where the introduction of a simple struct leads to code that reflects the problem structure more accurately (and is also slightly clearer to read). The struct has to package together a name and a mark:

```
struct PupilRec {
    Name    fName;
    int     fMark;
};
```

(Naming conventions again: use 'f' as the first character of the name of a data member in all structs and classes that are defined for a program.)

With this struct declared, an initialized array of `PupilRecs` can be defined:

```
PupilRec JuniorYear[] = {
    { "Armstrong, Alice S.",      57 } ,
    { "Azur, Hassan M.",         64 } ,
    { "Bates, Peter",            33 } ,
    ...
    { "Zarra, Daniela",          81 }
};
```

The function `MakeOrderedCopy()` can be rewritten to use these `PupilRec` structs:

```
void MakeOrderedCopy(const PupilRec orig[],int num,
    PupilRec reord[])
{
    int mark_count[101];

    for(int i = 0; i < 101; i++)
        mark_count[i] = 0;

    // Count number of occurrences of each mark
    for(i = 0; i < num; i++) {
        int mark = orig[i].fMark;
        mark_count[mark]++;
    }
    // Make that count of number of pupils with marks less than
    // or equal to given mark
    for(i=1; i<101; i++)
        mark_count[i] += mark_count[i-1];

    for(i=num - 1; i >= 0; i--) {
        int mark = orig[i].fMark;
        int position = mark_count[mark];
        position--; // correct to zero based array
        // copy data
        reord[position] = orig[i];
        mark_count[mark]--;
    }
}
```

*Arrays of structs as
"input" and
"output" parameters*

*Accessing data
member of one of
structs in the array*

Assignment of structs

The function takes two arrays of `PupilRec` structs. The `orig` parameter is the array with the records in alphabetic order; it is an "input parameter" so it is specified as `const`. The `reord` parameter is the array that is filled with the reordered copy of the data.

An expression like:

```
orig[i].fMark
```

illustrates how to access a data member of a chosen element from an array of structures.

Note the structure assignment:

```
reord[position] = orig[i];
```

The compiler "knows" the size of structs so allows struct assignment while assignment of arrays is not allowed.

17.2 POINTS AND RECTANGLES

The following structs and functions illustrate the functionality that you will find in the graphics libraries for your system. Many of the basic graphics functions will use points and rectangles; for example, a line drawing function might take a "start point" and an "end point" as its parameters, while a `DrawOval()` function might require a rectangle to frame the oval.

The graphics packages typically use short integers to represent coordinate values. A "point" will need two short integer data fields:

```
struct Point {
    short fx;
    short fy;
};
```

A struct to represent a rectangle can be defined in a number of alternative ways, including:

```
struct Rectangle {
    short ftop;
    short fleft;
    short fwidth;
    short fheight;
};
```

and

```
struct Rectangle {
    Point fcorner;
    short fwidth;
    short fheight;
};
```

Provided that `struct Point` has been declared before `struct Rectangle`, it is perfectly reasonable for `struct Rectangle` to have data members that are instances of type `Point`. The second declaration will be used for the rest of the examples in this section.

The graphics libraries define numerous functions for manipulating points and rectangles. The libraries would typically include variations of the following functions:


```
int Point_In_Rect(const Point& pt, const Rectangle& r);
    Returns "true" (1) if point pt lies with Rectangle r.

int Equal_Points(const Point& p1, const Point& p2);
    Returns "true" if points p1 and p2 are the same.

void Add_Point(const Point& p1, const Point& p2, Point& res);
    Changes the "output parameter" res to be the 'vector sum'
    of points p1 and p2.

Point MidPoint(const Point& p1, const Point& p2);
    Returns the mid point of the given points.

int ZeroRect(const Rectangle& r)
    Return "true" if r's width and height are both zero.

Rectangle UnionRect(const Rectangle& r1, const Rectangle& r2);
    Returns the smallest circumscribing rectangle of the given
    rectangles r1 and r2.

Rectangle IntersectRect(const Rectangle& r1,
    const Rectangle& r2);
    Returns a rectangle that represents the intersection of the
    given rectangles, or a zero rectangle if they don't
    intersect.

Rectangle Points2Rect(const Point& p1, const Point& p2);
    Returns a rectangle that includes both points p1 and p2
    within its bounds or at least on its perimeter
```

These functions are slightly inconsistent in their prototypes, some return structs as results while others have output parameters. This is deliberate. The examples are meant to illustrate the different coding styles. Unfortunately, it is also a reflection of most of the graphics libraries! They tend to lack consistency. If you were trying to design a graphics library you would do better to decide on a style; functions like `Add_Point()` and `MidPoint()` should either all have struct return types or all have output parameters. In this case, there are reasons to favour functions that return structs. Because points and rectangles are both small, it is acceptable for the functions to return structs as results. Code using functions that return structs tends to be a little more readable than code using functions with struct output parameters.

This example is like the "curses" and "menu selection" examples in Chapter 12. We need a header file that describes the facilities of a "points and rectangles" package, and an implementation file with the code of the standard functions. Just so as to illustrate the approach, some of the simpler functions will be defined as "inline" and their definitions will go in the header file.

The header file would as usual have its contents bracketed by conditional compilation directives:

```

#ifndef __MYCOORDS__
#define __MYCOORDS__

the interesting bits

#endif

```

As explained in section 12.1, these conditional compilation directives protect against the possibility of erroneous multiple inclusion.

The structs would be declared first, then the function prototypes would be listed. Any "inline" definitions would come toward the end of the file:

*Header file
"mycoords.h"*

```

#ifndef __MYCOORDS__
#define __MYCOORDS__

struct Point {
    short fx;
    short fy;
};

struct Rectangle {
    ...
};

int Point_In_Rect(const Point& pt, const Rectangle& r);

...

inline int Equal_Points(const Point& p1, const Point& p2)
{
    return ((p1.fx == p2.fx) && (p1.fy == p2.fy));
}

inline int ZeroRect(const Rectangle& r)
{
    return ((r.fwidth == 0) && (r.fheight == 0));
}

#endif

```

*inline functions
defined in the header
file*

"Inline" functions have to be defined in the header. After all, if the compiler is to replace calls to the functions by the actual function code it must know what that code is. When compiling a program in another file that uses these functions, the compiler only knows the information in the #included header.

The definitions of the other functions would be in a separate mycoords.cp file:

```
#include "mycoords.h"
```

```

int Point_In_Rect(const Point& pt, const Rectangle& r)
{
    // Returns "true" if point pt lies with Rectangle r.
    if((pt.fx < r.fcorner.fx) || (pt.fy < r.fcorner.fy))
        return 0;

    int dx = pt.fx - r.fcorner.fx;
    int dy = pt.fy - r.fcorner.fy;

    if((dx > r.fwidth) || (dy > r.fheight))
        return 0;

    return 1;
}

```

*Note multiply
qualified names of
data members*

A rectangle has a `Point` data member which itself has `int fx, fy` data members. The name of the data member that represents the x-coordinate of the rectangle's corner is built up from the name of the rectangle variable, e.g. `r1`, qualified by the name of its point data member `fcorner`, qualified by the name of the field, `fx`.

Function `Add_Point()` returns its result via the `res` argument:

```

void Add_Point(const Point& p1, const Point& p2, Point& res)
{
    // Changes the "output parameter" res to be the 'vector sum'
    // of points p1 and p2.
    res.fx = p1.fx + p2.fx;
    res.fy = p1.fy + p2.fy;
}

```

while `MidPoint()` returns its value via the stack (it might as well use `Add_Point()` in its implementation).

```

Point MidPoint(const Point& p1, const Point& p2)
{
    // Returns the mid point of the given points.
    Point m;
    Add_Point(p1, p2, m);
    m.fx /= 2;
    m.fy /= 2;

    return m;
}

```

Function `Points2Rect()` is typical of the three `Rectangle` functions:

```

Rectangle Points2Rect(const Point& p1, const Point& p2)
{
    // Returns a rectangle that includes both points p1 and p2
    // within its bounds or at least on its perimeter
}

```

```

Rectangle r;
int left, right, top, bottom;

if(p1.fx < p2.fx) { left = p1.fx; right = p2.fx; }
else { left = p2.fx; right = p1.fx; }

if(p1.fy < p2.fy) { top = p1.fy; bottom = p2.fy; }
else { top = p2.fy; bottom = p1.fy; }

r.fcorner.fx = left;
r.fcorner.fy = top;

r.fwidth = right - left;
r.fheight = bottom - top;

return r;
}

```

If you were developing a small library of functions for manipulation of points and rectangles, you would need a test program like:

```

#include <iostream.h>
#include "mycoords.h"

int main()
{
    Point p1;
    p1.fx = 1;
    p1.fy = 7;

    Rectangle r1;
    r1.fcorner.fx = -3;
    r1.fcorner.fy = -1;
    r1.fwidth = 17;
    r1.fheight = 25;

    if(Point_In_Rect(p1, r1))
        cout << "Point p1 in rect" << endl;
    else cout << "p1 seems to be lost" << endl;

    Point p2;
    p2.fx = 11;
    p2.fy = 9;
    Point p3;
    Add_Point(p1, p2, p3);

    cout << "I added the points, getting p3 at ";
    cout << p3.fx << ", " << p3.fy << endl;

    Point p4;
    p4.fx = 12;

```

```

    p4.fy = 16;

    if(Equal_Points(p3, p4))
        cout << "which is where I expected to be" << endl;
    else cout << "which I find surprising!" << endl;

    ...
    ...

    Rectangle r4 = UnionRect(r2, r3);
    cout << "I made rectangle r4 to have a corner at ";
    cout << r4.fcorner.fx << ", " << r4.fcorner.fy << endl;
    cout << "and width of " << r4.fwidth <<
        ", height " << r4.fheight << endl;

    Rectangle r5 = IntersectRect(r4, r1);
    cout << "I made rectangle r5 to have a corner at ";
    cout << r5.fcorner.fx << ", " << r5.fcorner.fy << endl;
    cout << "and width of " << r5.fwidth <<
        ", height " << r5.fheight << endl;

    return 0;
}

```

The test program would have calls to all the defined functions and would be organized to check the results against expected values as shown.

You may get compilation errors relating to `struct Point`. Some IDEs automatically include the system header files that define the graphics functions, and these may already define some other `struct Point`. Either find how to suppress this automatic inclusion of headers, or change the name of the structure to `Pt`.

17.3 FILE OF RECORDS

Rather than writing their own program, anyone who really wants to keep files of business records would be better off using the "database" component of one of the standard "integrated office packages". But someone has to write the "integrated office package" of the future, maybe you. So you had better learn how to manipulate simple files of records.

The idea of a file of records was briefly previewed in section 9.6 and is again illustrated in Figure 17.1.

A file record will be exactly the same size as a memory based struct. Complete structs can be written to, and read from, files. The i/o transfer involves simply copying bytes (there is no translation between internal binary forms of data and textual strings). The operating system is responsible for fitting the records into the blocks on disk and keeping track of the end of the file.

"Record structured" file:

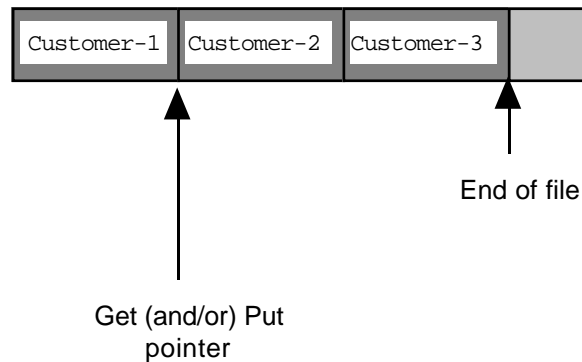


Figure 17.1 A file of "customer records"

Sequential access Files of record can be processed sequentially. The file can be opened and records read one by one until the end of file marker is reached. This is appropriate when you want to all records processed. For example, at the end of the month you might want to run through the complete file identifying customers who owed money so that letters requesting payment could be generated and dispatched.

Random access Files of records may also be processed using "random access". Random access does not mean that any data found at random will suffice (this incorrect explanation was proposed by a student in an examination). The "randomness" lies in the sequence in which records get taken from file and used. For example, if you had a file with 100 customer records, you might access them in the "random" order 25, 18, 49, 64, 3,

File systems allow "random access" because you can move the "get" (read) or "put" (write) pointer that the operating system associates with a file before you do the next read or write operation. You can read any chosen record from the file by moving the get pointer to the start of that record. This is easy provided you know the record number (imagine that the file is like an array of records, you need the index number into this array). You simply have to multiply the record number by the size of the record, this gives the byte position where the "get pointer" has to be located.

You have to be able to identify the record that you want. You could do something like assign a unique identifying number (in the range 0...?) to each customer and require that this is specified in all correspondence, or you could make use of an auxiliary table (array) of names and numbers.

You would want to use "random access" if you were doing something like taking new orders as customers came to, or phoned the office. When a customer calls, you want to be able to access their record immediately, you don't want to read all the preceding records in the file. (Of course reading the entire file wouldn't matter if you have only 100 records; but possibly you have ambitions and wish to grow to millions.)

Working with record files and random access requires the use of some additional facilities from the `istream` and `fstream` libraries.

New `istream` and `fstream` features
`fstream`

First, the file that holds the data records will be an "input-output" file. If you need to do something like make up an order for a customer, you need to read (input) the customer's record, change it, then save the changed record by writing it back to file (output). Previously we have used `ifstream` objects (for inputs from file) and `ofstream` objects (for outputs to file), now we need an `fstream` object (for bidirectional i/o). We will need an `fstream` variable:

```
fstream    gDataFile;
```

which will have to be connected to a file by an `open()` request:

```
gDataFile.open(gFileName, ios::in | ios::out );
```

The `open` request would specify `ios::in` (input) and `ios::out` (output); in some environments, the call to `open()` might also have to specify `ios::binary`. As this file is used for output, the operating system should create the file if it does not already exist.

Next, we need to use the functions that move the "get" and "put" pointers. (Conceptually, these are separate pointers that can reference different positions in the file; in most implementations, they are locked together and will always refer to the same position.) The get pointer associated with an input stream can be moved using the `seekg()` function; similarly, the put pointer can be moved using the `seekp()` function. These functions take as arguments a byte offset and a reference point. The reference point is defined using an enumerated type defined in the `istream` library; it can be `ios::beg` (start of file), `ios::cur` (current position), or `ios::end` (end of the file). So, for example, the call:

*Positioning the
get/put pointers*

```
gDataFile.seekg(600, ios::beg);
```

would position the get pointer 600 bytes after the beginning of the file; while the call:

```
gDataFile.seekp(0, ios::end);
```

would position the put pointer at the end of the file.

The libraries also have functions that can be used to ask the positions of these pointers; these are the `tellp()` and `tellg()` functions. You can find the size of a file, and hence the number of records in a record file, using the following code:

*Finding your current
position in a file*

```
gDataFile.seekg(0, ios::end);
long pos = gDataFile.tellg();
gNumRecs = pos / sizeof(Customer);
```

**Read and write
transfers**

The call to `seekg()` moves the get pointer to the end of the file; the call to `tellg()` returns the byte position where the pointer is located, and so gives the length of the file (i.e. the number of bytes in the file). The number of records in the file can be obtained by dividing the file length by the record size.

Data bytes can be copied between memory and file using the read and write functions:

```
read(void *data, int size);
write(const void *data, size_t size);
```

(The prototypes for these functions may be slightly different in other versions of the iostream library. The type `size_t` is simply a typedef equivalent for unsigned int.) These functions need to be told the location in memory where the data are to be placed (or copied from) and the number of bytes that must be transferred.

**Memory address
"Pointers"**

The first argument is a `void*`. In the case of `write()`, the data bytes are copied from memory to the disk so the memory data are unchanged, so the argument is a `const void*`. These `void*` parameters are the first example that we've seen of *pointers*. (Actually, the string library uses pointers as arguments to some functions, but we disguised them by changing the function interfaces so that they seemed to specify arrays).

A pointer variable is a variable that contains the memory address of some other data element. Pointers are defined as derived types based on built in or programmer defined struct (and class) types:

```
int      *iptr;
double   *dptr;
Rectangle *rptr;
```

and, as a slightly special case:

```
void      *ptr_to_somedata;
```

These definitions make `iptr` a variable that can hold the address of some integer data item, `dptr` a variable that holds the address of a double, and `rptr` a variable that holds the address where a `Rectangle` struct is located.

The variable, `ptr_to_somedata`, is a `void*`. This means that it can hold the address of a data item of any kind. (Here `void` is not being used to mean empty, it is more that it is "unknown" or at least "unspecified").

As any C programmer who has read this far will have noted, we have been carefully avoiding pointers. Pointers are all right when you get to know them, but they can be cruel to beginners. From now on, almost all your compile time and run-time errors are going to relate to the use of pointers.

But in these calls to the `read()` and `write()` functions, there are no real problems. All that these functions require is the memory address of the data that are involved in

the transfer. In this example, that is going to mean the address of some variable of a struct type `Customer`.

In C and C++, you can get the address of any variable by using the `&` "address-of" operator. Try running the following program (addresses are by convention displayed in hexadecimal rather than as decimal numbers):

The & "address-of" operator

```
float pi = 3.142;

int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int main()
{
    int data = 101;
    void *ptr;

    ptr = &data;
    cout << "data is at " << hex << ptr << endl;

    ptr = &pi;
    cout << "pi is at " << hex << ptr << endl;

    ptr = &(array[2]);
    cout << "array[2] is at " << hex << ptr << endl;

    ptr = &(array[3]);
    cout << "array[3] is at " << hex << ptr << endl;
    return 0;
}
```

You should be able to relate the addresses that you get to the models that you now have for how programs and data are arranged in memory. (If you find it hard to interpret the hexadecimal numbers, switch back to decimal outputs.)

The `&` operator is used to get the addresses that must be passed to the `read()` and `write()` functions. If we have a variable:

```
Customer c;
```

we can get it loaded with data from a disk file using the call:

```
gDataFile.read(&c, sizeof(Customer));
```

or we can save its contents to disk by the call:

```
gDataFile.write(&c, sizeof(Customer));
```

(Some compilers and versions of the `iostream` library may require `"(char*)"` before the `&` in these calls. The `&` operator and related matters are discussed in Chapter 20.)

Specification

Write a program that will help a salesperson keep track of customer records.

The program is to:

- 1 Maintain a file with records of customers. These records are to include the customer name, address, postcode, phone number, amount owed, and details of any items on order.
- 2 Support the following options:
 - list all customers
 - list all customers who currently owe money
 - show the record of a named customer
 - create a record for a new customer
 - fill in an order for a customer (or clear the record for goods delivered and paid for)
 - quit.
- 3 Have a list of all products stocked, along with their costs, and should provide a simple way for the salesperson to select a product by name when making up a customer order.

The program is not to load the entire contents of the data file into memory. Records are to be fetched from the disk file when needed. The file will contain at most a few hundred records so sophisticated structures are not required.

Design

First Iteration The overall structure of the program will be something like:

```
Open the file and do any related initializations
Interact with the salesperson, processing commands until
    the Quit command is entered
Close the file
```

These steps obviously get expanded into three main functions.

The "Close File" function will probably be trivial, maybe nothing more than actually closing the file. Apart from actually opening the file (terminating execution if it won't open) the "Open File" function will probably have to do additional work, like determining how many records exist. During design of the main processing loop, we may identify data that should be obtained as the file is opened. So, function "open file" should be left for later consideration.

The main processing loop in the routine that works with the salesperson will be along the following lines:

```

"Run the shop"
  quitting = false
  while (not quitting)
    prompt salesperson for a command
    use command number entered to select
    •          list all customers
    •          list debtors
    •          ...
    •          deal with quit command, i.e. quitting = true

```

Obviously, each of the options like "list all customers" becomes a separate function.

Several of these functions will have to "get" a record from the file or "put" a record to file, so we can expect that they will share "get record", "put record" and possibly some other lower level functions.

The UT functions developed in Chapter 12 (selection from menu, and keyword lookup) might be exploited. Obviously, the menu selection routine could handle the task of getting a command entered by the salesperson. In fact, if that existing routine is to be used there will be no further design work necessary for "Run the shop". The sketched code is easy to implement.

Second Iteration

So, the next major task is identifying the way the functions will handle the tasks like listing customers and getting orders. The routines for listing all customers and listing debtors are going to be very similar:

The "list ..." functions

```

"list all customers"
  if there are no customers
    just print some slogan and return

  for i = 0 , i < number of customers, . i++
    get customer record i
    print details of customer

"list debtors"
  initialize a counter to zero

  for i = 0 , i < number of customers, . i++
    get customer record i
    if customer owes money
      print details of customer
      increment counter

  if counter is zero report "no debtors"

```

You might be tempted to fold these functions into one, using an extra boolean argument to distinguish whether we are interested in all records or just those with debts. However, if the program were later made more elaborate you would probably find greater differences in the work done in these two cases (e.g. you might want the "list

debtors" operation to generate form letters suggesting that payment would be appreciated). Since the functions can be expected to become increasingly different, you might as well code them as two functions from the start.

The sketched pseudo-code uses for loops to work through the file and relies on a (global?) variable that holds the number of records. As noted in the introduction to this section, it is easy to work out the number of records in a file like this; this would get done when the file is opened.

The two functions could work with a local variable of a struct type "Customer". This would get passed as a reference parameter to a "get record" function, and as a "const reference" to a "print details" function.

*Auxiliary "get
record" and "print
details" functions*

The two additional auxiliary functions, "get record" and "print details", also need to be sketched:

```
"get record"
    convert record number to byte position
    read from file into record
    if i/o error
        print error message and exit

"print details"
    output name, address, postcode, phone number

    if amount owed
        print the amount
    else ? (either blank, or "nothing owing")

    if any items on order
        print formatted list of items
```

The specification didn't really tie down how information was to be displayed; we can chose the formats when doing the implementation.

*Functions for other
processing options*

The other required functions are those to add a new customer record to the file, show the record of a customer identified by name, and place an order for a customer (the specification is imprecise, but presumably this is again going to be a customer identified by name).

Adding a customer should be easy. The "add customer" routine would prompt the salesperson for the name, address, postcode, and phone number of the new customer. These data would be filled into a struct, then the struct would be written to the end of the existing file. The routine would have to update the global counter that records the number of records so that the listing routines would include the new record.

```
"add customer"
    ? check any maximum limits on the number of customers

    prompt for data like name, filling in a struct,

    set amount owed, and items on order to zero
```

```
write the struct to the file

update number of customers
```

The file itself would not limit the number of records (well, not until it got to contain so many millions of records that it couldn't fit on a disk). But limits could (and in this case do) arise from other implementation decisions.

Writing the struct to file would be handled by a "put record" function that matches the "get record":

```
"put record"
  convert record number to byte position
  copy data from memory struct to file
  if i/o error
    print error message and exit
```

The other two functions, "show customer" and "record order", both apparently need to load the record for a customer identified by name. The program could work reading each record from the file until it found the correct one. This would be somewhat costly even for little files of a few hundred records. It would be easier if the program kept a copy of the customer names in memory. This should be practical, the names would require much less storage than the complete records.

*Getting the record for
a named customer*

It would be possible for the "open file" routine to read all existing records, copying the customer names into an array of names. This only gets done when the program starts up so later operations are not slowed. When new records are added to the file, the customer name gets added to this array.

The memory array could hold small structures combining a name and a record number; these could be kept sorted by name so allowing binary search. This would necessitate a sort operation after the names were read from file in the "open file" routine, and an "insert" function that would first find the correct place for a new name and then move other existing names to higher array locations so as to make room for it.

Alternatively, the array could just contain the names; the array indexes would correspond to the record numbers. Since the names wouldn't be in any particular order, the array would have to be searched linearly to find a customer. Although crude, this approach is quite acceptable in this specific context. These searches will occur when the salesperson enters a "show customer" or "place order" command. The salesperson will be expecting to spend many seconds reading the displayed details, or minutes adding new orders. So a tiny delay before the data appear isn't going to matter. A program can check many hundreds of names in less than a tenth of a second; so while a linear search of the table of names might be "slow" in computer terms, it is not slow in human terms and the times of human interaction are going to dominate the workings of this program.

Further, we've got that `UT_PickKeyword()` function. It searches an array of "words", either finding the match or listing the different possible matches. This would

actually be quite useful here. If names can be equated with the `UT_Words` we can use the pick key word function to identify a name from its first few characters. Such an "intelligent" interactive response would help the salesperson who would only have to enter the first few characters of a name before either getting the required record or a list of the name and similar names.

So, decisions: Customer names will be `UT_Words` (this limits them to less than 15 characters which could be a problem), things like addresses might as well be `UT_Texts`. There will be a global array containing all the names. Functions that require the salesperson to select a customer will use the keyword search routine (automatically getting its ability to handle abbreviations).

These decisions lead to the following design sketches for the two remaining processing options:

```
"show customer"
  use keyword picker function to prompt for data (customer name)
    and find its match in names array
    (returns index of match)

  use Get Record to get the record corresponding to index

  Print Details of record got from file
```

and

```
"record order"
  use keyword picker function to prompt for data (customer name)
    and find its match in names array
    (returns index of match)

  use get record to get the record corresponding to index

  reset customer record to nothing ordered, nothing owed

  loop
    get next order item
    update amount owed
  until either no more order items or maximum number ordered

  put updated record back in file
```

The loop getting items in function "record order" is once again a candidate for promotion to being a separate function. If it had to do all the work, function "record order" would become too complex. Its role should be one of setting up the context in which some other function can get the order items. So, its sketch should be revised to:

```
"record order"
  use keyword picker function to prompt for data (customer name)
    and find its match in names array
```

```
(returns index of match)

use get record to get the record corresponding to index

get items

put updated record back in file
```

This new "get items" function now has to be planned. The specification stated that this routine should either make up an order for a customer or clear the record for goods delivered and paid for. This might not prove ideal in practice because it means that you can't deliver partial orders, and you can't accept supplementary orders; but it will do to start with. It means that the "get items" routine should start by clearing any existing record of amount owed and items ordered. Then the routine should ask the user whether any item is to be ordered. While the reply is "yes", the routine should get details of the item, add it to the record and its cost to the amount owed, and then again ask whether another item is to be ordered. The Customer struct can have an array to hold information about ordered items. Since this array will have a fixed size, the loop asking for items would need to terminate if the array gets filled up.

The specification requires the program to have a table of goods that can be ordered and a convenient mechanism allowing the salesperson to enter the name of a chosen article. The key word picking function can again be pressed into service. There will need to be a global array with the names of the goods articles, and an associated array with their costs.

An initial sketch for "get items" is:

```
"get items"
  change amount owing data member of record to zero

  ask (YesNo() function) whether another item to be ordered

  while item to be ordered and space left
    use keyword picker function to prompt for
      data (item name) and find its match
      in goods array
      (returns index of match)

    copy name of item into record
    update amount owed by cost of item

    update count of items ordered

    ask (YesNo()) whether another item needed
```

This has sketch has identified another function, `YesNo()`, that gets a yes/no input from the user.

Open File revisited

Earlier consideration of the open file function was deferred. Now, we have a better idea of what it must do. It should open the file (or terminate the program if the file won't open). It should then determine the number of records. Finally, it should scan through all records copying the customer names into an array in memory.

The array for names has a fixed size. So there will be a limit on the number of customers. This will have to be catered for in the "add customer" function.

Third iteration through the design

What is a Customer? It is about time to make some decisions regarding the data.

A Customer had better be a struct that has data members for:

- 1 customer name (already decided to use a UT_Word, i.e. up to about 15 characters);
- 2 customer address (a UT_Text, i.e. up to 60 characters);
- 3 postcode and phone, these could also be UT_Words;
- 4 amount owing (a double);
- 5 a count of items on order;
- 6 an array with the names of the items on order, need to fix a size for this.

Other fields might need to be added later. The following structs declarations should suffice:

```
#ifndef __MYCUSTOMER__
#define __MYCUSTOMER__

#include "UT.h"

struct Date {
    int fDay, fMonth, fYear;
};

const int kMAXITEMS = 5;

struct Customer {
    UT_Word      fName;
    UT_Text      fAddress;
    UT_Word      fPostcode;
    UT_Word      fDialcode;
    UT_Word      fOrders[kMAXITEMS];
    int          fNumOrder;
    double       fAmountOwing;
    Date         fLastOrder;
};

#endif
```


An extra `Date` struct has been declared and a `Date` data member has been included in the `Customer` struct. This code doesn't use dates; one of the exercises at the end of the chapter involves implementing a code to handle dates.

As a `Customer` uses `UT_Word` and `UT_Text` this header file has to include the `UT.h` header that contains the typedef defining these character array types.

The main implementation file is going to contain a number of global arrays:

- `gStock[]`, an array of `UT_Words` with names of items stocked by shop;
- `gItemCosts[]`, an array of doubles with the costs of items;
- `gCommands[]`, an array with the phrases that describe the commands that the salesperson can select;
- `gNames[]`, an array to hold customer names.

Other global (or filescope) variables will be needed for the file name, the `fstream` object that gets attached to the file, and for a number of integer counters (number of records, number of commands, number of items in stock list).

The functions have already been considered in detail and don't require further iterations of design. Their prototypes can now be defined:

```
void GetRecord(Customer& rec, int cNum);
```

Function prototypes

```
void PutRecord(Customer& rec, int cNum);
```

```
void PrintDetails(const Customer& c);
```

```
void ShowCustomer(void);
```

```
void ListAll(void);
```

```
void ListDebtors(void);
```

```
void AddCustomer(void);
```

```
int YesNo(void);
```

```
void GetItems(Customer& c);
```

```
void RecordOrder(void);
```

```
void RunTheShop(void);
```

```
void OpenTheDataFile(void);
```

```
void CloseTheDataFile(void);
```

```
int main();
```

- Other linked files** The code written specifically for this problem will have to be linked with the UT code from Chapter 12 so as to get the key word and menu selection functions.
- Header files needed** Quite a large number of systems header files will be needed. The program uses both `iostream` and `fstream` libraries for files. The `exit()` function will get used to terminate the program if an i/o error occurs with the file accesses, so `stdlib` is also needed. Strings representing names will have to be copied using `strcpy()` from the string library. The "yes/no" function will have to check characters so may need the `ctype` header that defines standard functions like `tolower()`. Error checking might use `assert`.

Implementation

The implementation file will start with the `#includes`:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <ctype.h>

#include "UT.h"
#include "mycustomer.h"
```

(If you think about it carefully, you will see that we end up `#including` `UT.h` twice; that sort of thing happens very easily and it is why we have those `#ifdef ... #endif` brackets on all header files).

The declarations of globals come next. Several are initialized:

```
const char gFileName[] = "CustRec.XXX";
const int kMAXCUSTOMERS = 200;

fstream    gDataFile;
int        gNumRecs;
UT_Word    gNames[kMAXCUSTOMERS];

UT_Word gStock[] = {
    "Floppy disks",
    ...,
    "Marker pens",
    "Laser pointer"
};

double gItemCosts[] = {
    18.50, /* disks $18.50 per box */
    ...,
    6.50, /* pens */
};
```

```

    180.0  /* laser pointer */
};

int gNStock = sizeof(gStock) / sizeof(UT_Word);

UT_Text gCommands[] = {
    "Quit",
    "List all customers",
    "List all debtors",
    "Add Customer",
    "Show Customer",
    "Record Order"
};

int gNCommands = sizeof(gCommands) / sizeof(UT_Text);

```

In many ways it would be better to introduce a new struct that packages together an item name and its cost. Then we could have an array of these "costed item" structs which would reduce the chance of incorrect costs being associated with items. However, that would preclude the use of the existing keyword functions that require a simple array of UT_Words.

The function definitions come next:

```

void GetRecord(Customer& rec, int cNum)
{
    /* cNum is customer number (0-based) */
    /* convert into offset into file */
    long where = cNum * sizeof(Customer);
    gDataFile.seekg(where, ios::beg);
    gDataFile.read(&rec, sizeof(Customer));

    if(!gDataFile.good()) {
        cout << "Sorry, can't read the customer file"
              << endl;
        exit(1);
    }
}

void PutRecord(Customer& rec, int cNum)
{
    long where = cNum * sizeof(Customer);
    gDataFile.seekp(where, ios::beg);
    gDataFile.write(&rec, sizeof(Customer)); //maybe
    (char*)&rec

    if(!gDataFile.good()) {
        cout << "Sorry, can't write to the customer file"
              << endl;
        exit(1);
    }
}

```

```
}
```

Terminating the program after an i/o error may seem a bit severe, but really there isn't much else that we can do in those circumstances.

```
void PrintDetails(const Customer& c)
{
    cout << "----" << endl;
    cout << "Customer Name : " << c.fName << endl;
    cout << "Address      : " << c.fAddress << ", "
        << c.fPostcode << endl;
    cout << "Phone        : " << c.fDialcode << endl;
    if(c.fAmountOwing > 0.0)
        cout << "Owes          : $" << c.fAmountOwing
            << endl;
    else cout << "Owes nothing" << endl;
    if(c.fNumOrder == 1) cout << "On order: " << c.fOrders[0]
        << endl;
    else
    if(c.fNumOrder > 0) {
        cout << "On order" << endl;
        for(int j = 0; j < (c.fNumOrder-1); j++)
            cout << c.fOrders[j] << ", ";
        cout << "and ";
        cout << c.fOrders[c.fNumOrder-1] << endl;
    }
    cout << "---" << endl;
}
```

The PrintDetails() function gets a little elaborate, but it is trying to provide a nice listing of items with commas and the word "and" in the right places.

```
void ShowCustomer(void)
{
    UT_Text aPrompt = "Customer Name : ";
    int who = UT_PickKeyWord(aPrompt, gNames, gNumRecs);
    if(who < 0)
        return;
    Customer c;
    GetRecord(c, who);
    PrintDetails(c);
}
```

There is one problem in using the "pick keyword" function. If the salesperson enters something that doesn't match the start of any of the names, the error message is "there is no keyword ...". Possibly the "pick keyword" function should have been designed to take an extra parameter that would be used for the error message.

```
void ListAll(void)
```

```

{
    if(gNumRecs == 0) {
        cout << "You have no customers" << endl;
        return;
    }

    for(int i = 0; i < gNumRecs; i++) {
        Customer c;
        GetRecord(c, i);
        PrintDetails(c);
    }
}

void ListDebtors(void)
{
    int count = 0;
    for(int i = 0; i < gNumRecs; i++) {
        Customer c;
        GetRecord(c, i);
        if(c.fAmountOwing > 0.0) {
            PrintDetails(c);
            count++;
        }
    }
    if(count == 0) cout << "Nothing owed" << endl;
}

```

Function `AddCustomer()` checks whether the program's array of names is full and prevents extra names being added. The input statements use `getline()`. This is because addresses are going to be things like "234 High Street" which contain spaces. If we tried to read an address with something like `cin >> c.fAddress`, the address field would get "234", leaving "High ..." etc to confuse the input to post code. The routine isn't robust; you can cause lots of troubles by entering names that are too long to fit in the specified data member.

```

void AddCustomer(void)
{
    if(gNumRecs == kMAXCUSTOMERS) {
        cout << "Sorry, you will have to edit program "
              << "before it can handle" << endl;
        cout << "    more customers." << endl;
        return;
    }

    Customer c;
    // N.B. This input routine is "unsafe"
    // there are no checks successful reads etc

    cout << "Name      : ";
    cin.getline(c.fName, UT_WRDLENGTH-1, '\n');
}

```

```

        cout << "Address      : ";
        cin.getline(c.fAddress, UT_TXTLENGTH-1, '\n');

        cout << "Post code   : ";
        cin.getline(c.fPostcode, UT_WRDLENGTH-1, '\n');

        cout << "Phone        : ";
        cin.getline(c.fDialcode, UT_WRDLENGTH-1, '\n');

        c.fNumOrder = 0;
        c.fAmountOwing = 0.0;

        PutRecord(c, gNumRecs);
        strcpy(gNames[gNumRecs], c.fName);

        gNumRecs++;
    }

    int YesNo(void)
    {
        char ch;
        cout << "Order an item? (Y or N)";
        cin >> ch;
        ch = tolower(ch);
        return (ch == 'y');
    }

    void GetItems(Customer& c)
    {
        c.fAmountOwing = 0.0;

        int count = 0;
        while(YesNo() && (count < kMAXITEMS)) {
            UT_Text aPrompt = "Identify Type of Goods";
            int which =
                UT_PickKeyWord(aPrompt, gStock, gNStock);

            strcpy(c.fOrders[count], gStock[which]);
            c.fAmountOwing += gItemCosts[which];
            count++;
        }
        c.fNumOrder = count;
    }

```

Look a bug in `GetItems()`! Can you spot it? It isn't serious, the program won't crash. But it makes the user interaction clumsy.

If you can't spot the bug, run the code and try to order more than the limit of five items. You should then observe a certain clumsiness.

The bug can be fixed by a trivial change to the code.

```
void RecordOrder(void)
{
    UT_Text aPrompt = "Enter Customer Name";
    int who = UT_PickKeyWord(aPrompt, gNames, gNumRecs);

    Customer c;
    GetRecord(c, who);

    GetItems(c);

    PutRecord(c, who);
}

void RunTheShop(void)
{
    UT_Text aPrompt = "Command";
    int quitting = 0;
    while(!quitting) {
        int command =
            UT_MenuSelect(aPrompt, gCommands,
                          gNCommands, 0);

        // Need to consume any trailing spaces or newlines
        // after the command number
        char ch;
        cin.get(ch);
        while(ch != '\n')
            cin.get(ch);

        switch(command) {
case 0:            /* Quit */
                    quitting = 1;
                    break;
case 1:            /* List all customers */
                    ListAll();
                    break;
case 2:            /* List all debtors */
                    ListDebtors();
                    break;
case 3:            /* Add Customer */
                    AddCustomer();
                    break;
case 4:            /* Show Customer */
                    ShowCustomer();
                    break;
case 5:            /* Record Order */
                    RecordOrder();
                    break;
        }
    }
}
```

```
}
```

The `RunTheShop()` function has one complication. The salesperson has to enter a number when picking the required menu option. The digits get read but any trailing spaces and newlines will still be waiting in the input stream. These could confuse things if the next function called needed to read a string, a character, or a complete line. So the input stream is cleaned up by reading characters until get the `\n` at the end of the input line.

```
void OpenTheDataFile(void)
{
    // Open the file, allow creation if not already there
    gDataFile.open(gFileName, ios::in | ios::out );
    // may need also ios::binary
    if(!gDataFile.good()) {
        cout << "? Couldn't open the file. Sorry." << endl;
        exit(1);
    }

    gDataFile.seekg(0, ios::end);
    long pos = gDataFile.tellg();
    gNumRecs = pos / sizeof(Customer);

    assert(gNumRecs <= kMAXCUSTOMERS);
    for(int i = 0; i < gNumRecs; i++) {
        Customer c;
        GetRecord(c, i);
        strcpy(gNames[i], c.fName);
    }
}
```

Logically, there is no way that the file could hold more than the maximum number of records (the file has to be created by this program and the "add customer" function won't let it happen).

Don't believe it. Murphy's law applies. The program can go wrong if the file is too large, so it will go wrong. (Something will happen like a user concatenating two data files to make one large one.) Since the program will overwrite arrays if the file is too large, it better not even run. Hence the `assert()` checking the number of records.

```
void CloseTheDataFile(void)
{
    gDataFile.close();
}

int main()
{
    OpenTheDataFile();
    RunTheShop();
}
```



```
    CloseTheDataFile();  
    return 0;  
}
```

On the whole, the program runs OK:

```
Command  
Enter option number in range 1 to 6, or ? for help  
6  
Enter Customer Name  
Ga  
Possible matching keywords are:  
Gates, B.  
Garribaldi, J  
Gamble,P  
Gam  
i.e. Gamble,P  
Order an item? (Y or N)y  
Identify Type of Goods  
Las  
i.e. Laser pointer  
Order an item? (Y or N)Y  
Identify Type of Goods  
Tone  
i.e. Toner  
Order an item? (Y or N)n  
Command  
Enter option number in range 1 to 6, or ? for helpEnter option  
number in range 1 to 6, or ? for help  
3  
----  
Customer Name : Gates, B.  
Address       : The Palace, Seattle, 923138  
Phone        : 765 456 222  
Owes         : $186.5  
On order  
Laser pointer, and Marker pens  
---  
----  
Customer Name : Gamble,P  
Address       : 134 High St, 89143  
Phone        : 1433  
Owes         : $220  
On order  
Laser pointer, and Toner  
---
```

EXERCISES

- 1 Fix the "bug" in `GetItems()` from 17.3.
- 2 Change the way that the Customer records program handles the names to the alternative approach described in the text (sorted array of name-index number structures).
- 3 Implement code to support the Date data structure and arrange that Customer orders include a Date.

18 Bits and pieces

Although one generally prefers to think of data elements as "long integers" or "doubles" or "characters", in the machine they are all just bit patterns packed into successive bytes of memory. Once in a while, that is exactly how you want to think about data.

Usually, you only get to play with bit patterns when you are writing "low-level" code that directly controls input/output devices or performs other operating system services. But there are a few situations where bits get used as parts of high level structures. Two are illustrated in sections 2 and 3 of this chapter. The first section summarizes the facilities that C++ provides for bit manipulations.

Section 18.4 covers another of the more obscure "read only" features of C++. This is the ability to cut up a (long integer) data word into a number of pieces each comprising several bits.

18.1 BIT MANIPULATIONS

If you need to work with bits, you need a data type to store them in. Generally, unsigned longs are the most convenient of the built in data types. An unsigned long will typically hold 32 bits. (There should be a header file specifying the number of bits in an unsigned long, but this isn't quite standardized among all environments.) If you need bit records with more than 32 bits you will have to use an array of unsigned longs and arrange that your code picks the correct array element to test for a required bit. If you need less than 32 bits, you might chose to use unsigned shorts or unsigned chars.

*Use unsigned longs
to store bit patterns*

Unsigned types should be used. Otherwise certain operations may result in '1' bits being added to the left end of your data.

You can't actually define bit patterns as literal strings of 0s and 1s, nor can you have them input or output in this form. You wouldn't want to; after, all they are going to be things like

*Input and output and
constants*

```
01001110010110010111011001111001
00101001101110010101110100000111
```

Instead, hexadecimal notation is used (C and C++ offer octal as an alternative, no one still uses octal so there is no point your learning that scheme). The hexadecimal (hex) scheme uses the hex digits '0', '1', ... '9', 'a', 'b', 'c', 'd', 'e', and 'f' to represent groups of four bits:

<i>Hexadecimal symbols for bit patterns</i>	hex	bits	hex	bits
	0	0000	1	0001
	2	0010	3	0011
	4	0100	5	0101
	6	0110	7	0111
	8	1000	9	1001
	a	1010	b	1011
	c	1100	d	1101
	e	1110	f	1111

(The characters 'A' ... 'F' can be used instead of 'a' ... 'f'.) Two hex digits encode the bits in an unsigned byte, eight hex digits make up an unsigned long. If you define a constant for say an unsigned long and only give five hex digits, these are interpreted as being the five right most digits with three hex 0s added at the left.

Hexadecimal constants are defined with a leading 0x (or 0X) so that the compiler gets warned that the following digits are to be interpreted as hex rather than decimal:

```
typedef unsigned long Bits;

const Bits kBITS1 = 0x8e58401f;
```

The iostream functions handle hex happily:

```
#include <iostream.h>

typedef unsigned long Bits;

void main()
{
    Bits b1 = 0x1ef2;
    cout.setf(ios::showbase);
    cout << hex << b1 << endl;

    Bits b2;
    cin >> b2;
    cout << b2 << endl;
}
```

When entering a hex number you must start with 0x; so an input for that program could be 0xa2. You should set the "ios::showbase" flag on the output stream otherwise the

numbers don't get printed with the leading 0x; it is confusing to see output like 20 when it really is 0x20 (i.e. the value thirty two).

You have all the standard logical operations that combine true/false values. They are illustrated here for groups of four bits (both binary and equivalent hex forms are shown):

What can you do with bit patterns?

0110	- (complement, or "not")	->	1001	complement
0x6			0x9	

This would be coded using the ~ ("bitwise Not operator"):

```
result = ~value;
```

0110	and	1010	->	0010	and
0x6		0xa		0x2	

This would be coded using & ("bitwise And operator"):

```
result = val1 & val2;
```

(Ouch. We have just met & as the "get the address of" operator. Now it has decided to be the "bitwise And operator". It actually has both jobs. You simply have to be careful when reading code to see what its meaning is. Both these meanings are quite different from the logical and operator, &&, that was introduced in Part II.)

0110	or	1010	->	1110	or
0x6		0xa		0xe	

This would be coded using | ("bitwise Or operator"):

```
result = val1 | val2;
```

We have had many examples with the | operator being used to make up bit patterns used as arguments, e.g. the calls to open() that had specifiers like ios::in | ios::out. The enumerators ios::in and ios::out are both values that have one bit encoded; if you want both bits, for an input-output file, you or them to get a result with both bits set.

0110	xor	1010	->	1100	exclusive or
0x6		0xa		0xc	

This would be coded using ^ ("bitwise Xor operator"):

```
result = val1 ^ val2;
```

(The "exclusive or" of two bit patterns has a one bit where either one or the other but not both of its two inputs had a 1 bit.)

You should test out these basic bit manipulations with the following program:

```
#include <iostream.h>

typedef unsigned long Bits;

int main()
{
    Bits val1, val2, val, result;
    cout.setf(ios::showbase);
    cout.setf(ios::hex,ios::basefield);

    cout << "Enter bit pattern to be complemented : ";
    cin >> val;
    result = ~val;
    cout << "Val : " << val << ", Not val : " << result <<
endl;

    cout << "Enter two bit patterns to be Anded : " << endl;
    cout << "Val1 :"; cin >> val1;
    cout << "Val2 :"; cin >> val2;
    result = val1 & val2;
    cout << "Val1 : " << val1 << ", Val2 : " << val2
        << ", And gives " << result << endl;

    cout << "Enter two bit patterns to be Ored : " << endl;
    cout << "Val1 :"; cin >> val1;
    cout << "Val2 :"; cin >> val2;
    result = val1 | val2;
    cout << "Val1 : " << val1 << ", Val2 : " << val2
        << ", Or gives " << result << endl;

    cout << "Enter two bit patterns to be Xored : " << endl;
    cout << "Val1 :"; cin >> val1;
    cout << "Val2 :"; cin >> val2;
    result = val1 ^ val2;
    cout << "Val1 : " << val1 << ", Val2 : " << val2
        << ", Xor gives " << result << endl;

    return 0;
}
```

Try to work out in advance the hex pattern that you expect as a result for the inputs that you choose. Remember that an input value gets filled out on the left with 0 hex digits so an input of 0x6 will become 0x00000006 and hence when complemented will give 0xffffffff9.

Abbreviated forms

Of course there are abbreviated forms. If you are updating a bit pattern by combining it with a few more bits, you can use the following:

```

result &= morebits;      // i.e result = result & morebits;
result |= morebits;
result ^= morebits;

```

In addition to the bitwise Ands, Ors etc, there are also operators for moving the bits around, or at least for moving all the bits to the left or to the right. *Moving the bits around*

These bit movements are done with the shift operators. They move the bits in a data element left or right by the specified number of places. A left shift operator moves the bits leftwards, adding 0s at the right hand side. A right shift operator moves bits rightwards. This is the one you need to be careful with regarding the use of unsigned values. The right shift operator adds 0s at the left if a data type is unsigned; but if it is given a signed value (e.g. just an ordinary long) it duplicates the leftmost bit when shifting the other bits to the right. This "sign extension" is not usually the behaviour that you want when working with bit patterns. Bits "falling out" at the left or right end of a bit pattern are lost.

The shift operators are:

Shift operators

```

<< left shift
>> right shift

```

Again ouch. You know these as the "takes from" and "gives to" operators that work with iostreams.

In C++, many things get given more than one job (or, to put it another way, they get "overloaded"). We've just seen that & has sometimes got to go and find an address and sometimes it has to combine bit patterns. The job at hand is determined by the context. It is just the same for the >> and << operators. If a >> operator is located between an input stream variable and some other variable it means "gives to", but if a >> operator is between two integer values it means arrange for a right shift operation.

Overloaded operators

The following code fragment illustrates the working of the shift operators:

```

#include <iostream.h>

typedef unsigned long Bits;

int main()
{
    Bits val, result;
    int i;
    cout.setf(ios::showbase);
    cout.setf(ios::hex,ios::basefield);
    cout << "Enter val "; cin >> val;
    cout << "Enter shift amount "; cin >> i;

    result = val << i;
    cout << "Left shifting gives " << result << endl;
}

```

```

        result = val >> i;
        cout << "Right shifting gives " << result << endl;

        return 0;
    }

```

An example output is:

```

Enter val 0x01234567
Enter shift amount 8
Left shifting gives 0x23456700
Right shifting gives 0x12345

```

Moving by 8 places as requested in the example means that two hex digits (4-bits each) are lost (replaced by zeros).

While you probably find no difficulty in understanding the actual shift operations, you may be doubting whether they have any useful applications. They do; some uses are illustrated in the examples presented in the next two sections.

18.2 MAKING A HASH OF IT

It is common to need to generate a "key value" that summarizes or characterises a complex data object. If data objects are in some sense "equal", then their generated key values must be equal. A key generation process need not be perfect; it is acceptable for two unequal data objects to have the same key value, but ideally the chance of this happening should be very low. A key value can be a (32-bit) unsigned long integer, though preferably it is something larger (e.g. a 64 bit "long long" if your compiler supports such things). The examples here will use unsigned longs to represent such keys.

Where might such keys be needed? There are at least two common applications:

- 1 As a filter to improve performance when searching a collection for matching data
- 2 As a summary signature of some data that can be used to check that these data are unchanged since the summary was generated.

The idea of the search filter is that you use the key to eliminate most of the data in a collection, selecting just those data elements that have equal keys. These data elements can then be checked individually, using more elaborate comparisons to find an exact match.

Hashing

The process of generating a key is known as "hashing", and it is something of an art form (i.e. there aren't any universal scientific principles that can lead you to a good hashing function for a specific type of data). Here we consider only the case of text strings because these are the most common data to which hashing is applied

18.2.1 Example hashing function for a character string

If you have a string that you want to summarize in a key, then that key should depend on every character in the string. You can achieve this by any algorithm that mixes up (or "smashes together" or "hashes") the bits from the individual characters of the string.

The "XOR" function is a good way to combine bits because it depends equally on each of its two inputs. If we want a bit pattern that combines bits from all the characters in a string we need a loop that xors the next character into a key, then moves this key left a little to fill up a long integer.

Of course, when you move the key left, some bits fall out the left end. These are the bits that encode the first few characters. If you had a long string, the result could end up depending only on the last few characters in the string.

That problem can be avoided by saving the bits that "fall out the left end" and feeding them back in on the right, xoring them with the new character data.

The following function implements this string hashing:

```
Bits HashString(const char str[])
{
    Bits    Result = 0;
    int     n = strlen(str);
    Bits    Top5Bits = 0xf8000000;
    Bits    Carry = 0x0;
    const int kleftmove = 5;
    const int krightmove = 27;
    for(int i = 0; i < n; i++) {
        Carry = Result & Top5Bits;
        Carry = Carry >> krightmove;
        Result = Result << kleftmove;
        Result ^= Carry;
        Result ^= str[i];
    }
    return Result;
}
```

The statements:

```
Carry = Result & Top5Bits;
Carry = Carry >> krightmove;
```

get the bits occupying the left most five bits of the current key and move them back to the right. The variable `Top5Bits` would commonly be referred to as a "mask". It has bits set to match just those bits that are required from some other bit pattern; to get the bits you want (from `Result`) you "perform an and operation with the mask".

The bits in the key are moved left five places by the statement:

```
Result = Result << kleftmove;
```

Then the saved leftmost bits are fed back in, followed by the next character from the string:

```
Result ^= Carry;
Result ^= str[i];
```

The following outputs show the encodings obtained for some example strings:

mellow :	0xdc67bd97
yellow :	0xf467bd97
meadow :	0xdc611d97
callow :	0xc027bd97
shallow :	0x1627bd8b
shade :	0x70588e5
2,4,6-trinitrotoluene :	0xabe69e14
2,4,5-trinitrotoluene :	0xabe59e14
Bailey, Beazley, and Bradley :	0x64c55ad0
Bailey, Beazley, and Bradney :	0x64c552d0

This small sample of test strings doesn't have any cases where two different strings get the same key, but if tried a (much) larger sample of strings you would find some cases where this occurred.

Checking a hash key is quicker than comparing strings character by character. A single comparison of the key values 0x64c55ad0 and 0x64c442d0 reveals that two strings (the two B, B, & Bs) are dissimilar. If you were to use `strcmp()`, the loop would have to compare more than twenty pairs of characters before the dissimilarity was noted.

If you have a table of complex strings that has to be searched many times, then you could gain by generating the hash codes for each string and using these values during the search process. The modified table would have structs that contained the hash key and the string. The table might be sorted on hash key value and searched by binary search, or might simply be searched linearly. A hash key would also be generated for any string that was to be found in the table. The search would check test the hash keys for equality and only perform the more expensive string match in the (few) cases where the hash keys matched.

You can make the lookup mechanism even faster. A hash key is an integer, so it could represent the index of where an item (string) should be stored in an array. The array would be initialized by generating hash keys for each of the standard strings and then copying those strings into the appropriate places in the table. (If two strings have the same hash key, the second one to be encoded gets put into the next empty slot in the array.) When a string has to be found in the table, you generate the hash key and look at that point in the array. If the string there didn't match, you would look at the strings

in the next few locations just in case two strings had ended up with the same key so causing one to be placed at a location after the place where it really should go.

Of course there is a slight catch. A hash key is an integer, but it is in the range 0...4000million. You can't really have an array with four thousand million strings.

18.2.2 A simple "hash table"

The idea of using the hash key as a lookup index is nice, even though impractical in its most basic form. A slightly modified version works reasonably well.

The computed hash key is reduced so that instead of ranging from zero to four thousand million, its range is something more reasonable – zero to a few hundred or few thousand. Tables (arrays) with a few thousand entries are quite feasible. The hash key can be reduced to a chosen range 0...N-1 by taking its value modulo N:

Modulo arithmetic

```
key = HashString(data);
```

```
key = key % N;
```

Of course that "folds together" many different values. For example, if you took numbers modulo 100, then the values 115, 315, 7915, 28415 etc all map onto the same value (15). When you reduce hash keys modulo some value, you do end up with many more "hash collisions" where different data elements are associated with the same final key. For example, if you take those key values shown previously and reduce them modulo 40, then the words meadow and yellow "collide" because both have keys that are converted to the value 7.

Hash collisions

The scheme outlined in the previous subsection works with these reduced size keys. We can have a table of strings, initially all null strings (just the '\0' character in each). Strings can be inserted into this table, or the table can be "searched" to determine whether it already contains a string. (The example in the next section illustrates an application using essentially this structure).

Figure 18.1 illustrates the structure of this slightly simplified hash table. There is an array of one thousand words (up to 19 characters each). Unused entries have '\0'; used entries contain the strings. The figure shows the original hash keys as generated by the function given earlier; these are reduced modulo 1000 to get the index.

The following data structures and functions are needed:

```
const int kLSIZE      = 20;
const int kTBLSIZE    = 1000;

typedef char LongWord[kLSIZE];

LongWord theTable[kTBLSIZE];
```

Simplified Hash Table

Index	Word	Original hash key
	\0	
	\0	
99	Function\0	(2584456099)
297	Program\0	(3804382297)
	\0	
604	Application\0	(2166479604)
618	Testing\0	(3440093618)
	\0	
	\0	
	\0	

Figure 18.1 A simplified form of hash table.

```

void InitializeHashTable(void);
    Initializes all table entries to '\0'
int NullEntry(int ndx);
    Checks whether the entry at [ndx] is a null string
int MatchEntry(int ndx, const char str[]);
    Checks whether the entry at [ndx] equals str (strcmp())
int SearchForString(const char str[]);
    Searches table to find where string str is located,
    returns position or -1 if string not present
void InsertAt(int ndx, const char str[]);
    Copies string str in table entry [ndx]
int InsertString(const char str[]);
    Organizes insertion, finding place, calling InsertAt();

```

returns position where data inserted, or -1 if table full.

The code for these functions is simple. The "initialize table" function sets the leading byte in each word to '\0' marking the entry as unused.

```
void InitializeHashTable(void)
{
    for(int i=0; i< kTBLSIZE; i++)
        theTable[i][0] = '\0';
}
```

Initializing the table

The insertion process has to find an empty slot for a new string, function "null entry" tests whether a specified entry is empty. Function "match entry" checks whether the contents of a non-empty slot matches a sought string.

```
int NullEntry(int ndx)
{
    return (theTable[ndx][0] == '\0');
}

int MatchEntry(int ndx, const char str[])
{
    return (0 == strcmp(str, theTable[ndx]));
}
```

Checking entries in the table

Insertion of entries is handled by the two functions "insert at" and "insert string". Function "insert at" simply copies a string into a (previously empty) table location:

```
void InsertAt(int ndx, const char str[])
{
    assert (strlen(str) < kLSIZE);
    strcpy(theTable[ndx],str);
}
```

Inserting an entry at its proper position

The insert string function does most of the work. First, it "hashes" the string using the function shown earlier to get a hash key. This key is then reduced modulo the length of the array to get a possible index value that defines where the data might go.

The function then has a loop that starts by looking at the chosen location in the array. If this location is "empty" (a null string), then the word can be inserted at that point. The "insert at" function is called, and the insertion position returned.

If the location is not empty, it might already contain the same string (the same word may be being entered more than once). This is checked, and if the word does match the table entry then the function simply returns its position.

In other cases, a "collision" must have occurred. Two words have the same key. So the program has to find an empty location nearby. This function uses the simplest scheme; it looks at the next location, then the one after that and so forth. The position

indicator is incremented each time around the loop; taking its value modulo the array length (so on an array with one thousand elements, if you started searching at 996 and kept finding full, non-matching entries, you would try entries 997, 998, 999, then 0, 1, etc.)

Eventually, this search should either find the matching string (if the word was entered previously), or find an empty slot. Of course, there is a chance that the array is actually completely full. If you ever get back to looking at the same location as where you started then you know that the array is full. In this case the function returns the value -1 to indicate an error.

*Finding the place to
insert a string*

```
int InsertString(const char str[])
{
    unsigned long k = HashString(str);
    k = k % kTBLSIZE;
    int pos = k;

    int startpos = pos;

    for(;;) {
        if(NullEntry(pos)) {
            InsertAt(pos, str);
            return pos;
        }
        if(MatchEntry(pos, str)) return pos;
        pos++;
        if(pos >= kTBLSIZE)
            pos -= kTBLSIZE;
        if(pos == startpos)
            return -1;
    }
}
```

The function that searches for a string uses a very similar strategy:

*"Looking up" a
string*

```
int SearchForString(const char str[])
{
    unsigned long k = HashString(str);
    k = k % kTBLSIZE;
    int pos = k;
    int startpos = pos;
    for(;;) {
        if(NullEntry(pos)) return -1;
        if(MatchEntry(pos, str)) return pos;
        pos++;
        if(pos >= kTBLSIZE)
            pos -= kTBLSIZE;
        if(pos == startpos)
            return -1;
    }
}
```

```
}
```

Although it works, this is not a particularly good implementation of a hash table. The main problem is that it wastes a lot of space with all those "null words". A better implementation is given later after pointers have been introduced.

Another problem with the implementation is the "linear search" strategy used to find an empty slot after a collision of hash keys. This strategy tends to result in clustering of entries and more costly searches. There are alternative strategies that you will be shown in more advanced courses on "data structures and algorithms".

You want to avoid a hash table getting too close to being full. The fuller it is, the greater the chance of hash collisions and then lengthy sequential searches through subsequent table entries. The usual advice is the table should be no more half full when you've loaded all the data. So, you should have a table of about five thousand entries if you need handle a vocabulary of a couple thousand words.

18.2.3 Example: identifying the commonly used words

Specification

Write a program that will produce a table giving in order the fifty words that occur most frequently in a text file, and details of the number of times that each occurred.

The program is to:

- 1 Prompt for a file name, and then to either open the file or terminate if the file cannot be accessed.
- 2 Read characters from the file. All letters are to be converted to lower case. Sequences of letters are to be assembled into words (maximum word length < 20 characters). A word is terminated by any non-alphabetic character.

(Non-alphabetic characters are discarded, they serve only as word terminators.)
- 3 Save all distinct words and maintain a count with each; this count is initialized to 1 on the first occurrence and is incremented for each subsequent occurrence.
- 4 When the file has been read completely, the words are to be sorted by frequency and details of the fifty most frequent words are to be printed.

Design

The program will have a number of distinct phases. First, it has to read the file identifying the words and recording counts. If we had a hash table containing little structures with words and counts, an insert of a new word could initialize a count while

First iteration

an insert of an already existing word would update the associated count. A hash table makes it easy to keep track of unique words. These word and count data might have to be reorganised when the end of the file is reached; the structs would need to be moved into an array that can be sorted easily using a standard sort. Next, the data would get sorted (modified version of `Quicksort()` from Chapter 13 maybe?). Finally, the printouts are needed. An initial sketch for `main()` would be:

```
open file
get the words from file
reorganize words
sort
print selection
```

Most of these "functions" are simple. We've dealt with file opening many times before. The hard part will be getting words from the file. While we still have to work out the code for this function, we do know that it is going to end up with a hash table some of whose entries are "null" and others are words with their counts. The sort step is going to need an array containing just the data elements that must be sorted. But that is going to be easy, we will just have to move the data around in the hash table so as to collect all the words at the start of the array. Sorting requires just a modification of `Quicksort`. The print out is trivial.

```
open file
    prompt for filename
    attempt to open file
    if error
        print warning message and exit

reorganize words
    i = 0
    for j = 0 ; j < hash-table-size; j++
        if(! null_entry(j))
            entry[i] = entry[j], i++

sort
    modified version of quicksort and partition
    uses an array of word structures,
    sorting on the "count" field of these structures
    (sorts in ascending order so highest frequency in
     last array elements)

print selection
    (need some checks, maybe less than 50 words!)
    for j = numwords-1, j>= numwords - 50, j--
        print details of entry[j]
```


The sort routine will need to be told the number of elements. It would be easy for this to be determined by the "reorganize" routine; it simply has to return the value of its 'i' counter.

These functions shouldn't require any further design iterations. They can be coded directly from these initial specifications. With "sort" and "open file" it is just a matter of "cutting and pasting" from other programs, followed by minor editing. The data type of the arguments to the `Quicksort()` function must be changed, and the code that accesses array elements in `Partition()` must be adjusted. These elements are now going to be structs and the comparison tests in `Partition()` will have to reference specific data members.

The "get the words" process requires more analysis, but the basic structure of the function is going to be something simple like:

```
"get the words"
  Initialize hash table
  while get another word
    insert word
```

(It would be sensible for the insertion to be checked; the program should stop if the hash table has become full causing the insert step to fail.)

The program needs slightly modified versions of the hash table functions given in the last section. There is no need for a search function. The other functions used to manipulate the hash table have to be changed to reflect the fact that the table entries aren't just character arrays, instead they are little structs that contain a character array and a count. The main change will be in the insert function. If a word is already in the array, its count is to be updated:

*Second iteration
through design*

```
insert word
  Get hash key
  reduce key modulo array size
  initialize search position from reduced key
  loop
    if entry at position is null
      insert at ...
      return position
    if entry at position has string equal to argument
      update count in entry at position
      return position
    increment position (mod length of array)
  check for table full (return failure -1 indicator)
```

The "get word" function should fill in a character array with letters read from the file. The function should return true if it finds a word, false if there is no word (this will only occur at the end of file). When called, the function should start by discarding any leading whitespace, digit, or punctuation characters. It can start building up a word with the first letter that it encounters.

Successive letters should be added to the word; before being added they must be converted to lower case. This letter adding loop terminates when a non-letter is found. The function has to check for words in the file that are larger than the maximum allowed for. The program can be terminated if excessive length words are found.

The function has to put a '\0' after the last letter it adds to the word.

These considerations result in the following sketch for "get word":

```

get word (given reference word argument to fill in)
  initialize word[0] to '\0'
    (just in case there is nothing left in file)
  do
    get character
    if at end of file
      return
    while character isn't a letter

    while character is letter
      add to word
      check word length exceeded
      get next character
      if end of file
        break loop
    add '\0' to word

```

Third iteration

All that remains is to decide on data structures and function prototypes. We need a struct that combines a character array and a count:

```

const int kLSIZE          = 20;
typedef char LongWord[kLSIZE];
struct WordInfo {
    LongWord    fWord;
    long        fCount;
};

```

Twenty characters (19 + terminating '\0') should suffice for most words. The struct can have a LongWord and a long count.

The "hash table" will need a large array of these structs:

```

const int kTBLSIZE        = 4000;
WordInfo gTable[kTBLSIZE];

```

(This gTable array needs about 90,000 bytes. Symantec 8 on the PowerPC handles this; but Symantec 7 on a Mac-Quadra cannot handle static data segment arrays this large. You will also have problems in the Borland environment; you will have to use 32-bit memory addressing models for this project.)

The only other global data element would be the ifstream for the input file.

```
Bits HashString(const char str[]);
    Hashing function shown earlier

void InitializeHashTable(void);
    Modified version of function given previously.  Initializes
    both data members of all structs in "hash table" array;
    struct's fWord character array set to null string, fCount
    set to zero.

int NullEntry(int ndx);
int MatchEntry(int ndx, const char str[]);
    Modified versions of functions given previously.  Use fWord
    data member of struct.

void InsertAt(int ndx, const char str[]);
    Modified version of function given previously.  Uses
    strcpy() to set fWord data member of struct; initializes
    fCount data member to 1.

int InsertWord(const char str[]);
    Organize insertion of new word, or update of fCount data
    member for existing word.

void OpenFile();

int GetWord(LongWord& theWord);
    Fills word with next alphabetic string from file
    (terminates
    program if word too large).

void GetTheWords(void);
    Organize input or words and insertion into hash table.

int CompressTable(void);
    Closes up gaps in table prior to sorting.

int Partition( WordInfo d[], int left, int right);
    Quicksort's partitioning routine a for a WordInfo array.

void Quicksort( WordInfo d[], int left, int right);
    Modified Quicksort.

void PrintDetails(int n);
    Prints the 50 most frequent words (or all words if fewer
    than 50) with their counts.

int main();
```

Implementation

Only a few of the functions are shown here. The rest are either trivial to encode or are minor adaptations of functions shown earlier.

All the hash table functions from the previous section have modifications similar to that shown here for the case of `InitializeHashTable()`:

```
void InitializeHashTable(void)
{
    for(int i=0; i< kTBLSize; i++) {
        gTable[i].fWord[0] = '\0';
        gTable[i].fCount = 0;
    }
}
```

The "insert" function has slightly modified behaviour:

```
int InsertWord(const char str[])
{
    unsigned long k = HashString(str);
    k = k % kTBLSize;
    int pos = k;

    int startpos = pos;

    for(;;) {
        if(NullEntry(pos)) {
            InsertAt(pos, str);
            return pos;
        }
        if(MatchEntry(pos, str)) {
            gTable[pos].fCount++;
            return pos;
        }
        pos++;
        if(pos >= kTBLSize)
            pos -= kTBLSize;
        if(pos == startpos)
            return -1;
    }
}
```

The `GetWord()` function uses two loops. The first skips non alphabetic characters; the second builds the words:

```
int GetWord(LongWord& theWord)
{
    int n = 0;
    char ch;
```

```

    theWord[0] = '\0';

    do {
        gDataFile.get(ch);
        if(gDataFile.eof())
            return 0;
    } while (!isalpha(ch));

    while(isalpha(ch)) {
        theWord[n] = tolower(ch);
        n++;
        if(n==(kLSIZE-1)) {
            cout << "Word is too long" << endl;
            exit(1);
        }
        gDataFile.get(ch);
        if(gDataFile.eof())
            break;
    }
    theWord[n] = '\0';
    return 1;
}

```

The `GetTheWords()` function reduces to a simple loop as all the real work is done by the auxiliary functions `GetWord()` and `InsertWord()`:

```

void GetTheWords(void)
{
    InitializeHashTable();
    LongWord aWord;
    while(GetWord(aWord)) {
        int pos = InsertWord(aWord);
        if(pos < 0) {
            cout << "Oops, table full" << endl;
            exit(1);
        }
    }
}

```

The `CompressTable()` function shifts all entries down through the array. Index *i* identifies the next entry to be filled in. Index *j* runs up through all entries. If a non null entry is found, it is copied and *i* incremented:

```

int CompressTable(void)
{
    int i = 0;
    int j = 0;
    for(j = 0; j < kTBLSIZE; j++)
        if(!NullEntry(j)) {
            gTable[i] = gTable[j];

```

```

        i++;
    }
    return i;
}

```

The `Partition()` function has minor changes to argument list and some statements:

```

int Partition( WordInfo d[], int left, int right)
{
    int val = d[left].fCount;
    int lm = left-1;
    int rm = right+1;
    for(;;) {
        do
            rm--;
        while (d[rm].fCount > val);

        ...

        ...
    }
}

```

Function `PrintDetails()` uses a loop that runs backwards from the last used entry in the table, printing out entries:

```

void PrintDetails(int n)
{
    int min = n - 50;
    min = (min < 0) ? 0 : min;
    for(int i = n-1, j = 1; i >= min ; i --, j++)
        cout << setw(5) << j << ": "
            << gTable[i].fWord << ", \t"
            << gTable[i].fCount << endl;
}

```

As usual, `main()` simplifies to a sequence of function calls:

```

int main()
{
    OpenFile();
    GetTheWords();
    int num = CompressTable();
    Quicksort(gTable, 0, num - 1);
    PrintDetails(num);
    return 0;
}

```

Part of the output from a run against a test data file is:

```
Enter filename
txtf.txt
 1: the,      262
 2: of,       126
 3: to,       104
 4: a,        86
 5: and,      74
 6: in,       70
 7: is,       50
 8: that,    41
 9: s,        39
10: as,       32
11: from,    30
12: it,      29
13: by,      29
14: are,     27
15: for,     27
16: be,      27
```

Most of these are exactly the words that you would expect. What about things like 's'? You get oddities like that. These will be from all the occurrence of 's at the ends of words; the apostrophe ended a word, a new word starts with the s, then it ends with the following space.

Because the common words are all things like "the", they carry no content information about the document processed. Usually programs used to analyze documents have filters to eliminate these standard words. An exercise at the end of this chapter requires such an extension to the example program.

18.3 CODING "PROPERTY VECTORS"

Where might you want actual bit data, that is collections of bits that individually have meaning?

One use is in information retrieval systems.

Suppose you have a large collection of news articles taken from a general science magazine (thousands of articles, lengths varying from 400 to 2000 words). You want to have this collection arranged so that it can be searched for "articles of interest".

Typically, information retrieval systems allow searches in which articles of interest are characterized as those having some minimum number of keywords from a user specified group of keywords. For example, a request might require that at least four of the following keywords be found in an article:

```
aids, hiv, siv, monkey, virus, immune
```

This example query would find articles describing scientific studies on the HIV virus and/or the related virus that infects monkeys (SIV).

You wouldn't want the search program to read each article into memory and check for occurrences of the user specified keywords, that would be much too slow. However, if the keywords used for searches are restricted to those in a predefined set, then it is fairly easy to implement search schemes that are reasonably fast.

These search schemes rely on indexes that relate keywords to articles. The simplest approach is illustrated in Figure 18.2. The main file contains the source text of the news articles. A second "index file" contains small fixed size data records, one for each article in the main file.

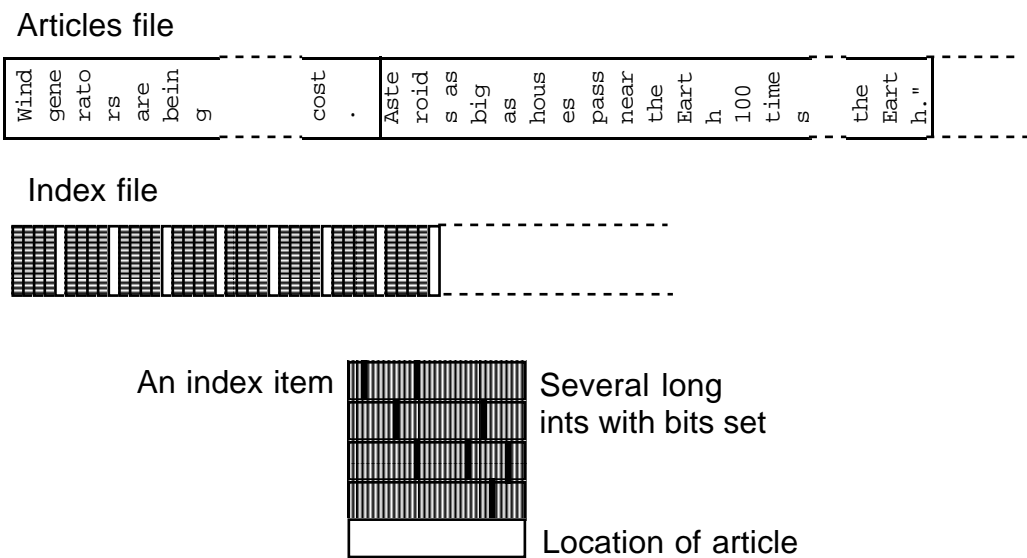


Figure 18.2 Simple Information Retrieval system.

Each index record contains a "bit map". Each bit corresponds to one of the keywords from the standard set. If the bit is '1' it means that the article contained that keyword; if the keyword is not present in the article, the corresponding bit in the map is a zero. Because the articles vary in size, it is not possible to work out where an article starts simply from knowing its relative position in the sequence of articles. So, the index record corresponding to an article contains details of where that article starts in the main file.

Two programs are needed. One adds articles to the main articles file and appends index records to the index file. The second program performs the searches.

Both programs share a table that identifies the (keyword, bit number) details. It is necessary to allow for "synonyms". For example, the articles used when developing

this example included several that discussed chimpanzees (evolution, ecology, social habits, use in studies of AIDS virus, etc); in some articles the animals were referred to as chimpanzees, in others they were "chimps". "Chimp" and chimpanzee are being used as synonyms. If both keywords "chimpanzee" and "chimp" are associated with the same bit number, it is possible to standardise and eliminate differences in style and vocabulary.

A small example of a table of (keyword, bit numbers) is:

```
typedef char LongWord[20];

struct VocabItem {
    LongWord    fWord;
    short       fItemNumber;
};

VocabItem vTable[] = {
    { "aids", 0 },
    { "hiv", 1 },
    { "immunity", 2 },
    { "immune", 2 },
    { "drug", 3 },
    { "drugs", 3 },
    { "virus", 4 },
    ...
};
```

A real information retrieval system would have thousands of "keywords" that map onto a thousand or more standard concepts each related to a bit in a bit map. The example here will be smaller; the bit maps will have 128 bits thus allowing for 128 different key concepts that can be used in the searches.

The program used to add articles to the file would start by using the data in the `VocabItem` table to fill in some variation on the hash table used in the last example. The various files needed would be opened (index file, articles file, text file with new article). Words can then be read from the source text file using code similar to that in the last example (the characters can be appended to the main articles file as they are read from the source text file). The words from the file are looked up in the hash table. If there is no match, the word is discarded. If the word is matched, then the word is one of the keywords; its bit number is taken the matched record and used to set a bit in a bit map record that gets built up. When the entire source text has been processed, the bit map and related location data get appended to the index file.

Adding data to the file

The query program starts by prompting the user to enter the keywords that are to characterize the required articles. The user is restricted to the standard set of keywords as defined in the `VocabItem` table; this is easy to do by employing code similar to the "pick keyword" function that has been illustrated in earlier examples. As keywords are picked, their bit numbers are used to set appropriate bits in a bit map. Once all the keywords have been entered, the user is prompted to specify the minimum number of

Running a query

matches. Then each index record gets read in turn from the index file. The bit map for the query and that from the index file are compared to find the number of bits that are set in both. If this number of common bits exceeds the minimum specified, then the article should be of interest. The "matched" article is read and displayed.

Bitmaps

The bitmaps needed in this application will be simply small arrays of unsigned long integers. If we have to represent 128 "concepts", we need four long integers. If you go down to the machine code level, you may find that a particular machine architecture defines a numbering for the bits in a word. But for a high level application like this, you can choose your own. The chosen coding is shown in Figure 18.3.

Bitmap: 4 unsigned longs

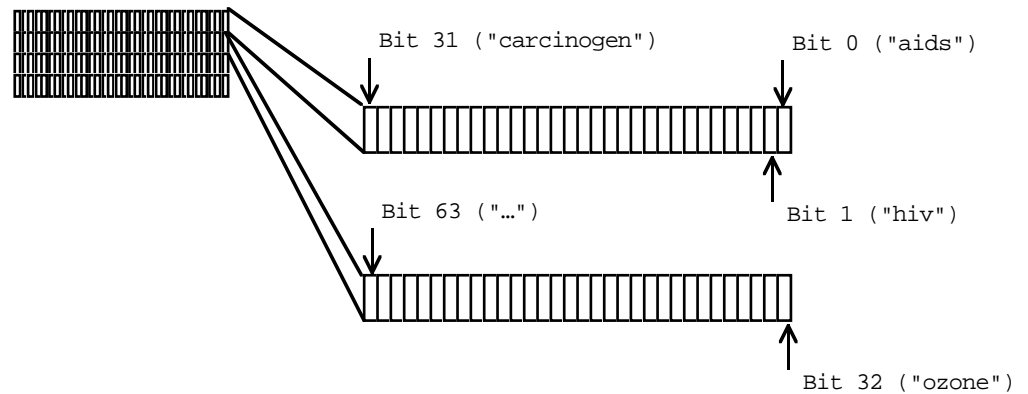


Figure 18.3 Bit maps and chosen bit numbering.

Setting a required bit

In order to set a particular bit, it is necessary to first determine which of the unsigned longs contains that bit (divide by 32) and then determine the bit position (take the bit number modulo 32). Thus bit 77 would be the 13th bit in `bitmap[2]`.

Counting the bits in common

Bits in common to two bit maps can be found by "Anding" the corresponding array entries, as shown in Figure 18.4. The number of bits in common can be calculated by adding up the number of bits set in the result. There are several ways of counting the number of bits set in a bit pattern. The simplest (though not the fastest) is to have a loop that tests each bit in turn, e.g. to find the number of bits in an unsigned long `x`:

```
int count = 0;
int j = 1;
for(int i=0;i<32;i++) {
    if(x & j)
        count++;
    j = j << 1;
}
```

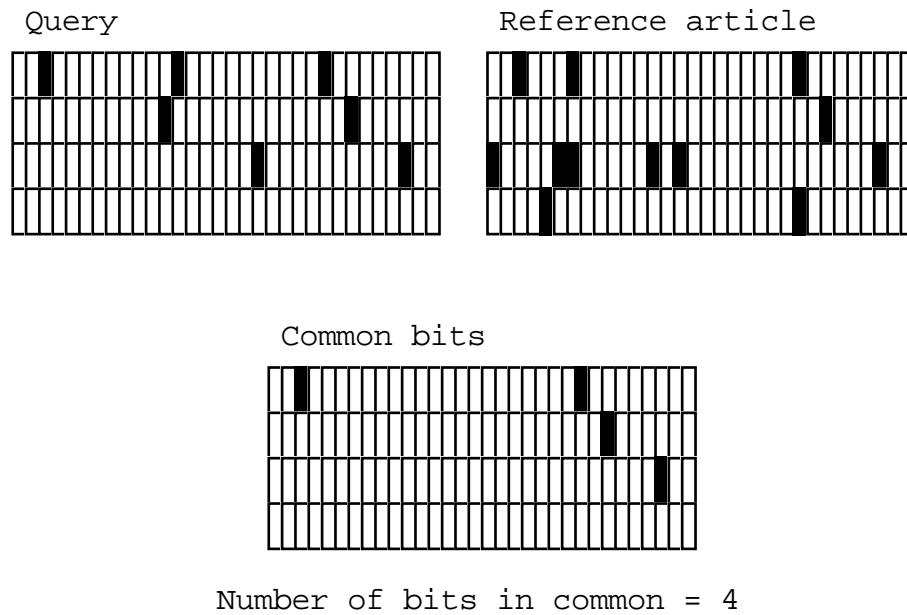


Figure 18.4 Finding and counting the common bits.

Specification

Implement an article entry program and a search program that together provide an information retrieval system that works in the manner described above.

Design

These programs are actually easier than some of the earlier examples! The first sketches for `main()`s for the two programs are:

```

article addition program
  open files
  process text
  close files

```

and

```

search program
  open files(
  get the query
  search for matches

```

Addition program

The open files routine for the addition program has to open an input file with the source text of the article that is to be processed and two output files the main articles file and the index file. New information is to be appended to these files. The files could be opened in "append mode" (but this didn't work with one of the IDEs which, at least in the version used, must have a bug in the iostream run time library). The files can be opened for output specifying that the write position be specified "at the end" of any existing contents.

```
open file
  prompt for and open file with text input
    (terminate on open error)
  open main and index files for output, position "at end"
    (terminate on open error)
```

The close files routine will be trivial, it will simply close all three files.

The main routine is the "process text" function. This has to initialize the hash table, then fill it with the standard words, before looping getting and dealing with words from the text file. When all the words have been dealt with, an assembled bit map must be written to the index file; this bit map has to be zeroed out before the first word gets dealt with.

```
process text file
  initialize and load up hash table
  zero bit map
  while Get Word from file
    deal with word
  write bit map and related info
```

Hash table The hash table will be like that in the example in the previous section. It will contain small structs. This time instead of being a word and a count, they are "vocab items" that consist of a name and a bit number.

The words are easy to "deal with". The hash table is searched for a match, if one is found its bit number is set in the bit map.

Search program

The open files routine for this program needs to open both index and articles file. No other special actions are required.

The task of "getting the query" can be broken down as follows:

```
get the query
```

```

zero bit map representing query
loop
    get a keyword
    set appropriate bit in bit map
until no more keywords needed
ask for number of keys that must match

```

The loop will be similar to those in previous examples. After each keyword is dealt with, the user will be asked for a Yes/No response to a "More Keywords?" prompt.

The "search for matches" routine will have roughly the following structure:

```

search for matches
while not end of file on index file
    get the bit map of an index record from file (and
        details of where article is located)
    get number of bits in common between index record's
        bit map and query bit map
    if number matched exceeds minimum
        show matching article

```

The "show matching article" function will be something like:

```

show match
    move "get pointer" to appropriate position in articles'
file
    read character
    while not end marker
        print character
        read next character

```

The articles in the file had better be separated by some recognizable marker character! A null character ('\0') would do. This had better be put at the end of each article by the addition program when it appends data to that file. The "show matching article" function would probably need some extra formatting output to pretty things up. It might be useful if either it, or the calling search routine, printed details of the number of keywords matched.

Bitmaps and bit map related functions

Both programs share a need for bit maps and functions to do things like zero out the bit maps when initializing things, setting chosen bits, and counting common bits. These requirements can be met by having a small separate package that deals with bitmaps. This will have a header file that contains a definition of what a bit map is and a list of function prototypes. A second implementation file will contain the function definitions.

A bit map will be a small array of unsigned longs. This can be specified using a typedef in the header file. The required functions seem to be:

```

Zero bits
    clear all bits in the bit map

Set bit
    work out which array element and which bit
    or a 1 into the appropriate bit

Count common bits
    build a temporary bit map that represents the "And"
        of two bit maps
    for each array element in temporary bit map
        count its bits and add to overall total
    return overall total

```

Further design steps

Many of the functions in both programs are already either simple enough to be coded directly, or are the same as functions used in other examples (e.g. the `YesNo()` function used when asking for more keywords in the search program). A few require further consideration.

The "Get Word" function for the addition program can be almost identical to that in the example in 18.2. The only addition is that every character read must be copied to the output articles file.

The "process text" function needs a couple of additions:

```

process text file
    initialize and load up hash table
    zero bit map
    note the position of the current "end of file"
      of articles file
    while Get Word from file
        deal with word
    write a terminating null to the articles file
    write bit map and related info (i.e. the position
      of previous end of file!)

```

These additions make certain that there is a null character separating articles as required by the search program's "show match" function. The other additions clarify the "related info" comment in the original outline. Each record written to the index file has to contain the location of the start of the corresponding article as well as a bit map. So the "process" function had better note the current length of the articles file before it adds any words. This is the "related info" that must then be written to the index file.

The "get a keyword" function needed in search can be modelled on the `PickKeyWord()` function and its support routines developed in Section 12.4. Apart from `PickKeyWord()` itself, there were the associated routines `FindExactMatch()`,

CountPartialMatches() and PrintPartialMatches(). All of these routines have to be reworked so that they use an array of VocabItems rather than a simple array of strings. This recoding is largely a matter of changing the data type of arguments and adding a data member name to some references to array elements.

Here this is simply a matter of identifying global (and filescope data), deciding on how to deal with the VocabItem array needed in both programs, and finalising the function prototypes. *Final steps in design*

Both programs share the array of VocabItems (the structs with keywords and bit numbers), and they also both need a count of the number of VocabItems defined. This information should be in a separate file that can be #included by both programs:

```
/*
 * Vocabulary file for information retrieval example.
 */

typedef char LongWord[20];

struct VocabItem {
    LongWord    fWord;
    short    fItemNumber;
};

VocabItem vTable[] = {
    { "aids", 0 },
    { "hiv", 1 },
    ...
    { "cancer", 28 },
    { "tumour", 29 },
    { "therapy", 30 },
    { "carcinogen", 31 },
    { "ozone", 32 },
    { "environment", 33 },
    { "environmental", 33 },
    ...
    { "toxin", 50 },
    { "poison", 50 },
    { "poisonous", 50 },
    ...
};

int NumItems = sizeof(vTable) / sizeof(VocabItem);
```

Filevocab.inc

The vTable array and the count NumItems will be "globals" in both the programs.

Both programs will require a number of ifstream and/or ofstream variables that get attached to files. These can be globals.

The addition program requires a "hash table" whose entries are `VocabItems`. This won't be particularly large as it only has to hold a quick lookup version of the limited information in the `vTable` array.

The search program could use globals for the bit map that represents a query and for the minimum acceptable match.

The typedef defining a "bit map" would go in a header file along with the associated function prototypes:

File mybits.h

```
#ifndef __MYBITS__
#define __MYBITS__

#define MAXBIT    127
#define MAPSZ     4

typedef unsigned long Bits;

typedef Bits Bitmap[MAPSZ];

void ZeroBits(Bitmap& b);

void SetBit(int theBit, Bitmap& b);

int CountCommonBits( Bitmap& b1,  Bitmap& b2);

#endif
```

The function prototypes for the remaining functions in the two programs are:

article addition program

```
Bits HashString(const char str[]);
    The hash function given earlier.

void InitializeHashTable(void);
    Fills hash table with "null VocabItems" (fWord field
    == '/0', fItemNumber field == -1).

int NullEntry(int ndx);
int MatchEntry(int ndx, const LongWord w);
void InsertAt(int ndx, const VocabItem v);
    Similar to previous examples except for use of VocabItem
    structs.

int SearchForWord(const LongWord w);
    Minor variation on previously illustrated hash table
    search function.

int InsertVocabItem(const VocabItem v);
```



```
    Inserts standard VocabItem into hash table.

void InsertKeyWords(void);
    Loops through all entries in vTable, inserting copies into
    hash table.

void OpenFiles(void);
void CloseFiles(void);

int GetWord(LongWord& theWord);
    Similar to previous get word, just copies input characters
    to output file in addition to other processing.

void ProcessText(void);
    Main loop of addition program.

int main();
```

search program

```
void OpenFiles(void);

int FindExactMatch(const VocabItem keyws[], int nkeys,
    const LongWord input);
int CountPartialMatches(const VocabItem keyws[], int nkeys,
    const LongWord input, int& lastmatch);
void PrintPartialMatches(const VocabItem keyws[], int nkeys,
    const LongWord input);
int PickKeyWord(const VocabItem keywords[], int nkeys);
    These functions are minor variations of those defined
    in 12.4

int YesNo(void);

void GetTheQuery(void);
    Loop building up bit map that represents the query.

void ShowMatch(long where);
    Prints article starting at byte offset 'where' in main
    articles file.

void SearchForMatches(void);

int main();
```

Implementation

Only a few of the functions are shown here. The others are either identical to, or only minor variations, of functions used in earlier examples.

Examples of functions from the mybits.cp file are:

```
void SetBit(int theBit, Bitmap& b)
{
    assert((theBit >= 0) && (theBit <= MAXBIT));
    int word = theBit / 32;
    int pos = theBit % 32;
    int mask = 1 << pos;
    b[word] |= mask;
}
```

Function SetBit() uses the scheme described earlier to identify the array element and bit position. A "mask" with one bit set is then built by shifting a '1' into the correct position. This mask is then Or-ed into the array element that must be changed.

Function CountCommonBits() Ands successive elements of the two bit patterns and passes the result to function CountBits(). The algorithm used by CountBits() was illustrated earlier.

```
int CountCommonBits( Bitmap& b1, Bitmap& b2)
{
    int result = 0;
    for(int i = 0; i < MAPSZ; i++) {
        Bits temp = b1[i] & b2[i];
        result += CountBits(temp);
    }
    return result;
}
```

The standard hash table functions all have minor modifications to cater for the different form of a table entry:

```
void InitializeHashTable(void)
{
    for(int i=0; i< kTBLSize; i++) {
        gTable[i].fWord[0] = '\0';
        gTable[i].fItemNumber = -1;
    }
}
```

The OpenFiles() function for the articles addition program has a mode that is slightly different from previous examples; the ios::ate parameter is set so that new data are added "at the end" of the existing data:

```
void OpenFiles(void)
{
    char fname[100];
    cout << "Enter name of file with additional news article"
          << endl;
```

```

cin >> fname;
gTextFile.open(fname, ios::in | ios::nocreate);
if(!gTextFile.good()) {
    cout << "Sorry, couldn't open input text file"
         << endl;
    exit(1);
}
gInfoData.open(InfoFName1, ios::out | ios::ate);
if(!gInfoData.good()) {
    cout << "Sorry, couldn't open main info. file" <<
         endl;
    exit(1);
}
gInfoIndex.open(InfoFName2, ios::out | ios::ate);
...
}

```

Function `GetWord()` is similar to the previous version, apart from the extra code to copy characters to the output file:

```

int GetWord(LongWord& theWord)
{
    int n = 0;
    char ch;
    theWord[0] = '\0';

    do {
        gTextFile.get(ch);
        if(gTextFile.eof())
            return 0;
        gInfoData.write(&ch,1);
    } while (!isalpha(ch));

    while(isalpha(ch)) {
        ...
        gInfoData.write(&ch,1);
    }
    theWord[n] = '\0';
    return 1;
}

```

Most of the work is done by `ProcessText()`:

```

void ProcessText(void)
{
    InitializeHashTable();
    InsertKeyWords();

    LongWord aWord;

```

```

    Bitmap aMap;
    ZeroBits(aMap);
    long where;
    where = gInfoData.tellp();
    while(GetWord(aWord)) {
        int pos = SearchForWord(aWord);
        if(pos >= 0) {
            int keynum = gTable[pos].fItemNumber;
            SetBit(keynum, aMap);
        }
    }
    char ch = '\0';
    gInfoData.write(&ch, 1);
    gInfoIndex.write(&aMap, sizeof(aMap));
    gInfoIndex.write(&where, sizeof(long));
}

```

The call to `tellp()` gets the position of the end of the file because the open call specified a move to the end. The value returned from `tellp()` is the starting byte address for the article that is about to be added.

The principal functions from the search program are:

```

void GetTheQuery(void)
{
    ZeroBits(gQuery);
    cout << "Enter the terms that make up the query" << endl;
    do {
        int k = PickKeyWord(vTable, NumItems);
        int bitnum = vTable[k].fItemNumber;
        SetBit(bitnum, gQuery);
    }
    while (YesNo());
    cout << "How many terms must match ? ";
    cin >> gMinMatch;
}

```

Function `GetTheQuery()` builds up the query bit map in the global `gQuery` then set `gMinMatch`.

Function `ShowMatch()` basically copies characters from the articles' file to the output. There is one catch here. The file will contain sequences of characters separated by an end of line marker. The actual marker character used will be chosen by the text editor used to enter the original text. This "end of line" character may not result in a new line when it is sent to the output (instead an entire article may get "overprinted" on a single line). This is catered for in the code for `ShowMatch()`. A check is made for the character commonly used by editors to mark an end of line (character with hex representation `0x0d`). Where this occurs a newline is obtained by `cout << endl`. (You

might have to change that hex constant if in your environment the editors use a different character to mark "end of line".)

```
void ShowMatch(long where)
{
    char ch;
    int linepos = 0;
    gDatafile.seekg(where, ios::beg);
    gDatafile.get(ch);
    while(ch != '\0') {
        if(ch == 0x0d)
            cout << endl;
        else cout << ch;
        gDatafile.get(ch);
    }
    cout << "\n-----\n";
}
```

The `SearchForMatches()` function simply reads and checks each record from the index file, using `ShowMatch()` to print any matches. An error message is printed if nothing useful could be found.

```
void SearchForMatches(void)
{
    Bitmap b;
    long wh;
    int matches = 0;
    gIndexfile.seekg(0, ios::beg);
    while(!gIndexfile.eof()) {
        gIndexfile.read(&b, sizeof(Bitmap));
        gIndexfile.read(&wh, sizeof(long));
        int n = CountCommonBits(b, gQuery);
        if(n>=gMinMatch) {
            cout << "Matched on " << n << " keys"
                << endl;
            ShowMatch(wh);
            matches++;
        }
    }
    if(matches == 0)
        cout << "No matches" << endl;
}
```

A test file was built using approximately seventy articles from a popular science magazine as input. The results of a typical search are:

```
Enter the terms that make up the query
fusion
Another search term? (Y or N) : y
power
```

```

Another search term? (Y or N) : n
How many terms must match ? 2
Matched on 2 keys
Cold fusion is alive and well and thriving on Japanese money in an
"attractive" part of France, says Martin Fleischmann, the chemistry
professor who in 1989 claimed to have produced nuclear fusion in a
test tube at room temperature.
Cold fusion said Fleischmann and his American colleague Stanley
...
negligible compared with the heat liberated.

-----

```

18.4 PIECES (BIT FIELDS)

This is definitely a "read only" topic in C++, and outside of a few text books and some special purpose low-level code it is topic where you won't find much to read. Any low-level code using the features described in this section will relate to direct manipulation of particular groups of bits in the control registers of hardware devices. We will not cover such machine specific detail and consider only the (relatively rare) usage in higher level data structures.

Suppose you have some kind of data object that has many properties each one of which can take a small number of values (mostly the same sorts of thing that you would consider suitable for using enumerated types) e.g.:

```

colour: coral, pearl, smoke-grey, steel-blue, beige;
size:   small, medium, large, x-large, xx-large,
finish: matte, silk, satin, gloss
style:  number in range 0..37

```

If you did chose to work with enumerated types, the compiler would make each either a single unsigned character, or an unsigned short integer. Your records would need at least three bytes for `colour`, `size`, and `finish`; another unsigned byte would be needed for the `style`. Using enumerated types, the typical struct representing these data would be at least four bytes, 32-bits, in size.

But that isn't the minimum storage needed. There are five colors, for that you need at most three bits. Another three bits could hold the size. The four finishes would fit in two bits. Six bits would suffice for the style. In principle you could pack those data into 14 bits or two bytes.

This is permitted using "bit fields":

```

#include <stdlib.h>
#include <iostream.h>

struct meany {

```

```
    unsigned colour : 3;
    unsigned size : 3;
    unsigned finish : 2;
    unsigned style : 6;
};

int main()
{
    meany m;

    m.colour = 4;
    m.size = 1;
    m.finish = 2;
    m.style = 15;
    cout << m.style << endl;

    return EXIT_SUCCESS;
}
```

(With the compiler I used, a "meany" is 4 bytes, so I didn't get to save any space anyway.)

Note, you are trading space and speed. For a small reduction in space, you are picking up quite an overhead in the code needed to get at these data fields.

There is one possible benefit. Normally, if you were trying to pack several small data fields into a long integer, you would need masking and shift operations to pick out the appropriate bits. Such operations obscure your code.

If you use bit fields, the compiler generates those same masking and shift operations to get at the various bit fields. The same work gets done. But it doesn't show up in the source level code which is consequently slightly easier to read.

*A real advantage of
bit fields*

19 Beginners' Class

Chapter 17 illustrated a few applications of structs. The first was a reworking of an earlier example, from Section 13.2, where data on pupils and their marks had to be sorted. In the original example, the names and marks for pupils were in separate arrays. Logically, a pupil's name and mark should be kept together; the example in Section 17.1 showed how this could be done using a struct. The marks data in these structs were only examined by the sort function, and about the only other thing that happened to the structs was that they got copied in assignment statements. In this example, the structs were indeed simply things that kept together related data.

In all the other examples from Chapter 17, the structs were used by many functions. Thus the `Points` could be combined with an `AddPoint()` function, while the `Customer` records in 17.3 were updated, transferred to and from disk, and printed. Although they used arrays rather than structs, the examples in Chapter 18 also had data structures that had many associated functions. Thus, the different forms of "hash table" all had functions for initialization, searching for a key, inserting a key, along with associated support functions (like the check for a "null" hash table entry). Similarly, the "bit maps" for the information retrieval system in Section 18.3 had a number of associated functions that could be used to do things like set specific bits.

More than just a data collection

You would have difficulties if you were asked "Show me how this code represents a 'Customer' (or a 'Hash Table', or a 'Bit map')". The program's representation of these concepts includes the code manipulating the structures as well as the structures themselves.

Data and associated functions

This information is scattered throughout the program code. There is nothing to group the functions that manipulate `Customer` records. In fact, any function can manipulate `Customer` records. There may have been a `PrintDetails(Customer&)` function for displaying the contents of a `Customer` record but there was nothing to stop individual data members of a `Customer` record being printed from `main()`.

When programs are at most a couple of hundred lines, like the little examples in Chapters 17 and 18, it doesn't much matter whether the data structures and associated functions are well defined. Such programs are so small that they are easy to "hack out" using any ad hoc approach that comes to mind.

As you get on to larger programs, it becomes more and more essential to provide well defined groupings of data and associated functionality. This is particularly important if several people must collaborate to build a program. It isn't practical to have several people trying to develop the same piece of code, the code has to be split up so that each programmer gets specific parts to implement.

Functional abstraction?

You *may* be able to split a program into separately implementable parts by considering the top level of a top-down functional decomposition. But usually, such parts are far from independent. They will need to share many different data structures. Often lower levels of the decomposition process identify requirements for similar routines for processing these structures; this can lead to duplication of code or to inconsistencies. In practice, the individuals implementing different "top level functions" wouldn't be able to go off and proceed independently.

or "data abstraction"

An alternative decomposition that focussed on the data might work better. Consider the information retrieval example (Section 18.3), you could give someone the task of building a "bit map component". This person would agree to provide a component that transferred bit maps between memory and disk, checked equality of bit maps, counted common bits in two maps, and performed any other necessary functions related to bit maps. Another component could have been a "text record component". This could have been responsible for displaying its contents, working with files, and (by courtesy of the bit map component) building a bit map that encoded the key words that were present in that text record.

The individuals developing these two components could work substantially independently. Some parts of a component might not be testable until all other components are complete and brought together but, provided that the implementors start by agreeing to the component interfaces, it is possible to proceed with the detailed design and coding of each component in isolation.

Data abstraction: its more than just a header file!

The code given in Chapter 18 had an interface (header) file that had a typedef for the "bit map" data and function prototypes for the routines that did things like set specific bits. A separate code file provided the implementations of those routines. As a coding style, this is definitely helpful; but it doesn't go far enough. After all, a header file basically describes the form of a data structure and informs other programmers of some useful functions, already implemented, that can be used to manipulate such structures. There is nothing to enforce controls; nothing to make programmers use the suggested interface.

Building a wall around data

Of course, if implementors are to work on separate parts of a program, they must stick to the agreed interfaces. Consider the example of programmers, A, B, C, and D who were working on a geometry modelling system. Programmer A was responsible for the coordinate component; this involved points (`struct pt { double x, y; }`) and functions like `offset_pt(double dx, double dy)`. Programmer B was uncooperative and did not want to wait for A to complete the points component. Instead of using calls to `offset_pt()` B wrote code that manipulated the data fields directly (`pt p; ... p.x += d1;`). B then left the project and was replaced by D.

Programmer C needed polar coordinates and negotiated with A for functions `radius(const pt p)` and `theta(const pt p)`. After implementing these functions, A and C did a timing analysis of their code and found that these new functions were being called very often and represented a measurable percentage of the run time.

Programmer A, who after all was officially responsible for the definition of points and their behaviour, updated things so that the struct contained polar as well as cartesian coordinates (`struct pt { double x, y, r, theta; };`). A changed the new `radius()` and `theta()` functions so that these simply returned the values of the extra data members and then changed all the functions like `offset_pt()` so that these updated the polar coordinates as well as the cartesian coordinates.

Programmers C and D duly recompiled their parts to reflect the fact that the definition of `struct pt` had changed with the addition of the extra data members.

Programmer A was fired because the code didn't work. C and D wasted more than a week on A's code before they found the incorrect, direct usage of the struct in the code originally written by B.

The problem here was that A couldn't define a "wall" around the data of a point structure. The header file had to describe the structure as well as the prototypes of A's routines. Since the structure was known, uncooperative programmers like B could access it.

Things would have been different if A could have specified things so that point structures could only be used via A's own routines. (In particular, A would still have a job, while B would probably have quit before the project really started.)

Modern programs typically require many "components". These components consist of the declaration of new type of data object and of the functions that are to be used to manipulate instances of this new type. You do require "walls around the data". When you define a new data type, you should be able to specify how it is to be used. If you can define components like this, then you can expect to be able to put a program together by combining separately developed components that work in accord with specified interfaces.

The need for components

It is possible use the older C language to implement components. Possible. But it is not always easy and it does depend on the discipline of the programmers. C doesn't enforce compliance with an interface. (One could argue that C doesn't enforce anything!) In contrast, the C++ language has extensive support for the programmer who needs to define and work with new data types.

Language support for reliable data types

The new language feature of C++ that supports a component oriented style of programming is "class". A "class" is meant to describe a conceptual entity; that is describe something that owns data and provides services related to its data.

Classes

A class declaration defines a new data type. The declaration does specify the form of a data structure and in this respect it is like a struct declaration. Once the compiler has seen the class declaration, it permits the definition of variables of this new type; these variables are normally referred to as "*instances of a class*" or "*objects*".

In addition to describing the data members, a class declaration allows a programmer to specify how objects that are instances of the class can be used. This is done by

identifying those functions that are allowed to access and change data members in an object. These functions are mainly "*member functions*". Member functions form part of the definition of the new data type.

The code for member functions is not normally included in the class declaration. Usually, a class declaration is in a header file; the code implementing the member functions will be in a separate code file (or files). However, at the point where they are defined, member functions clearly identify the class of which they are a part. Consequently, although the description of a class may be spread over a header file and several code files, it is actually quite easy to gather it all together in response to a request like "Show me how this code represents a 'Customer'."

Classes and design

Classes facilitate program design

The major benefit of classes is that they encourage and support a "component oriented" view of program development. Program development is all about repeatedly reanalyzing a problem, breaking it down into smaller more manageable pieces that can be thought about independently. The decomposition process is repeated until you know you have identified parts that are simple enough to be coded, or which may already exist. We have seen one version of this overall approach with the "top down functional decomposition" designs presented in Part III. Components (and the classes that describe them) give you an additional way of breaking up a problem.

Finding components

In fact, for large programs you should start by trying to identify the "components" needed in a program. You try to partition the problem into components that each own some parts of the overall data and perform all the operations needed on the data that they own. So you get "bit map" components, "hash table" components, "word" components and "text record" components. You characterize how the program works in terms of interactions among these components – e.g. "if the `hash_table` identifies a `vocab_item` that matches the word, then get the bit number from that `vocab_item` and ask the `bit_map` to set that bit". You have to identify all uses of a component. This allows you to determine what data that component owns and what functions it performs. Given this information, you can define the C++ class that corresponds to the identified component.

Designing the classes

Once you have completed an initial decomposition of the programming problem by identifying the required classes, you have to complete a detailed design for each one. But each class is essentially a separate problem. Instead of one big complex programming problem, you now have lots little isolated subproblems. Once the data and member functions of a class have been identified, most of the work on the design of a class relates to the design and implementation of its individual member functions.

Designing the member functions

When you have reached the level of designing an individual member function of a class you are actually back in the same sort of situation as you were with the examples in Parts II and III. You have a tiny program to write. If a member function is simple, just assignments, loops, and selection statements, it is going to be like the examples in

Part II where you had to code a single `main()`. If a member function has a more complex task to perform, you may have to identify additional auxiliary functions that do part of its work. So, the techniques of top down functional decomposition come into play. You consider coding the member function as a problem similar to those in Part III. You break the member function down into extra functions (each of which becomes an extra member function of the class) until you have simplified things to the point where you can implement directly using basic programming constructs.

Breaking a problem into separately analyzable component parts gives you a way of handling harder problems. But there are additional benefits. Once you start looking for component parts, you tend to find that your new program requires components that are similar to or identical to components that you built for other programs. Often you can simply reuse the previously developed components.

Reusing components

This is a bit like reusing algorithms by getting functions from a function library, such as those described in Chapter 13. However, the scale of reuse is greater. If you get a `sort()` function from a function library, you are reusing one function. If you get a ready built bitmap class, you get an integrated set of functions along with a model for a basic data structure that you require in your program. Increasingly, software developers are relying on "class libraries" that contain classes that define ready made versions of many commonly required components.

Issues relating to the decomposition of problems into manageable parts, and reuse, exist as an underlying leitmotif or theme behind all the rest of the materials in the part of the text.

Topics in this chapter

Section 19.1 introduces C++ classes. It covers the declaration of a simple class and the definition of its member functions. Classes can be complex. Many aspects are deferred to later chapters (and some aspects aren't covered in this text at all).

The next two sections present simple examples. Class `Bitmap`, section 19.2, is a more complete representation of the concept of a bit map data object similar to that introduced in the information retrieval example in section 19.3. Class `Number`, section 19.3, represents an integer. You might think that with shorts, longs, (and long longs) there are already enough integers. Those represented as instances of class `Number` do however have some rather special properties.

19.1 CLASS DECLARATIONS AND DEFINITIONS

19.1.1 Form of a class declaration

A class declaration introduces the name of a new data type, identifies the data members present in each variable of this type (i.e. each "class instance" or "object"), identifies the

member functions that may be used to manipulate such variables, and describes the access (security) controls that apply. A declaration has the form:

```
class Name {
    details ...
    more details ...
};
```

The declaration starts with the keyword `class`. The name of the class is then given. The usual naming rules apply, the name starts with a letter and contains letters, digits and underscore characters. (Sometimes, there may be additional naming conventions. Thus you may encounter an environment where you are expected to start the name of a class with either 'T' or 'C'. Such conventions are not followed here.)

The body of the declaration, with all the details, comes between { and } brackets. The declaration ends with a semicolon. Compilers get really upset, and respond with incomprehensible error messages, if you omit that final semicolon.

The details have to include a list of the associated functions and data members. Initially the examples will be restricted slightly, the associated functions will all be "member functions", that is they all are functions that are inherently part of the definition of the concept represented by the class. (There are other ways of associating functions with classes, they get touched on later in Section 23.2.) These lists of member functions and data members also specify the access controls.

**Keywords 'public'
and 'private'**

There are only two kinds of access that need be considered at this stage. They are defined by the keywords `public` and `private`. A `public` member function (or data member) is one that can be used anywhere in the code of the program. A `private` data member (or member function) is one that can only be used within the code of the functions identified in the class declaration itself. Note the wording of the explanation. Generally, data members are all `private`, there may also be a few `private` member functions. The main member functions are all `public` (sometimes, but very rarely, there may be `public` data members).

You can specify the appropriate access control for each data member and member function individually, or you can have groups of `public` members interspersed with groups of `private` members. But most often, the following style is used:

```
class Name {
public:
    details of the "public interface" of the class
private:
    private implementation details
    and description of data members of each instance
    of the class...
};
```

In older text books, you may find the order reversed with the `private` section defined first. It is better to have the `public` interface first. Someone who wants to know how

to use objects of this class need only read the `public` interface and can stop reading at the keyword `private`.

The following examples are just to make things a little more concrete (they are simplified, refinements and extensions will be introduced later). First, there is a class defining the concept of a point in two dimensional space (such as programmer A, from the earlier example, might have wished to define):

```
class Point {
public:
    ...
    // Get cartesian coords
    double X();
    double Y();
    // Get polar coords
    double Radius();
    double Theta();
    // Test functions
    int ZeroPoint();
    int InFirstQuad();
    ...
    // Modify
    void Offset(double deltaX, double deltaY);
    ...
    void SetX(double newXval);
    ...
    // Comparisons
    int Equal(const Point& other);
    ...
private:
    void FixUpPolarCoords();
    ...
    double fX, fY, fR, fTheta;
};
```

(The ellipses, "...", indicate places where additional member functions would appear in actual class declaration; e.g. there would be several other test functions, hence the ellipsis after function `InFirstQuad()`. The ellipsis after the `public` keyword marks the place where some special initialization functions would normally appear; these "constructor" functions are described in section 19.1.4 below.)

The public interface specifies what other programmers can do with variables that are instances of this type (class). Points can be asked where they are, either in cartesian or polar coordinate form. Points can be asked whether they are at the coordinate origin or whether they are in a specified quadrant (functions like `ZeroPoint()` and `InFirstQuad()`). Points can be asked to modify their data members using functions like `Offset()`. They may also be asked to compare themselves with other points through functions like `Equal()`.

*The public role of
Points*

The private section of the class declaration specifies things that are concern of the programmer who implemented the class and of no one else. The data members

*The private lives of
Points*

generally appear here. In this case there are the duplicated coordinate details – both cartesian and polar versions. For the class to work correctly, these data values must be kept consistent at all times.

There could be several functions that change the cartesian coordinate values (`Offset()`, `SetX()` etc). Every time such a change is made, the polar coordinates must also be updated. During the detailed design of the class, the code that adjusts the polar coordinates would be abstracted out of the functions like `SetX()`. The code becomes the body for the distinct member function `FixUpPolarCoords()`.

This function is private. It is an implementation detail. It should only be called from within those member functions of the class that change the values of the cartesian coordinates. Other programmers using points should never call this function; as far as they are concerned, a point is something that always has consistent values for its polar and cartesian coordinates.

For a second example, how about class `Customer` (based loosely on the example problem in Section 17.3):

```
class Customer {
public:
    ...
    // Disk transfers
    int    ReadFrom(istream& input);
    int    WriteTo(ofstream& output);
    // Display
    void    PrintDetails(ofstream& out);
    // Query functions
    int    Debtor();
    ...
    // Changes
    void    GetCustomerDetails();
    void    GetOrder();
    ...
private:
    UT_Word    fName;
    ...
    UT_Word    fOrders[kMAXITEMS];
    int        fNumOrder;
    double     fAmountOwing;
    Date       fLastOrder;
};
```

Class `Customer` should define all the code that manipulates customer records. The program in Section 17.3 had to i) transfer customer records to/from disk (hence the `WriteTo()` and `ReadFrom()` functions, ii) display customer details (`PrintDetails()`), iii) check details of customers (e.g. when listing the records of all who owed money, hence the `Debtor()` function), and had to get customer details updated. A `Customer` object should be responsible for updating its own details, hence the member functions like `GetOrder()`.

The data members in this example include arrays and a struct (in the example in Section 17.3, `Date` was defined as a struct with three integer fields) as well as simple data types like `int` and `double`. This is normal; class `Point` is atypical in having all its data members being variables of built in types.

There is no requirement that the data members have names starting with 'f'. It is simply a useful convention that makes code easier to read and understand. A name starting with 'g' signifies a global variable, a name with 's' is a "file scope" static variable, 'e' implies an enumerator, 'c' or 'k' is a constant, and 'f' is a data member of a class or struct. There are conventions for naming functions but these are less frequently adhered to. Functions that ask for yes/no responses from an object (e.g. `Debtor()` or `InFirstQuad()`) are sometimes given names that end with '_p' (e.g. `Debtor_p()`); the 'p' stands for "predicate" (dictionary: 'predicate' assert or affirm as true or existent). `ReadFrom()` and `WriteTo()` are commonly used as the names of the functions that transfer objects between memory and disk. Procedures (void functions) that perform actions may all have names that involve or start with "Do" (e.g. `DoPrintDetails()`). You will eventually convince yourself of the value of such naming conventions; just try maintaining some code where no conventions applied.

*Minor point on
naming conventions*

19.1.2 Defining the member functions

A class declaration promises that the named functions will be defined somewhere. Actually, you don't have to define all the functions, you only have to define those that get used. This allows you to develop and test a class incrementally. You will have specified the class interface, with all its member functions listed, before you start coding; but you can begin testing the code as soon as you have a reasonable subset of the functions defined.

Usually, class declarations go in header files, functions in code files. If the class represents a single major component of the program, e.g. class `Customer`, you will probably have a header file `Customer.h` and an implementation file `Customer.cp`. Something less important, like class `Point`, would probably be declared in a header file ("`geom.h`") along with several related classes (e.g. `Point3D`, `Rectangle`, `Arc`, ...) and the definitions of the member functions of all these classes would be in a corresponding `geom.cp` file.

A member function definition has the general form:

```
return type
class_name::function_name(arguments)
{
    body of function
}
```

e.g.


```

void Customer::PrintDetails(ofstream& out)
{
    ...
}

double
Point::Radius()
{
    ...
}

```

(Some programmers like the layout style where the return type is on a separate line so that the class name comes at the beginning of the line.)

**Scope qualifier
operator**

The double colon `::` is the "scope qualifier" operator. Although there are other uses, the main use of this operator is to associate a class name with a function name when a member function is being defined. The definitions of all functions of class `Point` will appear in the form `Point::function_name`, making them easy to identify even if they occur in a file along with other definitions like `Point3D::Radius()` or `Arc::ArcAngle()`.

Some examples of function definitions are:

```

double Point::Radius()
{
    return fR;
}

void Point::SetX(double newXval)
{
    fX = newXval;
    FixUpPolarCoords();
}

void Point::Offset(double deltaX, double deltaY)
{
    fX += deltaX;
    fY += deltaY;
    FixUpPolarCoords();
}

```

Now you may find something slightly odd about these definitions. The code is simple enough. `Offset()` just changes the `fX` and `fY` fields of the point and then calls the private member function `FixUpPolarCoords()` to make the `fR`, and `fTheta` members consistent.

The oddity is that it isn't clear which `Point` object is being manipulated. After all, a program will have hundreds of `Points` as individual variables or elements of arrays of `Points`. Each one of these points has its own individual `fX` and `fY` data members.

The code even looks as if it ought to be incorrect, something that a compiler should stomp on. The names `fX` and `fY` are names of data members of something that is like a

struct, yet they are being used on their own while in all previous examples we have only used member names as parts of fully qualified names (e.g. `thePt.fX`).

If you had been using the constructs illustrated in Part III, you could have had a struct `Pt` and function `OffsetPt()` and `FixPolars()`:

```
struct Pt { double x, y, r, t; };

void OffsetPt(Pt& thePoint, double dx, double dy)
{
    thePoint.x += dx;
    thePoint.y += dy;
    FixPolars(thePoint);
}

void FixPolars(Pt& thePt)
{
    ...
}
```

The functions that change the `Pt` struct have a `Pt&` (reference to `Pt`) argument; the value of this argument determines which `Pt` struct gets changed.

This is obviously essential. These functions change structs (or class instances). The functions have to have an argument identifying the one that is to be changed. However, no such argument is apparent in the class member function definitions.

The class member functions do have an *implicit* extra argument that identifies the object being manipulated. The compiler adds the address of the object to the list of arguments defined in the function prototype. The compiler also modifies all references to data members so that they become fully qualified names.

*Implicit argument
identifying the object
being manipulated*

Rather than a reference parameter like that in the functions `OffsetPt()` and `FixPolars()` just shown, this extra parameter is a "pointer". Pointers are covered in the next chapter. They are basically just addresses (as are reference parameters). However, the syntax of code using pointers is slightly different. In particular a new operator, `->`, is introduced. This "pointer operator" is used when accessing a data member of a structure identified by a pointer variable. A few examples appear in the following code. More detailed explanations are given in the next chapter.

*A "pointer
parameter" identifies
the object*

You can imagine the compiler as systematically changing the function prototypes and definitions. So, for example:

```
void Point::SetX(double newXval)
{
    fX = newXval;
    FixUpPolarCoords();
}
```

gets converted to

```
void __Point__SetX_ptsd(Point* this, double newXval)
{
    this->fX = newXval;
    this->FixUpPolarCoords();
}
```

while

```
int Customer::Debtor()
{
    return (fAmountOwing > 0.0);
}
```

gets converted to

```
int __Customer__Debtor_ptsv(Customer* this)
{
    return (this->fAmountOwing > 0.0);
}
```

The `this` pointer argument will contain the address of the data, so the compiler can generate code that modifies the appropriate `Point` or checks the appropriate `Customer`.

Normally, you leave it to the compiler to put in "`this->`" in front of every reference to a data member or member function; but you can write the code with the "`this->`" already present. Sometimes, the code is easier to understand if the pointer is explicitly present.

19.1.3 Using class instances

Once a class declaration has been read by the compiler (usually as the result of processing a `#include` directive for a header file with the declaration), variables of that class type can be defined:

```
#include "Customer.h"
...
void RunTheShop()
{
    Customer theCustomer;
    ...
}
```

or

```
#include "geom.h"
```

```
void DrawLine(Point& p1, Point& p2)
{
    Point  MidPoint;
    ...
}
```

Just like structs, variables of class types can be defined, get passed to functions by value or by reference, get returned as a result of a function, and be copied in assignment statements.

Most code that deals with structs consists of statements that access and manipulate individual data members, using qualified names built up with the "." operator. Thus in previous examples we have had things like:

```
cout << theVocabItem.fWord;

strcpy(theTable[ndx].fWord, aWord);
```

With variables of class types you can't go around happily accessing and changing their data; the data are walled off! You have to ask a class instance to change its data or tell you about the current data values:

```
Point      thePoint;
...
thePoint.SetX(5.0);
...
if(thePoint.X() < 0.0) ...
```

These requests, calls to member functions, use much the same syntax as the code that accessed data members of structures. A call uses the name of the object qualified by the name of the function (and its argument list).

You can guess how the compiler deals with things. You already know that the compiler has built a function:

```
void __Point__SetX_ptsd(Point* this, double newXval);
```

from the original `void Point::SetX(double)`. A call like

```
thePoint.SetX(5.0);
```

gets converted into:

```
__Point__SetX_ptsd(&thePoint, 5.0);
```

(The & "get the address of" operator is again used here to obtain the address of the point that is to be changed. This address is then passed as the value of the implicit `Point* this` parameter.)

19.1.4 Initialization and "constructor" functions

Variables should be initialized. Previous examples have shown initialization of simple variables, arrays of simple variables, structs, and arrays of structs. Class instances also need to be initialized.

For classes like class `Point` and class `Customer`, initialization would mean setting values in data fields. Later, Chapter 23, we meet more complex "resource manager" classes. Instances of these classes can be responsible for much more than a small collection of data; there are classes whose instances own opened files, or that control Internet connections to programs running on other computers, or which manage large collections of other data objects either in memory or on disk. Initialization for instances of these more complex structures may involve performing actions in addition to setting a few values in data members.

constructors

So as to accommodate these more complex requirements, C++ classes define special initialization functions. These special functions are called "constructors" because they are invoked as an instance of the class is created (or constructed).

There are some special rules for constructors. The name of the function is the same as the name of the class. A constructor function does not have any return type.

We could have:

```
class Point {
public:
    Point(double initialX = 0.0, double initialY = 0.0);
    ...
};

class Customer {
public:
    Customer();
    ...
};
```

The constructor for class `Point` has two arguments, these are the initial values for the X, Y coordinates. Default values have been specified for these in the declaration. The constructor for class `Customer` takes no arguments.

These functions have to have definitions:

```
Point::Point(double initialX, double initialY)
{
    fX = initialX;
    fY = initialY;
    FixUpPolarCoords();
}
```

```

Customer::Customer()
{
    strcpy(fName, "New customer");
    strcpy(fAddress, "Delivery address");
    ...
    fAmountOwing = 0.0;
}

```

Here, the constructors do involve calls to functions. The constructor for class `Point` initializes the `fX`, `fY` data members but must also make the polar coordinates consistent; that can be accomplished by a call to the member function `FixUpPolarCoords()`. The constructor for `Customer` makes calls to `strcpy` from the string library.

A class can have an overloaded constructor; that is there can be more than one constructor function, with the different versions having different argument lists. The class `Number`, presented in Section 19.3, illustrates the use of multiple constructors.

*Overloaded
constructors*

Constructors can get quite elaborate and can acquire many specialized forms. Rather than introduce these now in artificial examples, these more advanced features will be illustrated later in contexts where the need arises.

When you need to define initialized instances of a class, you specify the arguments for the constructor function in the definition of the variable. For example:

```

Point    p1(17.0, 0.5);
Point    p2(6.4);
Point    p3;

```

Variable `p1` is a `Point` initially located at 17.0, 0.5; `p2` is located at 6.4, 0.0 (using the default 0.0 value for `initialY`). `Point p3` is located at 0.0, 0.0 (using the defaults for both `initialX` and `initialY`).

19.1.5 const member functions

Some member functions change objects, others don't. For example:

```

double Point::Y()
{
    return fY;
}

```

doesn't change the `Point` for which this function is invoked; whereas the `Point` is changed by:

```

void Point::SetY(double newYval)
{
    fY = newYval;
    FixUpPolarCoords();
}

```

```
}
```

It is worthwhile making this clear in the class declaration:

```
class Point {
public:
    ...
    // Get cartesian coords
    double X() const;
    double Y() const;
    // Get polar coords
    double Radius() const;
    double Theta() const;
    // Test functions
    int ZeroPoint() const;
    int InFirstQuad() const;
    ...
    // Modify
    void Offset(double deltaX, double deltaY);
    void SetX(double newXval);
    ...
    // Comparisons
    int Equal(const Point& other) const;
```

Functions that don't change the object should be qualified by the keyword `const` both in the declaration, and again in their definitions:

```
double Point::Radius() const
{
    return fR;
}
```

const class instances

Apart from acting as an additional documentation cue that helps users understand a class, `const` functions are necessary if the program requires `const` instances of a class.

Now constancy isn't something you often want. After all, a `const Customer` is one who never places an order, never sets an address, never does anything much that is useful. But there are classes where `const` instances are meaningful. You can imagine uses for `const Points`:

```
const Point MinWindowSize(100.0, 100.0);
```

But there is a catch. The compiler is supposed to enforce constancy. So what is the compiler to do with code like:

```
// I need a bigger window
MinWindowSize.SetX(150.0);
// Test against window limits
if(xv < MinWindowSize.X()) ...
```

The second is a legitimate use of the variable `MinWindowSize`, the first changes it. The first should be disallowed, the second is permitted.

The compiler can't know the rules unless you specify them. After all, the code for class `Point` may have been compiled years ago, all the compiler has to go on is the description in the header file. The compiler has to assume the worst. If you don't tell it that a member function leaves an object unchanged, the compiler must assume that it is changed. So, by default, a compiler refuses to let you use class member functions to manipulate `const` instances of that class.

If your class declaration identifies some members as `const` functions, the compiler will let you use these with `const` instances.

19.1.6 inline member functions

Because the data in a class instance are protected, access functions have to be provided, e.g.:

```
double Point::X() const
{
    return fX;
}
```

Code needing the x-coordinate of a point must call this access function:

```
Point p1;
...
while(p1.X() > kMIN) { ...; p1.Offset(delta, 0.0); }
```

Calling a function simply to peek at a data member of an object can be a bit costly. In practice, it won't matter much except in cases where the access function is invoked in some deeply nested inner loop where every microsecond counts.

Now inline functions were invented in C++ to deal with situations where you want function semantics but you don't want to pay for function overheads. Inlines can be used to solve this particular version of the problem.

As always, the definition of an inline function has to have been read so that the compiler can expand a function call into actual code. Since classes are going to be declared in header files that get `#included` in all the code files that use class instances, the inline functions definitions must be in the header file.

The following is the preferred style:

```
#ifndef __MYGEOM__
#define __MYGEOM__

class Point {
public:
```



```

    ...
    double X();
    ...
private:
    ...
    double fX, fY, fR, fTheta;
};

other class declarations if any, e.g. class Arc { };

inline double Point::X() { return fX; }

#endif

```

Any inline functions get defined at the end of the file.

An alternative style is:

```

#ifndef __MYGEOM__
#define __MYGEOM__

class Point {
public:
    ...
    double X() { return this->fX; }
    ...
private:
    ...
    double fX, fY, fR, fTheta;
};

other class declarations if any, e.g. class Arc { };
#endif

```

Here, the body of the inline function appears in the function declaration (note, an explicit `this->` qualifier on the data members is highly advisable when this style is used). The disadvantage of this style is that it makes the class declaration harder to read. Remember, your class declaration is going to be read by your colleagues. They want to know what your class can do, they don't really care how the work gets done. If you put the function definitions in the class declaration itself, you force them to look at the gory details. In contrast, your colleagues can simply ignore any inlines defined at the end of a header file, they know that these aren't really their concern.

19.2 EXAMPLE: BITMAPS (SETS)

The information retrieval example, Section 18.3, illustrated one simple use of a bitmap in an actual application. The bitmaps in that example used '1/0' bits to indicate whether a document contained a specific keyword.

Really, the example illustrated a kind of "set". The bitmap for a document represented the "set of keywords" in that document. The system had a finite number of keywords, so there was a defined maximum set size. Each possible keyword was associated with a specific bit position in the set.

Such sets turn up quite often. So a "bit map" component is something that you might expect to "reuse" in many applications.

This section illustrates the design and implementation of a class that represents such bitmaps and provides the functions necessary to implement them. The implementation is checked out with a small test program.

Designing a general component for reuse is actually quite hard. Normally, you have a specific application in mind and the functionality that you identify is consequently biased to the needs of that application. When you later attempt to reuse the class, you find some changes and extensions are necessary. It has been suggested that a reusable class is typically created, and then reworked completely a couple of times before it becomes a really reusable component. So, the class developed here may not exactly match the needs of your next application, but it should be a good starting point.

Designing a simple class

This example serves as an introduction to the design of a class. Remember a class is a C++ description of a particular type of component that will be used in programs. Objects that are instances of the class will own some of the program's data and will provide all the services related to the owned data. The place to start designing a class is deciding what data its instances own and what services they provide.

What data does an individual bitmap own? Each instance of class `Bitmap` will have an array of unsigned long integers. There shouldn't be any other data members. To make things simple, we can have the `Bitmap` class use a fixed size array, with the size defined by a `#define` constant.

The data owned

The next step is to decide what "services" a bitmap object should provide. You make up a list of useful functions. The list doesn't have to be complete, you can add further functionality later. Some useful services of a bitmap would be:

*Services provided by
a bitmap*

- Zeroing; reset all the bits to zero (the constructor function should call this, bit maps should start with all their bits set to 0).
- Set bit *i*; and Clear bit *i*;
Need functions to set (make '1') and clear (make '0') a specified bit. This could be done by having separate `set()` and `clear()` functions, or a `set_as()` function that had two arguments – the bit number and the setting. Actually, it would probably be convenient if the class interface offered both forms because sometimes one form is more convenient than the other.
- Test bit *i*;

Will often want to test the status of an individual bit.

- Flip bit i;
Change setting of a specified bit, if it was '1' it becomes '0' and vice versa.
- ReadFrom and WriteTo
Will want to transfer the data as a block of binary information.
- PrintOn
Produce a (semi)-readable printout of a bitmap as a series of hex values.
- Count
Get Bitmap to report how many bits it has set to '1'.
- Invert
Not clear whether this function would be that useful, but it might. The invert function would flip all the bits; all the '1's become '0's, all the '0's become '1's (the bitmap applies a "Not" operation to itself).

The next group of functions combine two bitmaps to produce a third as a result. Having a class instance as a function's result is just the same as having a struct as a result. As explained for structs, you must always consider whether this is wise. It can involve the use of lots of space on the stack and lots of copying operations. In this case it seems reasonable. Bitmap objects aren't that large and the function semantics are sensible.

- "And" with bit map "other"
Returns the bitmap that represents the And of the bitmap with a second bitmap.
- "Inclusive Or" with bitmap "other".
- "Exclusive Or" with bitmap "other".

and for consistency

- "Not"
Returns a bit map that is the inverse of this bitmap.

Finally, we should probably include an "Equals" function that checks equality with a second bitmap.

The discussion of design based on top down functional decomposition (Chapter 15) favoured textual rather than diagrammatic representation. Diagrams tend to be more useful in programs that use classes. Simple diagrams like that shown in Figure 19.1 can provide a concise summary of a class.

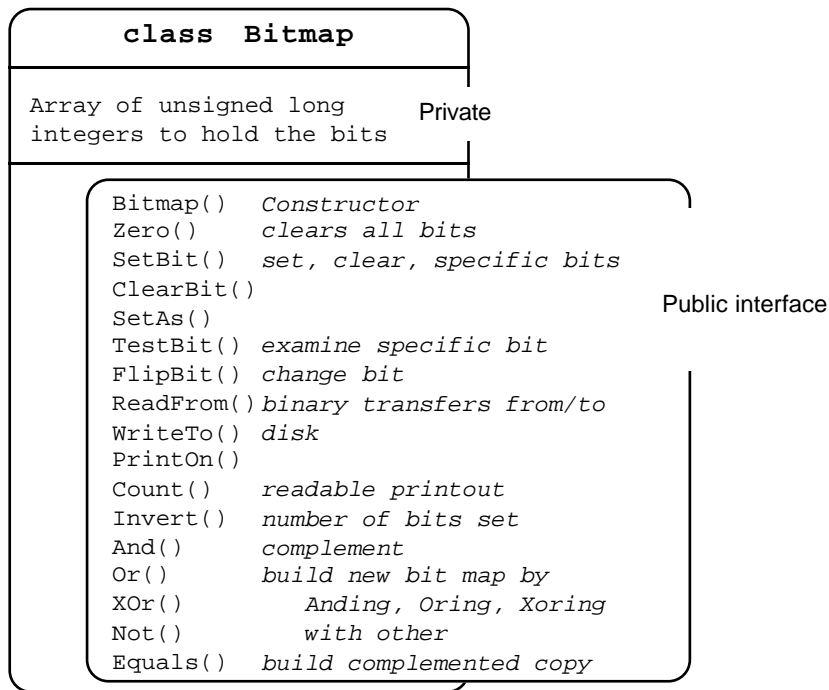


Figure 19.1 A simple preliminary design diagram for a class.

The figure shows the class with details of its private data and functions all enclosed within the class boundary. The public member functions stick outside the boundary. In this example, all the member functions are public.

The first iteration through the design of a class is complete when you have composed a diagram like Figure 19.1 that summarizes the resources owned by class instances and their responsibilities.

The next stage in class design involves firming up the function prototypes, deciding on the arguments, return types, and const status. (There may not be much use for const Bitmaps, but you should still make the distinction between access functions that merely look at the data of a class member and modifying procedures that change data.)

In general, it would also be necessary to produce pseudo-code outlines for each function. This isn't done here as the functions are either trivial or are similar to those explained in detail in Section 18.3.

The prototypes become:

```

void    Bitmap();
void    Zero(void);
void    SetBit(int bitnum);
void    ClearBit(int bitnum);
void    SetAs(int bitnum, int setting);
    
```

```

int      TestBit(int bitnum) const;
void     FlipBit(int bitnum);
void     ReadFrom(fstream& in);
void     WriteTo(fstream& out) const;
void     PrintOn(ostream& printer) const;
int      Count(void) const;
void     Invert(void);

Bitmap   And(const Bitmap& other) const;
Bitmap   Or(const Bitmap& other) const;
Bitmap   XOr(const Bitmap& other) const;

Bitmap   Not(void) const;

int      Equals(const Bitmap& other) const;

```

About the only point to note are the `const Bitmap&` arguments in functions like `And()`. We don't want to have `Bitmaps` passed by value (too much copying of data) so pass by reference is appropriate. The "And" operation does not affect the two bitmaps it combines, so the argument is `const`. The i/o functions take `fstream` reference arguments.

Only 0 and 1 values are appropriate for argument setting in function `SetAs()`. It would be possible to define an enumerated type for this, but that seems overkill. The coding can use 0 to mean 0 (reasonable) and non-zero to mean 1.

Implementation

Header file for class declaration

The `Bitmap` class will be declared in a header file `bitmap.h`:

```

#ifndef __MYBITSCLASS__
#define __MYBITSCLASS__

// Code use iostream and fstream,
// #include these if necessary, fstream.h
// does a #include on iostream.h
#ifndef __FSTREAM_H
#include <fstream.h>
#endif

// Code assumes 32-bit unsigned long integers
#define MAXBITS 512
#define NUMWORDS 16

typedef unsigned long Bits;

class Bitmap {
public:
    Bitmap();

```

```

    void    Zero(void);
    void    SetBit(int bitnum);
    void    ClearBit(int bitnum);
    void    SetAs(int bitnum, int setting);
    int     TestBit(int bitnum) const;
    void    FlipBit(int bitnum);
    void    ReadFrom(fstream& in);
    void    WriteTo(fstream& out) const;
    void    PrintOn(ostream& printer) const;
    int     Count(void) const;
    void    Invert(void);

    Bitmap  And(const Bitmap& other) const;
    Bitmap  Or(const Bitmap& other) const;
    Bitmap  XOr(const Bitmap& other) const;

    Bitmap  Not(void) const;

    int     Equals(const Bitmap& other) const;
private:
    Bits    fBits[NUMWORDS];
};

#endif

```

This header file has a `#include` on `fstream.h`. Attempts to compile a file including `bitmap.h` without `fstream.h` will fail when the compiler reaches the i/o functions. Since there is this dependency, `fstream.h` is `#included` (note the use of conditional compilation directives, if `fstream.h` has already been included, it isn't read a second time). There is no need to `#include` `iostream.h`; the `fstream.h` header already checks this.

The functions implementing the Bitmap concept are defined in `bitmap.c`:

*Function definitions
in the separate code
file*

```

#include "bitmap.h"

Bitmap::Bitmap()
{
    Zero();
}

void Bitmap::Zero(void)
{
    for(int i=0; i<NUMWORDS; i++)
        fBits[i] = 0;
}

```

The constructor, `Bitmap::Bitmap()`, can use the `Zero()` member function to initialize a bitmap. Function `Zero()` just needs to loop zeroing out each array element.

```

void Bitmap::SetBit(int bitnum)

```

```

{
    if((bitnum < 0) || (bitnum >= MAXBITS))
        return;
    int word = bitnum / 32;
    int pos = bitnum % 32;

    Bits mask = 1 << pos;
    fBits[word] |= mask;
}

```

Function `SetBit()` uses the mechanism explained in 18.3. Function `ClearBit()` has to remove a particular bit. It identifies the array element with the bit. Then, it sets up a mask with the specified bit set. Next it complements the mask so that every bit except the specified bit is set. Finally, this mask is "Anded" with the array element; this preserves all bits except the one that was to be cleared.

```

void Bitmap::ClearBit(int bitnum)
{
    if((bitnum < 0) || (bitnum >= MAXBITS))
        return;
    int word = bitnum / 32;
    int pos = bitnum % 32;

    Bits mask = 1 << pos;
    mask = ~mask;
    fBits[word] &= mask;
}

```

Function `SetAs()` can rely on the implementation of `ClearBit()` and `SetBit()`:

```

void Bitmap::SetAs(int bitnum, int setting)
{
    if(setting == 0)
        ClearBit(bitnum);
    else SetBit(bitnum);
}

```

Function `TestBit()` uses the same mechanism for identifying the array element and bit and creating a mask with the chosen bit set. This mask is "Anded" with the array element. If the result is non-zero, the chosen bit must be set.

```

int Bitmap::TestBit(int bitnum) const
{
    if((bitnum < 0) || (bitnum >= MAXBITS))
        return 0;
    int word = bitnum / 32;
    int pos = bitnum % 32;

    Bits mask = 1 << pos;

```

```

        return (fBits[word] & mask);
    }

```

Function `FlipBit()` uses an exclusive or operation to change the appropriate bit in the correct word (check that you understand how the xor operation achieves the desired result):

```

void Bitmap::FlipBit(int bitnum)
{
    if((bitnum < 0) || (bitnum >= MAXBITS))
        return;
    int word = bitnum / 32;
    int pos = bitnum % 32;

    Bits mask = 1 << pos;
    fBits[word] ^= mask;
}

```

Functions `ReadFrom()` and `WriteTo()` perform the binary transfers that copy the entire between file and memory:

```

void Bitmap::ReadFrom(fstream& in)
{
    in.read(&fBits, sizeof(fBits));
}

void Bitmap::WriteTo(fstream& out) const
{
    out.write(&fBits, sizeof(fBits));
}

```

These functions do not check for transfer errors. That can be done in the calling environment. When coding class `Bitmap`, you don't know what should be done if a transfer fails. (Your IDE's version of the `iostream` library, and its compiler, may differ slightly; the calls to read and write may need `"(char*)"` inserted before `&fBits`.)

The files produced using `WriteTo()` are unreadable by humans because they contain the "raw" binary data. Function `PrintOn()` produces a readable output:

```

void Bitmap::PrintOn(ostream& printer) const
{
    long savedformat = printer.flags();
    printer.setf(ios::showbase);
    printer.setf(ios::hex, ios::basefield);

    for(int i = 0; i < NUMWORDS; i++) {
        printer.width(12);
        printer << fBits[i];
        if((i % 4) == 3) cout << endl;
    }
}

```



```

    }
    printer.flags(savedformat);
}

```

Function `PrintOn()` has to change the output stream so that numbers are printed in hex. It would be somewhat rude to leave the output stream in the changed state! So `PrintOn()` first uses the `flags()` member function of `ostream` to get the current format information. Before returning, `PrintOn()` uses another overloaded version of the `flags()` function to set the format controls back to their original state.

```

int Bitmap::Count(void) const
{
    int count = 0;
    for(int n=0; n < NUMWORDS; n++) {
        Bits x = fBits[n];
        int j = 1;
        for(int i=0; i<32; i++) {
            if(x & j)
                count++;
            j = j << 1;
        }
    }
    return count;
}

```

The `Count()` function uses a double loop, the outer loop steps through the array elements, the inner loop checks bits in the current element. It might be worth changing the code so that the inner loop was represented as a separate `CountBitsInElement()` function. This would be a private member function of class `Bitmap`.

Functions `Invert()`, `Not()` and `Equals()` all have similar loops that check, change, or copy and change successive array elements from `fBits`:

```

void Bitmap::Invert(void)
{
    for(int i=0; i < NUMWORDS; i++)
        fBits[i] = ~fBits[i];
}

Bitmap Bitmap:: Not(void) const
{
    Bitmap b;
    for(int i = 0; i < NUMWORDS; i++)
        b.fBits[i] = ~this->fBits[i];
    return b;
}

int Bitmap::Equals(const Bitmap& other) const
{

```

```

    for(int I = 0; I < NUMWORDS; i++)
        if(this->fBits[I] != other.fBits[i]) return 0;
    return 1;
}

```

Note the explicit use of the `this->` qualifier in `Not()` and `Equals()`. These functions manipulate more than one `Bitmap`, and hence more than one `fBits` array. There are the `fBits` data member of the object executing the function, and that of some second object. The `this->` qualifier isn't needed but sometimes it makes things clearer.

You will also note that the `Bitmap` object that is performing the `Equals()` operation is looking at the `fBits` data member of the second `Bitmap` (`other`). What about the "walls" around `other`'s data?

*Accessing data
members of another
object of the same
class*

Classes basically define families of objects and there are no secrets within families. An object executing code of a member function of the class is permitted to look in the private data areas of any other object of that class.

The functions `And()`, `Or()`, and `XOr()` are very similar in coding. The only difference is the operator used to combine array elements:

```

Bitmap Bitmap::And(const Bitmap& other) const
{
    Bitmap b;
    for(int i = 0; i < NUMWORDS; i++)
        b.fBits[i] = this->fBits[i] & other.fBits[i];
    return b;
}

Bitmap Bitmap::Or(const Bitmap& other) const
{
    Bitmap b;
    for(int i = 0; i < NUMWORDS; i++)
        b.fBits[i] = this->fBits[i] | other.fBits[i];
    return b;
}

Bitmap Bitmap::XOr(const Bitmap& other) const
{
    Bitmap b;
    for(int i = 0; i < NUMWORDS; i++)
        b.fBits[i] = this->fBits[i] ^ other.fBits[i];
    return b;
}

```

It isn't sufficient to write the class, you must also test it! The test program should automatically exercise all aspects of the class. This test program becomes part of the class documentation. It has to be available to other programmers who may need to extend or modify the existing class and who will need to retest the code after their modifications.

Test program

The test program for class Bitmap is:

```
int main()
{
    Test1();
    Test2();
    Test3();
    return EXIT_SUCCESS;
}
```

There are three Test functions (and an auxiliary function). As the names imply, the functions were developed and implemented in sequence. Basic operations were checked out using Test1() before the code of Test2() and Test3() was implemented.

The first test function checks out simple aspects like setting and clearing bits and getting a Bitmap printed:

```
void Test1()
{
    // Try setting some bits and just printing the
    // resulting Bitmap
    int somebits[] = { 0, 3, 4, 6, 9, 14, 21, 31,
                      32, 40, 48, 56,
                      64, 91, 92, 93, 94, 95,
                      500, 501
                    };

    int n = sizeof(somebits) / sizeof(int);
    Bitmap b1;
    Set bits      SetBits(b1, somebits, n);

    cout << "Test 1" << endl;
    Print        cout << "b1:" << endl;
                b1.PrintOn(cout);

    Check the count
    function      cout << "Number of bits set in b1 is " << b1.Count()
                << endl;
                cout << "(that should have said " << n << ")" << endl;

    and the test bit
    function      if(b1.TestBit(20))
                cout << "Strange I didn't think I set bit 20" << endl;

                if(!b1.TestBit(3))
                cout << "Something just ate my bitmap" << endl;

    Inversion    b1.Invert();

                cout << "Look at b1, it's flipped" << endl;
                b1.PrintOn(cout);

                b1.Zero();
}
```

```

    SetBits(b1, somebits, n);
    cout << "sanity restored" << endl;

    b1.ClearBit(4);
    b1.FlipBit(6);
    b1.FlipBit(7);

    cout << "Check for three small changes:" << endl;
    b1.PrintOn(cout);
}

```

*Check other bit flips
etc*

An auxiliary function `SetBits()` was defined to make it easier to set several bits:

```

void SetBits(Bitmap& theBitmap, int bitstoset[], int num)
{
    for(int i=0; i<num; i++) {
        int b = bitstoset[i];
        theBitmap.SetBit(b);
    }
}

```

Function `Test2()` checks the transfers to/from files, and also the `Equals()` function:

```

void Test2()
{
    int somebits[] = { 0, 1, 2, 3, 5, 7, 11, 13,
                      17, 19, 23, 29, 31, 37,
                      41, 47
                      };
    int n = sizeof(somebits) / sizeof(int);
    Bitmap b1;
    SetBits(b1, somebits, n);
    fstream tempout("tempbits", ios::out);
    if(!tempout.good()) {
        cout << "Sorry, Couldn't open temporary output file"
              << endl;
        exit(1);
    }
    b1.WriteTo(tempout);
    if(!tempout.good()) {
        cout << "Seem to have had problems writing to file"
              << endl;
        exit(1);
    }
    tempout.close();
    Bitmap b2;
    fstream tempin("tempbits", ios::in | ios::nocreate);
    if(!tempin.good()) {
        cout << "Sorry, couldn't open temp file with"

```

Write a bit map to file

*Read a bitmap from
file*

```

        " the bits" << endl;
        exit(1);
    }
    b2.ReadFrom(tempin);
    if(!tempin.good()) {
        cout << "Seem to have had problems reading from"
              " file" << endl;
        exit(1);
    }
    tempin.close();

    cout << "I wrote the bitmap : " << endl;
    b1.PrintOn(cout);
    cout << "I read the bitmap : " << endl;
    b2.PrintOn(cout);

    Check equality
    if(b1.Equals(b2)) cout << "so i got as good as i gave" <<
                        endl;
    else cout << "which is sad" << endl;

}

```

The final function checks whether the implementations of `And()`, `Or()`, `XOr()` and `Not()` are mutually consistent. (It doesn't actually check whether they are right, just that they are consistent). It relies on relationships like: "A Or B" is the same as "Not (Not A And Not B)".

```

void Test3()
{
    int somebits[] = {
        2, 4, 6, 8, 16,
        33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55,
        68, 74, 80, 86,
        102, 112, 122, 132,
        145, 154, 415, 451
    };
    int n = sizeof(somebits) / sizeof(int);
    Bitmap b1;
    SetBits(b1, somebits, n);

    int otherbits[] = {
        2, 3, 6, 9, 16,
        23, 25, 27, 49, 52, 63, 75, 87, 99,
        102, 113, 132, 143,
        145, 241, 246, 362, 408, 422, 428, 429, 500, 508
    };
    int m = sizeof(otherbits) / sizeof(int);
    Bitmap b2;
    SetBits(b2, otherbits, m);

    Evaluate Or directly
    Bitmap b3 = b1.Or(b2);
}

```

```

    Bitmap b4 = b1.Not();
    Bitmap b5 = b2.Not();
    Bitmap b6 = b4.And(b5);
    b6.Invert();

    if(b6.Equals(b3)) cout << "The ands, ors and nots are "
                        "consistent" << endl;
    else cout << "back to the drawing board" << endl;

    b3 = b1.XOr(b2);
    b4 = b1.And(b2);
    b5 = b4.Not();
    b6 = b1.Or(b2);

    b6 = b6.And(b5);
    if(b3.Equals(b6)) cout << "XOr kd" << endl;
    else cout << "XOr what?" << endl;
}

```

*And by using a
sequence of And and
Not operations*

*Check consistency of
results*

*Don't forget to check
XOr*

A class isn't complete until you have written a testing program and checked it out. Class Bitmap passed these tests. It may not be perfect but it appears useable.

19.3 NUMBERS – A MORE COMPLEX FORM OF DATA

Bitmaps are all very well, but you don't use them that often. So, we need an example of a class that might get used more widely.

How about *integers*. You can add them, subtract them, multiply, divide, compare for equality, assign them, ... You may object "*C++ already have integers*". This is true, but not integers like

96531861715696714500613406575615576513487655109

or

-3344455556666667777777788888888999999

or

176890346568912029856350812764539706367893255789655634373681547
453896650877195434788809984225547868696887365466149987349874216
93505832735684378526543278906235723256434856

The integers we want to have represented by a class are LARGE integers; though just for our examples, we will stick to mediumish integers that can be represented in less than 100 decimal digits.

Why bother with such large numbers? Some people do need them. Casual users include: journalists working out the costs of pre-election political promises, economists estimating national debts, demographers predicting human population. The frequent users are the cryptographers.

There are simple approaches to encrypting messages, but most of these suffer from the problem that the encryption/decryption key has to be transmitted, before the message, by some separate secure route. If you have a secure route for transmitting the key, then why not use it for the message and forget about encryption?

Various more elaborate schemes have been devised that don't require a separate secure route for key transmission. Several of these schemes depend in strange ways on properties of large prime numbers, primes with 60...100 digits. So cryptographers often do get involved in performing arithmetic with large numbers.

Representing large integers

The largest integer that can be represented using hardware capabilities will be defined by the constant `INT_MAX` (in the file `limits.h`):

```
#define INT_MAX          2147483647
```

but that is tiny, just ten decimal digits. How might you represent something larger?

One possibility would be to use a string:

```
"123346753245098754645661"
```

Strings make some things easy (e.g. input and output) but it would be hard to do the arithmetic. The individual digits would have to be accessed and converted into numeric values in the range 0...9 before they could be processed. The index needed to get a digit would depend on the length of the string. It would all be somewhat inconvenient.

A better representation might be an array of numbers, each in the range 0...9, like that shown in Figure 19.2. The array dimension defines largest number that can be represented, so if we feel ambitious and want 1000 digit numbers then

```
fDigits[1000];
```

should make this possible.

Each digit would be an unsigned integer. As the range for the digits is only 0...9 they can be stored using "unsigned char". So, a large integer could be represented using an array of unsigned characters.

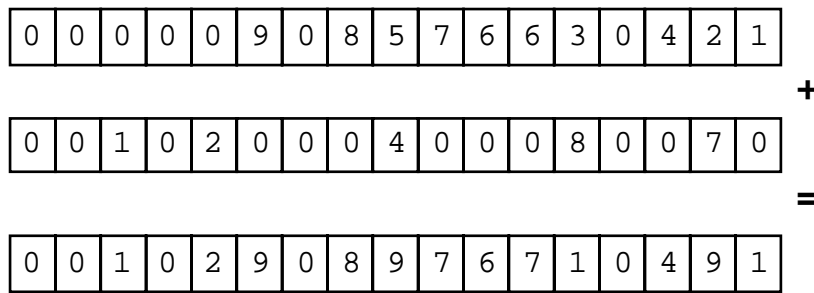
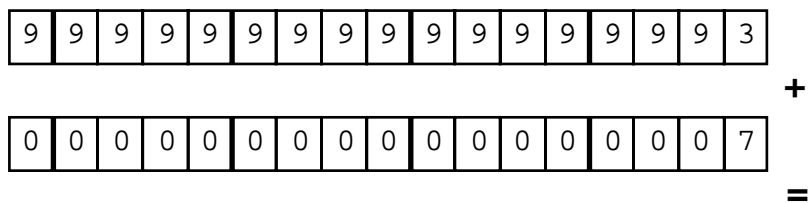


Figure 19.2 Large integers represented as arrays.

Arithmetic operations would involve loops. We could simply have the loops check all one hundred (or one thousand) digits. Generally the numbers are going to have fewer than the maximum number of digits. Performance will improve if the loops only process the necessary number of digits. Consequently, it will be useful if along with the array each "Number" object has details of its current number of digits.

Of course there is a problem with using a fixed size array. Any fixed sized number can *overflow*. Overflow was explained in Part I where binary hardware representations of numbers were discussed. Figure 19.3 illustrates the problem in the context of large integers. The implementation of arithmetic operations will need to include checks for overflow.



Overflow

Figure 19.3 "Overflow" may occur with any fixed size number representation.

There has to be a way of representing signed numbers. The smart way would be to use a representation known as "tens-complement". But that is getting really way out of our depth. So instead, we can use "sign and magnitude". The array of digits will represent the size (magnitude) of the number; there will be a separate data member that represents the sign.

Signed numbers

Sign and magnitude representations do have a problem. They complicate some arithmetic operations. For example, you have to check the signs of both numbers that

you are combining in order to determine whether you should be doing addition or subtraction of the magnitude parts:

+ve number	PLUS	+ve number	=>	Do addition
+ve number	PLUS	-ve number	=>	Do subtraction
-ve number	PLUS	+ve number	=>	Do subtraction
-ve number	PLUS	-ve number	=>	Do addition
+ve number	MINUS	+ve number	=>	Do subtraction
+ve number	MINUS	-ve number	=>	Do addition
...				

As well as performing the operation, you have to determine the sign of the result.

Doing the arithmetic

How does the arithmetic get done?

The algorithms that manipulate these large integers are going to be the same as those you learnt way back in primary school. We are going to have to have

addition with "carry" between columns,
 subtract with "borrow" between columns,
 long multiplication,
 long division.

Figure 19.4 illustrates the mechanism of addition with "carry". This can be encoded using a simple loop:

```

carry = 0;
for(int i=0; i < lim; i++) {
    int temp;
    temp = digit[i] of first number +
           digit[i] of second number +
           Carry;

    if(temp>=10) {
        Carry = 1; temp -= 10;
    }
    else Carry = 0;
    digit [i] of result = temp;
}

```

The limit for the loop will be determined by the number of digits in the larger magnitude value being added. The code has to check for overflow; this is easy, if Carry is non-zero at the end of the loop then overflow has occurred.

The process of subtraction with "borrow" is generally similar. Of course, you must subtract the smaller magnitude number from the larger (and reverse the sign if necessary).

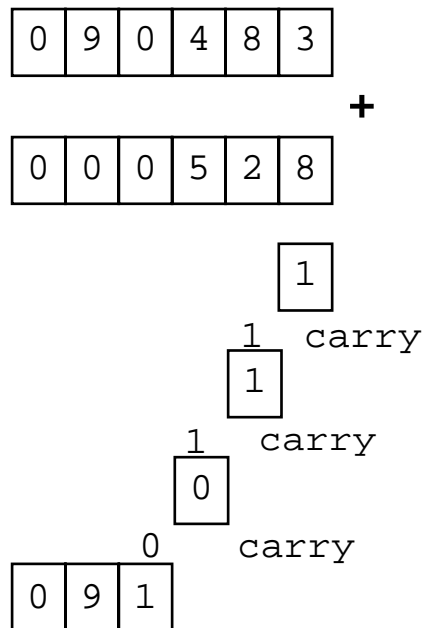


Figure 19.4 Mechanism of addition with carry.

One could do multiplication by repeated addition, but that is a bit slow. Instead a combination of "shifts", simple products, and addition steps should be used. The "shifts" handle multiplying by powers of ten, as shown in Figure 19.5. The process would have to be coded using several separate functions. An overall driver routine could step through the digits of one of the numbers, using a combination of a "times ten" and a "product" routine, and employing the functions written to handle addition.

```
for(int i=0;i<limit; i++)
  if(digit[i] != 0) {
    temp = other number * 10i
    use product function to multiply temp
      by digits[i]
    add product into result
  }
```

(Check the outline multiplication code; you should find that this is the algorithm that you use to do "long multiplication".)

The hardest operation is division. A form of "long division" has to be coded. This is actually quite a hard algorithm to code. But you don't have to. It is standard, so versions are available in subroutine libraries.

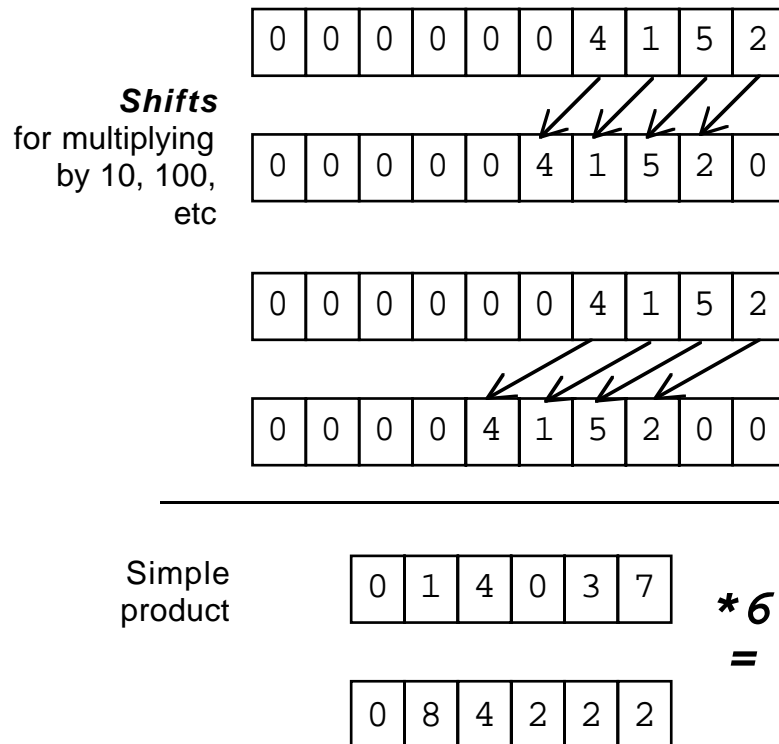


Figure 19.5 Shift and simple product sub-operations in multiplication.

P. Brinch Hansen recently published a version of the long division algorithm ("Software Practice and Experience", June 1994). His paper gives a model implementation (in Pascal) with the long division process broken down into a large number of separate routines that individually are fairly simple. (The implementation of the `Number` class includes a C++ version of Brinch Hansen's code. You should read the original paper if you need a detailed explanation of the algorithm.)

Specification

Implement a class that represents multi digit integer numbers. This "Numbers" class should support the basic arithmetic operations of addition, subtraction, multiplication, and division. It should be possible to transfer Numbers to/from disk, and to print readable representations of numbers. Comparison functions for numbers should be defined. It should be possible to initialize a Number from an ordinary long integer value, or from a given string of digits.

Design

First, consider the data. What data does an individual `Number` own? Each instance of class `Number` will have an array of unsigned long characters to store the digits. In addition, there has to be a "sign" data member (another unsigned character) and a length data member (a short integer). The size of the array of unsigned characters can be defined by a `#define` constant.

The data owned

```
unsigned char    fDigits[kMAXDIGITS+1]; // +1 explained later
unsigned char    fPosNeg;
short           fLength;
```

The next step is to decide what "services" a `Number` object should provide. As in the previous example, you make up a list of useful functions – things that a `Number` might be expected to do. The list is definitely not going to be the complete list of member functions. As already noted, functions like "Multiply" and "Divide" are complex and will involve auxiliary routines (like the "times 10" and "product" needed by "Multiply"). These extra auxiliary routines will be added to the list of member functions as they are identified during the design process. These extra functions will almost always be private; they will be simply an implementation detail.

*Services provided by
a Number*

When writing a test program to exercise a class, you may identify the need for additional member functions. Getting a class right does tend to be an iterative process.

The following functions form a reasonable starting point:

- Constructors.
Several of these. The simplest should just initialize a `Number` to zero. The specification required other constructors that could initialize a `Number` to a given long and to a given string.

Another constructor that would probably be useful would initialize a `Number` given an existing `Number` whose value is to be copied.
- Add, Subtract, Multiply, Divide.
These combine two numbers. It seems reasonable to make them functions that return a `Number` as a result. A `Number` isn't too large (around 100-110 bytes) so returning a `Number` in the stack is not too unreasonable. A function giving a `Number` as a result would also be convenient for programs using `Numbers`.
- Zero.
Probably useful, to have a function that tests whether a `Number` is zero, and another function that zeros out an existing `Number`.
- ReadFrom, WriteTo.
Binary transfer to files.

- **PrintOn.**
Output. Probably just print digit sequence with no other formatting. One hundred digit numbers just about fit on a line. If `Numbers` were significantly larger, it would be necessary to work out an effective way of printing them out over several lines.
- **Compare.**
Can return a -1, 0, or 1 result depending on whether a `Number` is less than, equal to, or bigger than the `Number` that it was asked to compare itself with.
- **ChangeSign.**
Probably useful.

The functions that combine `Numbers` to get a `Number` as a result will take the second `Number` as a reference argument.

Prototypes We can jump immediately to function prototypes and complete a partial declaration of the class:

```
#define kMAXDIGITS = 100;

class Number {
public:
    Number();
    Number(long);
    Number(char numstr[]);
    ...

    void    ReadFrom(istream& in);
    void    WriteTo(ofstream& out) const;
    void    PrintOn(ostream& printer) const;

    Number Add(const Number& other) const;
    Number Subtract(const Number& other) const;
    Number Multiply(const Number& other) const;
    Number Divide(const Number& other) const;
    int     Zero_p() const;
    void    MakeZero();

    int     Equal(const Number& other) const;
    void    ChangeSign();
    int     Compare(const Number& other) const;
private:
    // private functions still to be defined
    ...
    ...

    unsigned char fDigits[kMAXDIGITS+1];
    unsigned char fPosNeg; // 0 => positive, 1 => negative
    short        fLength;
};
```

It is worthwhile coming up with this kind of partial class outline fairly early in the design process, because the outline can then provide a context in which other design aspects can be considered. (The extra digit in the `fDigits` data member simplifies one part of the division routines.)

There aren't many general design issues remaining in this class. One that should be considered is error handling. There are several errors that we know may occur. We can ignore i/o errors, they can be checked by the calling environment that transfers `Numbers` to and from disk files. But we have to deal with things like overflow. Another problem is dealing with initialization of a `Number` from a string. What should we do if we are told to initialize a `Number` with the string "Hello World" (or, more plausibly, "1203o5l7" where there are letters in the digit string)?

Error handling

Chapter 26 introduces the C++ exception mechanism. Exceptions provide a means whereby you can throw error reports back to the calling code. If you can't see a general way of dealing with an error when implementing code of your class, you arrange to pass responsibility back to the caller (giving the caller sufficient information so that they know what kind of error occurred). If the caller doesn't know what to do, the program can terminate. But often, the caller will be able to take action that clears the error condition.

For now, we use a simpler mechanism in which an error report is printed and the program terminates.

When defining a class, you always need to think about "assignment":

Assignment of Numbers

```
Number a;  
Number b;  
...  
b = a;
```

Just as in the case of `Bitmaps`, assignment of `Numbers` presents no problems. A `Number` is simply a block of bytes; the compiler will generate "memory copy" instructions to implement the assignment. Later examples will illustrate cases where you may wish to prohibit assignment operations or you may need to change the compiler's default implementation.

Detailed design and implementation

Some of the member functions of class `Number` are simple, and for these it is possible to sketch pseudo-code outlines, or even jump straight to implementation. These member functions include the default constructor:

```
Number::Number()  
{  
    fPosNeg = 0;  
    for(int i=0; i<= kMAXDIGITS; i++) fDigits[i] = 0;  
    fLength = 0;
```

```
}
```

The next two constructors are slightly more elaborate:

```
Number::Number(long lval)
{
    fPosNeg = 0;
    if(lval<0) { fPosNeg = 1; lval = -lval; }
    for(int i=0; i<= kMAXDIGITS; i++) fDigits[i] = 0;
    fLength = 0;
    while(lval) {
        fDigits[fLength++] = lval % 10;
        lval = lval / 10;
    }
}

Number::Number(char numstr[])
{
    for(int i=0; i<= kMAXDIGITS; i++) fDigits[i] = 0;
    fPosNeg = 0; /* Positive */
    i = 0;
    fLength = strlen(numstr);

    if(numstr[i] == '+') { i++; fLength--; }
    else
    if(numstr[i] == '-') { fPosNeg = 1; i++; fLength--; }

    int    pos = fLength - 1;

    while(numstr[i] != '\0') {
        if(!isdigit(numstr[i])) {
            cout << "Bad data in number input\n";
            exit(1);
        }
        fDigits[pos] = numstr[i] - '0';
        i++;
        pos--;
    }

    while((fLength>0) && (fDigits[fLength-1] == 0)) fLength--;
}
```

Check for ± sign at start of string

Loop to process all characters in string

Fix up any case where given leading zeros

You shouldn't have any difficulty in understanding the constructor that initializes a Number from a long integer value. The code just has a loop that fills in digits starting with the units, then the tens and so forth.

The constructor that takes a character string first checks for a + or - sign. Then it loops processes all remaining characters in the string. If any are non-digits, the program terminates with an error message. Digit characters are converted into numeric values in range 0...9 and placed in the correct locations of the array. (Note how the

length of the string is used to determine the number of digits in the number, and hence the digit position in the array to be filled in with the first digit taken from the string.) The final loop just corrects for any cases where there were leading zeros in the string.

Very often you need to say something like "Give me a Number just like this one". *Copy constructor* This is achieved using a copy constructor. A copy constructor takes an existing class instance as a reference argument and copies the data fields:

```
Number::Number(const Number& other)
{
    fPosNeg = other.fPosNeg;
    fLength = other.fLength;
    for(int i=0; i<= kMAXDIGITS; i++)
        fDigits[i] = other.fDigits[i];
}
```

Other simple functions include the comparison function and the input/output functions:

```
int Number::Compare(const Number& other) const
{
    if(fPosNeg != other.fPosNeg) {
        /* The numbers have opposite signs.
        If this is positive, then return "Greater" else
        return "Less". */
        return (fPosNeg == 0) ? 1 : -1;
    }

    if(fLength > other.fLength)
        return (fPosNeg == 0) ? 1 : -1;

    if(fLength < other.fLength)
        return (fPosNeg == 0) ? -1 : 1;

    for(int i = fLength-1; i>=0; i--)
        if(fDigits[i] > other.fDigits[i])
            return (fPosNeg == 0) ? 1 : -1;
        else
            if(fDigits[i] < other.fDigits[i])
                return (fPosNeg == 0) ? -1 : 1;

    return 0;
}

void Number::WriteTo(ofstream& out) const
{
    // Remember, some iostreams+compilers may require (char*)
    out.write(&fDigits, sizeof(fDigits));
    out.write(&fPosNeg, sizeof(fPosNeg));
    out.write((char*)&fLength, sizeof(fLength)); // like this
}
```

Can tell result from signs if these not the same

Otherwise by number of digits if they differ

But sometimes have to compare digit by digit from top


```

void Number::PrintOn(ostream& printer) const
{
    if(fLength==0) { printer << "0"; return; }

    if(fPosNeg) printer << "-";
    int i = kMAXDIGITS;
    i = fLength - 1;

    while(i>=0) {
        printer << char('0' + fDigits[i]);
        i--;
    }
}

```

The WriteTo() function uses a separate call to write() for each data member. (the ReadFrom() function uses equivalent calls to read()). Sometimes this gets a bit clumsy; it may be worth inventing a struct that simply groups all the data members so as to facilitate disk transfers.

A few of the member functions are sufficiently simple that they can be included as inlines:

```

inline int Number::Zero_p() const
{
    return (fLength == 0);
}

inline void Number::ChangeSign()
{
    if(fPosNeg) fPosNeg = 0;
    else fPosNeg = 1;
}

```

As noted previously, such functions have to be defined in the header file with the class declaration.

Designing little programs!

The detailed design of the harder functions, like Subtract(), Multiply() and Divide(), really becomes an exercise in "top down functional decomposition" as illustrated by the many examples in Part III. You have the same situation. You have a "program" to write (e.g. the "Multiply program") and the data used by this program are simple (the data members of two class instances).

The "Add Program" and the "Subtract Program"

The addition and subtraction functions are to be called in code like:

```

Number a("1239871154378100173165461515");
Number b("71757656755466753443546541431765765137654");
Number c;
Number d;
c = a.Add(b);
d = a.Subtract(b);

```

The `Add()` (and `Subtract()`) functions are to return, via the stack, a value that represents the sum (difference) of the Numbers `a` and `b`. As noted in the introduction, sign and magnitude representations make things a bit more complex. You require lower level "do add" and "do subtract" functions that work using just the magnitude data. The actual `Add()` function will use the lower level "do add" if it is called to combine two values with the same sign, but if the signs are different the "do subtract" function must be called. Similarly, the `Subtract()` function will combine the magnitudes of the two numbers using the lower level "do add" and "do subtract" functions; again, the function used to combine the magnitude values depends upon the signs of the values.

Sign and magnitude representation complicates the process

There is a further complication. The actual subtraction mechanism (using "borrows" etc) only works if you subtract the smaller magnitude number from the larger magnitude number. The calls must be set up to get this correct, with the sign of the result being changed if necessary.

The following are sketches for the top level routines:

```
Add
// Returns sum of "this" and other
  initialize result to value of "this"

  if(other is zero)
    return result;

  if("this" and other have same sign) {
    do addition operation combining result with other;
    return result;
  }
  else
    return DoSubtract(other);
```

The code starts by initializing the result to the value of "this" (so for `a.Add(b)`, the result is initialized to the value of `a`). If the other argument is zero, it is possible to return the result. Otherwise we must chose between using a "do add" function to combine the partial result with the other argument, or calling `DoSubtract()` which will sort out subtractions.

The code for the top-level `Subtract()` function is similar, the conditions for using the lower-level addition and subtraction functions are switched:

```
Subtract
// Returns difference of "this" and other
  initialize result to value of "this"

  if(other is zero)
    return result;

  if("this" and other have same sign)
    return DoSubtract(other);
  else {
```

```

        do addition operation combining result with other
        return result;
    }
}

```

***New member
functions identified***

These sketches identify at least one new "private member function" that should be added to the class. It will be useful to have:

```
int SameSign(const Number& other) const;
```

This function checks the `fPosNeg` data members of an object and the second `Number` (`other`) passed as an argument (i.e. `return (this->fPosNeg == other.fPosNeg);`). This can again be an inline function.

We also need to initialize a variable to the same value as "this" (the object executing the function). This can actually be done by an assignment; but the syntax of that kind of assignment involves a deeper knowledge of pointers and so is deferred until after the next chapter. Instead we can use a small `CopyTo()` function:

```
void Number::CopyTo(Number& dest) const
{
    // This function is not needed.
    // Can simply have dest = *this; at point of call.
    // Function CopyTo introduced to delay discussion of *this
    // until pointers covered more fully.
    dest.fLength = this->fLength;
    dest.fPosNeg = this->fPosNeg;
    for(int i=0; i<= kMAXDIGITS; i++)
        dest.fDigits[i] = this->fDigits[i];
}

```

Both `Add()` and `Subtract()` invoked the auxiliary function `DoSubtract()`. This function has to sort out the call to the actual subtraction routine so that the smaller magnitude number is subtracted from the larger (and fix up the sign of the result):

```
DoSubtract
// Set up call to subtract smaller from larger magnitude

if(this is larger in magnitude than other)
    Initialize result to "this"
    call actual subtraction routine to subtract other
    from result
else
    Initialize a temporary with value of this
    set result to value of other;

    call actual subtraction routine to subtract temp.
    from result
    if(subtracting) change sign of result

```

```
return result;
```

Yet another auxiliary function shows up. We have to compare the magnitude of numbers. The existing `Compare()` routine considers sign and magnitude but here we just want to know which has the larger digit string. The extra function, `LargerThan()`, becomes an additional private member function. Its code will be somewhat similar to that of the `Compare()` function involving checks on the lengths of the numbers, and they have the same number of digits then a loop checking the digits starting with the most significant digit.

An outline for the low level "do add" routine was given earlier when the "carry" mechanism was explained (see Figure 19.4). The lowest level subtraction routine has a rather similar structure with a loop that uses "borrows" between units and tens, tens and hundreds, etc.

With the auxiliary functions needed for addition and subtraction the class declaration becomes:

*Class handling
additions and
subtractions*

```
class Number {
public:
    Number();
    ...
    // Public Interface unchanged
    ...
    int    Compare(const Number& other) const;
private:
    int    SameSign(const Number& other) const;
    void    DoAdd(const Number& other);
    Number DoSubtract(const Number& other, int subop) const;
    void    SubtractSub(const Number& other);

    int    LargerThan(const Number& other) const;
    void    CopyTo(Number& dest) const;

    unsigned char fDigits[kMAXDIGITS+1];
    unsigned char fPosNeg;
    short        fLength;
};
```

The implementation for representative functions is as follows:

```
Number Number::Subtract(const Number& other) const
{
    Number result;
    CopyTo(result);

    if(other.Zero_p()) return result;

    if(this->SameSign(other))
        return DoSubtract(other, 1);
```

```

        else {
            result.DoAdd(other);
            return result;
        }
    }

```

The Add() routine is very similar. It should be possible to code it, and the auxiliary DoSubtract() from the outlines given.

Both the DoAdd() and the SubtractSub() routines work by modifying a Number, combining its digits with the digits of another number. The implementation of DoAdd(), "add with carry", is as follows:

```

void Number::DoAdd(const Number& other)
{
    int    lim;
    int    Carry = 0;
    lim = (fLength >= other.fLength) ? fLength : other.fLength;
    for(int i=0; i<lim; i++) {
        int temp;
        temp = fDigits[i] + other.fDigits[i] + Carry;
        if(temp>=10) { Carry = 1; temp -= 10; }
        else Carry = 0;
        fDigits[i] = temp;
    }
    fLength = lim;
    if(Carry) {
        if(fLength == kMAXDIGITS) Overflow();
        fDigits[fLength] = 1;
        fLength++;
    }
}

```

Loops limited by magnitude of larger number

Deal with any final carry, check for Overflow

Handling overflow

A routine has to exist to deal with overflow (indicated by carry out of the end of the number). This would not need to be a member function. The implementation used a static filescope function defined in the same file as the Numbers code; the function printed a warning message and terminated.

The function SubtractSub() implements the "subtract with borrow" algorithm:

```

void Number::SubtractSub(const Number& other)
{
    int    Borrow = 0;
    int    newlen = 0;
    for(int i=0; i<fLength; i++) {
        int temp;
        temp = fDigits[i] - other.fDigits[i] - Borrow;
        if(temp < 0) { temp += 10; Borrow = 1; }
        else Borrow = 0;
        fDigits[i] = temp;
        if(temp) newlen = i+1;
    }
}

```

```

    }
    fLength = newlen;
    if(fLength==0) fPosNeg = 0;
}

```

(If the result is ± 0 , it is made +0. It is confusing to have positive and negative version of zero.)

In a phased implementation of the class, this version with just addition and subtraction would be tested.

The usage of the multiplication routine is similar to that of the addition and subtraction routines: *The "Multiply Program"*

```

Number a("1239871154378100173165461515");
Number b("71757656755466753443546541431765765137654");
Number c;
c = a.Multiply(b);

```

Function `Multiply()` again returns a `Number` on the stack that represents the product of the `Number` executing the routine ('a') and a second `Number` ('b').

Sign and magnitude representation doesn't result in any problems. Getting the sign of the product correct is easy; if the two values combined have the same sign the product is positive otherwise it is negative. The basics of the algorithm was outlined in the discussion related to Figure 19.5. The code implementing the algorithm is:

```

Number Number::Multiply(const Number& other) const
{
    Number Result; // Number defaults to zero

    if(other.Zero_p()) { return Result; }

    for(int i=0;i<fLength; i++)
        if(fDigits[i]) {
            Number temp(other);

            temp.Times10(i);
            temp.Product(fDigits[i]);
            Result = Result.Add(temp);
        }

    if(SameSign(other))
        Result.fPosNeg = 0;
    else Result.fPosNeg = 1;

    return Result;
}

```

Note use of "Copy constructor"

The routine forms a product like:

19704 * 6381

by calculating

4	*	6381	25524	(4*6381*10 ⁰)
0	*	63810	0	(0*6381*10 ¹)
7	*	638100	4466700	(7*6381*10 ²)
9	*	6381000	57429000	(9*6381*10 ³)
1	*	63810000	63810000	(1*6381*10 ⁴)
Total =			125731224	

**Further private
member functions**

Two more auxiliary functions are needed. One, `Times10()`, implements the shifts to multiply by a specified power of ten. The other, `Product()`, multiplies a `Number` by a value in the range 0...9. Both change the `Number` for which they are invoked. Partial results from the individual steps accumulate in `Result`.

```
void Number::Times10(int power)
{
    if((fLength+power)>kMAXDIGITS) Overflow();
    for(int i = fLength-1;i>=0;i--) fDigits[i+power] = fDigits[i];
    for(i = power-1;i>=0;i--) fDigits[i] = 0;
    fLength += power;
}
```

Function `Times10()` needs to check that it isn't going to cause overflow by shifting a value too far over. If the operation is valid, the digits are just moved leftwards and the number filled in at the right with zeros.

The `Product()` routine is a bit like "add with carry":

```
void Number::Product(int k)
{
    int lim;
    int Carry = 0;
    if(k==0) {
        MakeZero();
        return;
    }

    lim = fLength;
    for(int i=0;i<lim; i++) {
        int temp;
        temp = fDigits[i]*k + Carry;
        Carry = temp / 10;
        temp = temp % 10;
        fDigits[i] = temp;
    }
    if(Carry) {
        if(fLength == kMAXDIGITS) Overflow();
        fDigits[fLength] = Carry;
    }
}
```

```

        fLength++;
    }
}

```

You must remember this:

*The "Divide"
Program*

```

      0000029-----
49853 |1467889023451770986543175
      99706
      470829
      448677

```

(unless your primary school teachers let you use a calculator). You found long division hard then; it still is.

As with multiplication, getting the sign right is easy. If the dividend and divisor have the same sign the result is positive, otherwise it is negative. The `Divide()` routine itself can deal with the sign, and also with some special cases. If the divisor is zero, you are going to get overflow. If the divisor is larger than the dividend the result is zero. A divisor that is a one digit number is another special case handled by an auxiliary routine. The `Divide()` function is:

```

Number Number::Divide(const Number& other) const
{
    if(other.Zero_p()) Overflow();
    Number Result; // Initialized to zero

    if(!LargerThan(other))
        return Result; // return zero

    CopyTo(Result);

    if(other.fLength == 1)
        Result.Quotient(other.fDigits[0]);
    else
        Result.LongDiv(other);

    if(other.fPosNeg == fPosNeg) Result.fPosNeg = 0;
    else Result.fPosNeg = 1;

    return Result;
}

```

The auxiliary (private member) function `Quotient()` deals with simple divisions (e.g. $77982451 \div 3 = 2599\dots$). You start with the most significant digit, do the division to get the digit, transfer any remainder to the next less significant digit:

```

void Number::Quotient(int k)
{

```



```

        int    Carry = 0;
        int newlen = 0;
        for(int i= fLength-1;i>=0;i--) {
            int    temp;
            temp = 10*Carry + fDigits[i];
            fDigits[i] = temp / k;
            if((newlen==0) && (fDigits[i] !=0)) newlen = i+1;
            Carry = temp % k;
        }
        fLength = newlen;
    }

```

The main routine is `LongDiv()`. Brinch Hansen's algorithm is moderately complex and involves several auxiliary functions that once again become private member functions of the class. The code is given here without the explanation and analysis (available in Brinch Hansen's paper in *Software Practice and Experience*):

*Using lengths of
numbers to get idea
of size of quotient*

*loop filling in
successive digits of
quotient*

*Taking a guess at
next digit of quotient*

```

void Number::LongDiv(const Number& other)
{
    int    f;
    Number d(other);
    Number r;
    CopyTo(r);
    int    m = other.fLength;
    int    n = fLength;

    f = 10 / (other.fDigits[m-1] + 1);
    r.Product(f);
    d.Product(f);

    int newlen = 0;
    for(int k = n - m; k>=0; k--) {
        int    qt;
        qt = r.Trial(d,k,m);
        if(qt==0) {
            fDigits[k] = 0;
            continue;
        }

        Number dq(d);
        dq.Product(qt);
        if(r.Smaller(dq,k,m)) { qt--; dq = dq.Subtract(d); }
        if((newlen==0) && (qt !=0)) newlen = k+1;
        fDigits[k] = qt;
        r.Difference(dq,k,m);
    }
    fLength = newlen;
}

int Number::Trial(const Number& other, int k, int m)
{

```

```

        int    km = k + m;
        int    r3;
        int d2;
        km = k + m;
        r3 = (fDigits[km]*10 + fDigits[km-1])*10 + fDigits[km-2];
        d2 = other.fDigits[m-1]*10 + other.fDigits[m-2];
        int temp = r3 / d2;
        return (temp<9) ? temp : 9;
    }

int Number::Smaller(const Number& other, int k ,int m)
{
    int    i;
    i = m;
    for(;i>0;i--)
        if(fDigits[i+k] != other.fDigits[i]) break;
    return fDigits[i+k] < other.fDigits[i];
}

void Number::Difference(const Number& other, int k, int m)
{
    int    borrow = 0;
    for(int i = 0; i <= m; i++) {
        int diff = fDigits[i+k] -
            other.fDigits[i] - borrow + 10;
        fDigits[i+k] = diff % 10;
        borrow = 1 - (diff / 10);
    }
    if(borrow) Overflow();
}

```

With all these auxiliary private member functions, the final class declaration ***Final class declaration*** becomes:

```

class Number {
public:
    Number();
    Number(long);
    Number(const Number& other);
    Number(char numstr[]);
    Number(istream&);

    void    ReadFrom(istream& in);
    void    WriteTo(ofstream& out) const;
    void    PrintOn(ostream& printer) const;

    Number Add(const Number& other) const;
    Number Subtract(const Number& other) const;
    Number Multiply(const Number& other) const;
    Number Divide(const Number& other) const;
    int    Zero_p() const;
}

```

```

        void    MakeZero();
        int     Equal(const Number& other) const;
        void    ChangeSign();
        int     Compare(const Number& other) const;
private:
        int     SameSign(const Number& other) const;
        void    DoAdd(const Number& other);
        Number  DoSubtract(const Number& other, int subop) const;
        void    SubtractSub(const Number& other);

        void    Product(int k);
        void    Quotient(int k);

        void    Times10(int power);
        int     LargerThan(const Number& other) const;

        void    LongDiv(const Number& other);
        int     Trial(const Number& other, int, int);
        int     Smaller(const Number& other, int, int);
        void    Difference(const Number& other, int, int);

        void    CopyTo(Number& dest) const;

        unsigned char fDigits[kMAXDIGITS+1];
        unsigned char fPosNeg;
        short         fLength;
};

```

Testing

The class has to be tested. Simple programs like:

```

int main()
{
    Number a("123456789");
    Number b("-987654");
    Number c;
    c = a.Add(b); cout << "Sum "; c.PrintOn(cout); cout <<
endl;
    c = a.Subtract(b); cout << "Difference ";
                        c.PrintOn(cout); cout << endl;
    c = a.Multiply(b); cout << "Product ";
                        c.PrintOn(cout); cout << endl;
    c = a.Divide(b); cout << "Quotient ";
                        c.PrintOn(cout); cout << endl;
    return 0;
}

```

check the basics. As long the values are less than ten digits you can verify them on a calculator or with a spreadsheet program (or even by hand).

You would then proceed to programs such as one calculating the expressions $(a-b)*(c+d)$ and $(ac-bc+ad-bd)$ for different randomly chosen values for the Numbers `a`, `b`, `c`, and `d` (the two expressions should have the same value). A similar test routine for the multiplication and divide operations would check that $(a*b*c*d)/(a*c)$ did equal $(b*d)$; again the expressions would be evaluated for hundreds of randomly selected values. (A small random value can be obtained for a Number as follows: `long lv; lv = rand() % 25000; lv -= 12500; Number a(lv); ...`. Alternatively, an additional "Randomize()" function can be added to the class that will fill in Numbers with randomly chosen 40 digit sequences and a random sign. You limit the size of these random values so that you could multiply them without getting overflows). Such test programs need only generate outputs if they encounter cases where the pair of supposedly equal calculated values are not in fact equal.

Automatic test programs

If the class were intended for serious use in an application, you would then use a profiling tool like that described in Chapter 14. This would identify where a test program spent most time. The test program has to use Numbers in the same manner as the intended application. If the application uses a lot of division so should the test program.

Optimising the code

Such an analysis was done on the code and the `Product()` routine showed up as using a significant percentage of the run time. If you look back at the code of this routine you will see that it treats multiplication by zero as a special case (just zero out the result and return); other cases involve a loop that multiplies each digit in the Number, getting the carry etc. Multiplying by 1 could also be made a special case. If the multiplier is 1, the Number is unchanged. This special case would be easy to incorporate by an extra test with a return just before the main loop in the `Product()` routine. Retesting showed that this small change reduced the time per call to `Product()` from 0.054 to 0.049 μ s (i.e. worthwhile). Similar analysis of other member functions could produce a number of other speed ups; however, generally, such improvements would be minor, in the order of 5% to 10% at most.

19.4 A GLANCE AT THE "IOSTREAM" CLASSES

Now that you have seen example class declarations and functions, and seen objects (class instances) being asked to perform functions, the syntax of some of the input and output statements should be becoming a little clearer.

The `iostream` header file contains many class declarations, including:

```
class ostream ... (some detail omitted here) ... {
public:
    ...
    ostream &flush();
    // Flush any characters queued for output.
```

```

        ostream &put(char c);
        // Inserts a character

        ostream &write(const void *data, size_t size);
        // writes data, your version of ostream may use
        // different argument types
        ...
};

```

Functions like `put()`, and `write()` are public member functions of the class. So, if you have an ostream object `out`, you can ask that object to perform a write operation using the expression:

```
out.write(&dataval, sizeof(dataval));
```

In Chapter 23, the strange "takes from" `<<` and "gives to" `>>` operators will get explained.

EXERCISES

- 1 Implement class `Numbers`, filling in all the simple omitted routines and check that these `Numbers` do compute.
- 2 Add an "integer square root" member function that will use Newton's method to find the integer `x` such that `x` is the largest integer for which $x^2 \leq N$ for a given `Number N`.
- 3 Often, users of these large numbers want the remainder and not the quotient of a division operation. The `LongDiv()` routine actually computes but discards the remainder. Add an extra member function that will calculate the remainder. It should be based on the existing `Divide()` and `LongDiv()` functions.
- 4 Write a program that tests whether a given `Number` is prime. (The program will be somewhat slow, probably not worth testing any numbers bigger than 15 digits.)

20 Dynamic data and pointers

In all the examples considered so far, we have known how many data elements there will be, or we have at least decided on some maximum number that we are prepared to deal with. The data may be global, their space can be sorted out at compile-time (or link-load time); or the data may be temporary such as variables that belong to a function and which exist on the stack only while the code of that function is being executed. But a lot of problems involve data that aren't like that. You know that you have to deal with data objects, but you really don't know how many there will be, and nor do you know for how long you will need them.

The main example in this chapter is an "Air Traffic Control" game. The game involves the player controlling the movements of aircraft, giving them orders so that they line up with a runway, moving at an appropriate speed and rate of descent so that they can land safely. The game simulates the passage of time and introduces new aircraft into the controlled air space at a predefined frequency. The "lifetime" of an individual aircraft varies depending on the skill of the player. Aircraft get created by the game component; they get removed on successful landing (or by running out of fuel). If the player is skilful, the correct sequence orders is given to adjust the aircraft's speed, height and bearing through a number of stages until it is aligned with the runway. If the player makes a mistake, an aircraft may have to overshoot and circle around for a second try (or may run out of fuel). So the lifetime of aircraft does vary in essentially arbitrary ways. The number of aircraft in existence at a particular moment will also vary. At simple game levels there will be one or two (plus any making second attempts after aborted landings); at higher game levels there could be dozens.

The game program would represent the aircraft using data structures (either structs or instances of some class Aircraft). But these data structures have to be handled in ways that are quite different from any data considered previously.

Section 20.1 introduces the "heap" and the operators that can be used to create and destroy data objects in the heap. The "heap" is essentially an area of memory set aside

Objects with variable lifetimes

The "heap"

for a program to use to create and destroy data structures that have varied lifetimes like the example aircraft.

*Addresses and
"Pointers"*

The actual allocation of space in the heap for a new data object is handled by a run-time support routine. This "memory manager" routine will select some space in the heap when asked to create an object of a given size. The memory manager reports where the object has been placed by returning its address. This address becomes the value held in a "pointer variable". The heap-based object has to be accessed "indirectly via the pointer". Section 20.2 looks at issues like the definition of pointer variables, the use of pointers to access data members of an object, and operations on entire structures that are accessed through pointers.

*"Air Traffic
Controller"*

A simple Air Traffic Control game is used in Section 20.3 as a practical illustration. This version doesn't have a particularly attractive user interface; it is just a framework with which to illustrate creation, access, and destruction of objects.

*The "address of"
operator*

You can work with pointers to data elements other than those allocated on the heap. We have already had cases where the '&' address of operator was used to get a pointer value (when passing an address to the `read()` and `write()` low level i/o functions). Section 20.4 looks at the use of the `&` operator and a number of related issues. Many of the libraries that you use are still C language libraries and so you must learn something of C's styles of pointer usage.

Pointers and arrays

Another legacy of C is the rather poorly defined concept of an array. C programming idioms often abandon the concept of an array as a composite structure of many elements identifiable and accessible using their index value. Instead the C hacker style uses the assembly language level notions of a block of memory that can be accessed via address registers that have been loaded with the addresses of byte locations in memory. In this style, the contents of arrays are accessed using pointers and arithmetic operations are performed on pointers to change their values so as to identify different array elements. Because of the large amount of existing C code that you will need to work with, you need a "reading knowledge" of some of these pointer idioms. This is the content of Section 20.5. Note, it should be a "reading knowledge"; you should not write code employing these coding idioms.

Section 20.6 discusses "networks", showing how complicated structures can be built out of separate parts linked together with pointers.

20.1 THE "HEAP"

As illustrated in Figure 20.1, the memory allocated to a program is divided into four parts or "segments". The segments are: "code", "static data", "stack", and "heap". (This organization is conceptual only. The actual realization on a given computer architecture may well be considerably more complex.)

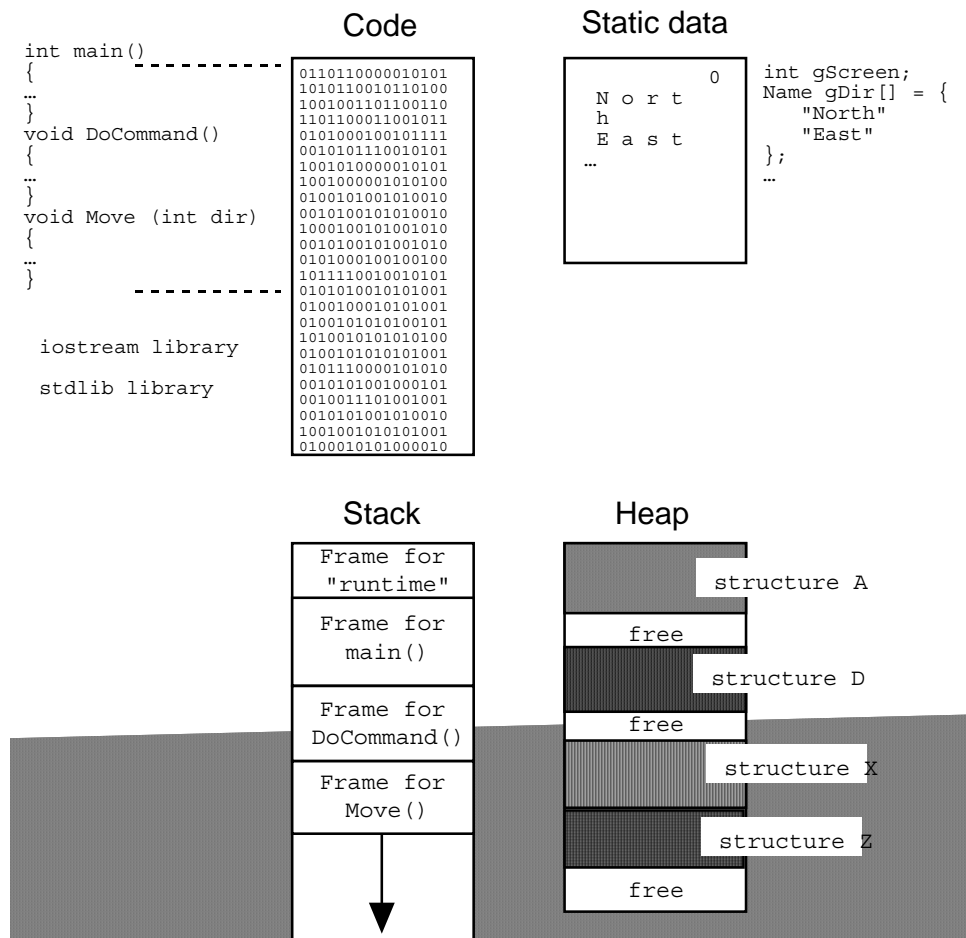


Figure 20.1 Program "segments" in memory: code, static data, stack, and "heap".

The "code" segment contains the bit patterns for the instructions. The contents of the code segment are composed largely by the compiler; the linking loader finalises some of the addresses needed in instructions like function calls. The code segment will include library routines that have been linked with the code specifically written for the program. On machines with "memory protection" hardware, the code segment will be effectively "read only".

Code segment

The "static data segment" is used for those data variables that are global or at least filescope. These are the variables that are defined outside of the body of a function. (In addition, there may be some variables that have been defined within functions, and which have function scope, but which have been explicitly allocated to the static data

"Static data segment"

segment. Such variables are rarely used; there are none in this text book.) Space for variables in the static data segment is allocated by the linking loader. The variables are initialized by a run-time support routine prior to entry to `main()`. In some cases, special "at exit" functions may manipulate these variables after a return from `main()` (i.e. after the program is nominally finished!). Such variables remain in existence for the entire duration of the program (i.e. their lifetime exceeds that of the program).

Stack The stack holds stack frames. The stack frames hold the local variables of a function together with the housekeeping details needed to record the function call and return sequence. A stack frame is created as a function is called and is freed on exit. Local variables of a function remain in existence during the execution of their own function and all functions that it calls.

The "heap" The heap is a completely separate region of memory controlled by a run-time "memory manager" support routine. (This is not the operating system's memory manager that sorts out what space should be given to different programs. This run-time memory manager is a library function linked to your code.) This run-time memory manager handles requests for fixed sized blocks of memory.

"new" operator In C it is normal to make requests direct to the memory manager specifying the size of blocks in terms of the number of bytes required. C++ has provided a higher level interface through the `new` operator explained below. Using `new`, a function can request the creation of a heap-based struct, a class instance, an array of variables of simple types (e.g. an array of characters) or even an array of class instances. The `new` operator works out the number of bytes needed and deals with all the other low level details associated with a call to the actual run-time memory manager.

When a program starts, the operating system gives it some amount of memory for its heap segment. The amount obviously varies with the system, but typical initial allocations would be in the range from a quarter megabyte to eight megabytes. On some systems, a program may start with a small area for its heap but is able to request that the OS enlarge the heap later. One of the start up routines would record details of the heap allocation and mark it all as "free".

Allocating space for data structures When asked for a block of memory, the memory manager will search the heap looking for a "free" area that is large enough provide the space required and hold a little additional housekeeping information. The memory manager needs to keep track of the space it allocates. As illustrated in Figure 20.2, its "housekeeping records" are placed as "headers" and "trailers" in the bytes just before and just after the space reserved for a new structure. These housekeeping records note the size of the block and mark it as "in use". When the memory manager function has finished choosing the space to be allocated and has filled in its records, it returns the address of the start of the data area.

Freeing unneeded structures Structures allocated on the heap eventually get discarded by the program (as when, in the example, the aircraft land or crash). Even if your programs starts with an eight megabyte heap, you will eventually run out of memory if you simply created, used, and then discarded the structures.

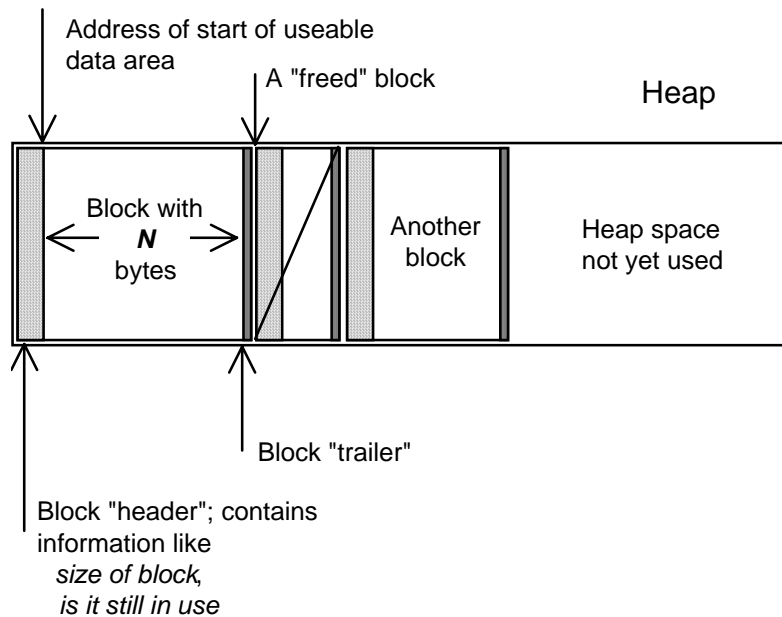


Figure 20.2 "Blocks" allocated on the heap.

If you create dynamic structures in the heap, you are expected to give them back to the memory manager when you no longer need them. The memory manager can then reclaim the space they occupied. The memory manager will mark their headers as "free". Subsequently, these blocks may get reallocated or merged with neighboring free blocks. In C++, you pass discarded data structures back to the memory manager using the `delete` operator.

delete operator

It is common for programmers to be a bit careless about giving discarded data structures back to the memory manager. Some dynamically allocated structures just get forgotten. Such structures become "dead space" in the heap. Although they aren't used they still occupy space. A part of the program's code that creates, uses, but fails to delete structures introduces a "memory leak". If that code gets called many times, the available heap space steadily declines ("my memory appears to be leaking away").

"Memory leaks"

If your code has a memory leak, or if you are allocating exceptionally large structures you may eventually run out of heap space. The C++ language has defined how the memory manager should handle situations where it is asked to create a new structure and it finds that there is insufficient space. These language features are a bit advanced; you will get to them in later studies. Initially, you can assume that requests for heap space will always succeed (if a request does fail, your program will be terminated and an error message will be printed).

Failure to allocate memory

Overheads when allocating heap structures

The headers and trailers added by the memory manager would typically come to about 16 bytes, maybe more. This is a "space overhead" associated with every structure allocated in the heap. A program that tries to allocate individual char or int variables in the heap almost certainly has a fundamental design error. The heap is meant to be used for creation of reasonable sized data objects like structs or arrays.

The work that the memory manager must perform to handle calls via `new` and `delete` is non-trivial. Quite commonly, profiling a program will reveal that a measurable percentage of its time is spent in the memory management routines. You should avoid creating and destroying structures inside deeply nested loops. The heap is meant to be used for the creation of data objects that have reasonable lifetimes.

Using the new operator

The following illustrate use of the `new` operator to create structures in the heap:

```
struct Point3d { double fX, fY, fZ, fR, fTheta, fPhi; }
class Bitmap; // as declared in Chapter 19
class Number; // as declared in Chapter 19
...

... = new Point3d;
...
... = new Bitmap;
...
... = new Number("77777666555");
...
... = new char[kMAX];
```

Each of these uses of the `new` operator results in the return of the address of the start of a newly allocated block of memory in the heap. These address values must be assigned to pointer variables, as explained in the next section.

The first example creates a `Point3d` data structure. The data block allocated in the heap would be just the right size to hold six double precision numbers.

The second example does a little more. The memory manager would allocate block of heap space sufficient to hold a bit map object (an instance of class `Bitmap` from chapter 19). Class `Bitmap` has a constructor function that initializes a bit map. The code generated for the `new` operator has a call to this constructor function so as to initialize the newly allocated data area.

The third example is similar, except that it involves an instance of class `Number`. Class `Number` has several possible constructors; the one needed here is the one that takes a character string. The code generated for the `new` operator would include a call to that constructor so the newly allocated `Number` would be correctly initialized to a value a little over seventy seven thousand million.

new [] operator

The final example creates an array of characters. (The size of the array is determined by the value in the `[]` brackets. Here the value is a constant, but an expression is allowed. This make it possible to work out at run time the size of the array needed for some specific data.) Technically, this last example is using a different operator. This is the `new []` operator (the "make me an array operator").

It is quite common for a program to need to create an array of characters; some examples will occur later.

Illustrations of uses of the `delete` (and `delete []`) operators come toward the end of the next section after pointer variables have been discussed.

20.2 POINTERS

20.2.1 Some "pointer" basics

Defining pointer variables

Pointers are a derived data type. The pointyness, represented by a '*', is a modifier to some basic data type (either built in like `int` or a programmer defined struct or class type). The following are definitions of pointer variables:

```
int *ptr1;
char *ptr2;
Bitmap *ptr3;
Aircraft *ptr4;
```

These definitions make `ptr1` a pointer to a data element that is an `int`; `ptr2` is a pointer to a character; `ptr3` is a pointer to a `Bitmap` object; and `ptr4` is a pointer to an `Aircraft` object. Each of these pointer variables can be used to hold an address; it has to be the address of a data variable of the specified type.

Pointers are type checked to the same degree as anything else is in C++. If you want to store a value in `ptr1`, the value will have to be the address of a variable that is an integer.

Definitions of pointer variables can cause problems. Some of the problems are due to the free format allowed. As far as a C++ compiler is concerned, the following are identical:

```
int *ptr1;
int* ptr1;
int * ptr1;
```

but strictly the `*` belongs with the variable name. It does matter. Consider the following definition:

```
int* pa, pb;
```

What are the data types of `pa` and `pb`?

In this case `pa` is a pointer to an integer (something that can hold the address of an integer variable) while `pb` is an integer variable. The `*` belongs on the variable name;

the definition really is `int *pa, pb;`. If you wanted to define two pointers you would have to write `int *pa, *pb;`.

Although the "pointyness" qualifier `*` associates with a variable name, we need to talk about pointer types independent of any specific instance variable. Thus, we will be referring to `int*` pointers, `char*` pointers, and `Aircraft*` pointers.

Pointers and arrays

At the end of the last section, there was an example that involved creating an array of characters on the heap. The address returned by `new char[10]` has the type "address of array of characters". Now `char*` is a pointer to a character can therefore hold the address of a character. What would be the correct type declaration for a pointer to an array of characters (i.e. something that can hold the address of an array of characters)?

For reasons partly explained in 20.5, a pointer to an array of characters is also `char*`. This makes reading code a bit more difficult. If you see a variable of pointer type being defined you don't know whether it is intended to hold the address of a single instance of the specified data type or is meant to be used to refer to the start of an array of data elements.

"Generic pointers"

Although pointers have types, you quite often need to have functions that use a pointer to data of arbitrary type. A good example is the low-level `write()` function. This function needs a pointer to the memory area that contains the data to be copied to disk and an integer specifying the number of bytes to be copied. The actual write operation involves just copying successive bytes from memory starting at the address specified by the pointer; the same code can work for any type of data. The `write()` function will accept a pointer to anything.

Originally in C, a `char*` was used when the code required a "pointer to anything". After all, a "pointer to anything" must hold the address of a byte, a `char` is a byte, so a pointer to anything is a `char*`. Of course, this just increases the number of possible interpretations of `char*`. It may mean a pointer to a character, or it may mean a pointer to an array of characters, or it may mean a pointer to unspecified data.

void* pointer type

These days, the special type `void*` is generally preferred when a "pointer to anything" is needed. However, a lot of older code, and almost all the older C libraries that you may use from C++, will still use `char*`.

Pointer casts

C++ checks types, and tries to eliminate errors that could arise if you assign the wrong type of data to a variable. So, C++ would quite reasonably object to the following:

```
char      *aPtr;
...
aPtr = new Point3d;
```

Here the new operator is returning "address of a Point3d", a char* is something that holds an "address of a character". The type "address of a Point3d" is not the same as "address of a character". So, the assignment should be challenged by the compiler, resulting in at least a warning if not an error message.

But you might get the same error with the code:

```
Point3d    *bPtr;
...
bPtr = new Point3d;
...
theOutputFile.write(bPtr, sizeof(Point3d));
```

Function write() requires a pointer with the address of the data object, you want it to write the contents of the Point3d whose address is held in bPtr. You would get an error (depends on your compiler and version of iostream) if the function prototype was something like:

```
write(char*, int);
```

The compiler would object that the function wanted a char* and you were giving it a Point3d*.

In situations like this, you need to tell the compiler that you want it to change the interpretation of the pointer type. Although the bPtr really is a "pointer to a Point3d" you want it to be treated as if it were a "pointer to char".

You achieve this by using a "type cast":

Casting to char or void**

```
char      *aPtr;
Point3d    *bPtr;
...
...
bPtr = new Point3d;
...
theOutputFile.write((char*)bPtr, sizeof(Point3d));

aPtr = (char*) new Point3d;
```

The construct:

```
(char*)
    (some address value from a pointer, a function,
     or an operator)
```

tells the compiler to treat the address value as being the address of a character. This allows the value to be assigned to a `char*` variable or passed as the value of a `char*` argument.

If a function requires a `void*` argument, most compilers allow you to use any type of pointer as the actual argument. A compiler allowing this usage is in effect putting a `(void*)` cast into your code for you. Occasionally, you might be required to put in an explicit `(void*)` cast.

Casting from specific pointer types like `Aircraft*`, `Point3d*`, or `Bitmap*` to general types like `char*` and `void*` is safe. Casts that convert general pointers back to specific pointer types are often necessary, but they do introduce the possibility of errors.

Casting a void* to a specific pointer type

In Chapter 21, we look at a number of general purpose data structures that can be used to hold collections of data objects. The example collection structures don't store copies of the information from the original data objects, instead they hold pointers to the data objects. These "collection classes" are intended to work with any kind of data, so they use `void*` data pointers. There is a difficulty. If you ask the object that manages the collection to give you back a pointer to one of the stored data objects you are given back a `void*` pointer.

Thus you get code like the following:

```
class Job {
public:
    Job(int codenum, Name customer, ....);
    ...
    int    JobNumber(void) const;
    ...
};

class Queue {
public:
    ...
    void Append(void* ptr_to_newitem);
    int    Length(void) const;
    void    *First(void);
    ...
};

// make a new job and add it to the Queue
Job*    j = new Job(worknum++, cName, ...);
...
theQueue.Append(j);
...
// Look at next queued job
?? = theQueue.First();
```

The `Queue` object returns a `void*` pointer that holds the address of one of the `Job` objects that was created earlier. But it is a `void*` pointer. You can't do anything much with a `void*`.

A type cast is necessary:

```
Job *my_next_task = (Job*) theQueue.First();
cout << "Now working on " << my_next_task->JobNumber() <<
endl;
```

A cast like this is perfectly reasonable and safe provided the program is properly designed. The programmer is telling the compiler, "you think it could be a pointer to any kind of data, I *know* that it is a pointer to a `Job` object, let me use it as such".

These casts only cause problems if there are design flaws. For example, another programmer might incorrectly imagine that the queue held details of customers rather than jobs and write code like:

```
Customer* c;
// get customer from queue, type cast that void*
c = (Customer*) theQueue.First();
```

The compiler has to accept this. The compiler can't tell that this programmer is using the queue incorrectly. Of course, the second programmer will soon be in difficulties with code that tries to treat a `Job` object as if it were a `Customer` object.

When you write or work with code that type casts from general (`void*`) to specific (`Job*` or `Customer*`) you should always check carefully to verify the assumptions being made in relation to the cast.

Null pointers and uninitialized pointers

Pointers don't have any meaningful value until you've made them point somewhere! **NULL**
You make a pointer variable point somewhere by assigning a value; in C++ this will most often be a value returned by the `new` operator. There is a constant, `NULL`, defined in several of the header files, that represents the concept of "nowhere". You can assign the constant `NULL` to a pointer variable of any type:

```
char      *ptr1 = NULL;
Aircraft  *ptrA = NULL;
...
```

Often you will be working with collections of pointers to data items, a pointer whose value is `NULL` is frequently used to mark the last element of the collection. The loops that control working through the collection are set up so that they stop on finding a `NULL` pointer. These `NULL` pointers serve much the same role as "sentinel values" used in loops that read input (as discussed in Chapter 9).

You can test whether a pointer is `NULL` using code like:

```
if(ptrA != NULL) {
    // Process Aircraft accessed via ptrA
    ...;
}
```

or:

```
if(ptrA) {
    // Process Aircraft accessed via ptrA
    ...;
}
```

In effect, `NULL` equates to 0 or "false". The second form is extremely common; the first version is actually slightly clearer in meaning.

Global and file-scope pointer variables are initialized to `NULL` by the linking-loader. Automatic pointer variables, those defined as local to functions, are not normally initialized. Their initial contents are arbitrary; they contain whatever bit pattern was in memory at the location that corresponds to their place in the function's stack frame.

***Beware of
uninitialized pointers***

An amazingly large proportion of the errors in C and C++ programs are due to programmers using pointers that have never been set to point anywhere. The arbitrary bit pattern in an uninitialized pointer may represent an "illegal address" (e.g. address -5, there is no byte whose address is -5). These illegal addresses are caught by the hardware and result in the operating system stopping the program with an error such as "segmentation fault", "bus error", "system error 2" etc.

Other uninitialized pointers may by chance hold addresses of bytes in one of the program's segments. Use of such an address may result in changes to arbitrary static, stack-based, or heap-based variables or even overwriting of code. Such errors can be quite hard to track down because the problems that they cause frequently don't show up until long after the time that the uninitialized pointer was used. Fortunately, modern compilers can spot many cases where code appears to be using an uninitialized variable; these cases result in warning messages that must be acted on.

Input and output of pointers?

No input, and not much output!

You can get the value of a pointer printed. This is sometimes useful for debugging purposes (the output format usually defaults to hex: for pointers):

```
cout << "ptrA now holds address " << hex << ptrA << endl;
```

In some cases you will need to convert the pointer to a long integer, e.g. `long(ptrA)`.

There is no purpose in writing the value of a pointer to a disk file (nor of writing out a structure that contains pointer data members). The data written would be useless if read back in on a subsequent run of the program.

The value in a pointer is going to be an address, usually one chosen by the run-time memory manager and returned by the `new` operator. If you run the program another time, the run-time memory manager might start with a different area of memory to work with and almost certainly will allocate data areas differently. Consequently, data objects end up at quite different places in memory on different runs. Yesterday's addresses are no use.

20.2.2 Using pointers

Assignment of pointers

Consider the following code fragment:

```
class Demo {
public:
    Demo(int i, char c);
    ...
private:
    int    fi;
    char   fc;
};

Demo::Demo(int i, char c) { fi = i; fc = c; }

int main()
{
    Demo    *ptr1 = NULL;
    Demo    *ptr2 = NULL;
    // stage 1
    ptr1 = new Demo(6, 'a');
    ptr2 = new Demo(7, 'b');
    // stage 2
    ptr2 = ptr1;
    // stage 3
    ...
}
```

The situation in memory after each stage is illustrated in Figure 20.3.

At the end of stage 1, the stack frame for `main()` has been built in the stack and the two pointer variables are both `NULL`; the heap is empty. In stage 2, the two data structures are built in the heap; the addresses returned by `new` are copied into the pointer variables so that these now "point to" their heap structures.

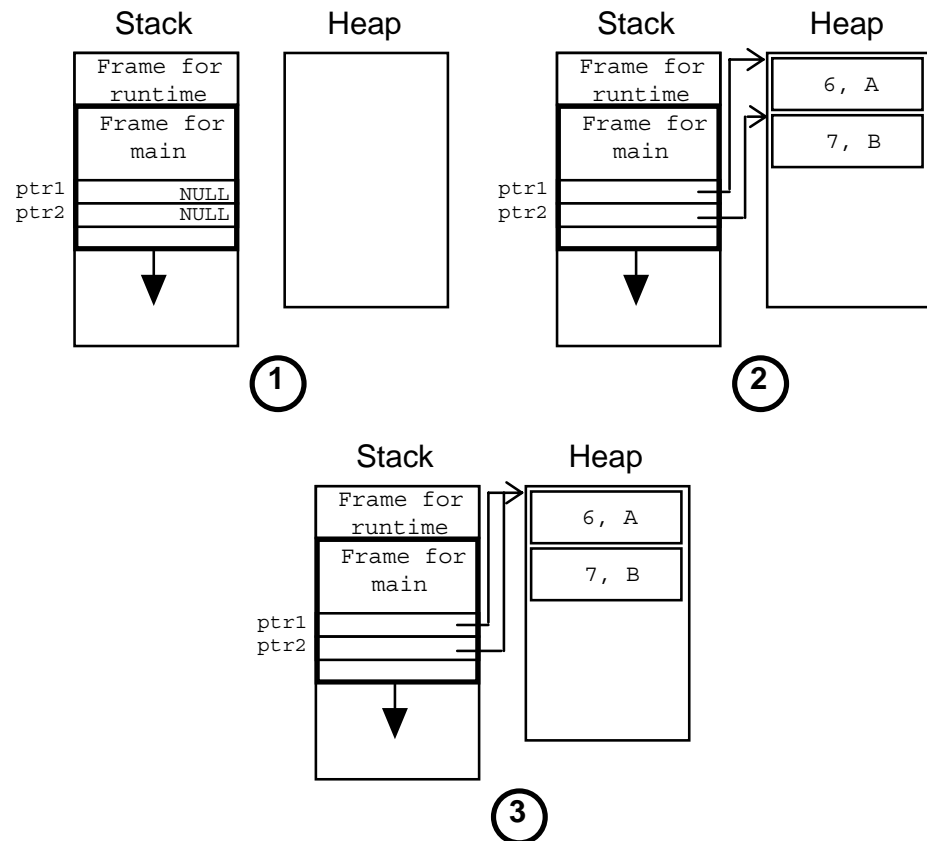


Figure 20.3 Pointer assignment.

In stage 3, the contents of `ptr1` (the address of the Demo object with values 6, A) is copied into `ptr2`. This makes both pointers hold the same address and so makes them point to the same object. (Note that this code would have a memory leak; the second Demo object, 7,B, has been abandoned but remains in the heap.)

Assignment of pointer variables simply means copying an address value from one to another. The data addressed by the pointers are not affected.

Using pointers to access the data members and member functions of structures

If a structure can be accessed by a pointer, its data members can be manipulated. C and C++ have two styles by which data members can be referenced.

The -> operator

The more common style uses the `->` (data member access) operator:

```

struct Thing {
    int      fNum;
    double   fD;
    char      fX;
};

int main()
{
    Thing *pThing;
    pThing = new Thing;
    pThing->fNum = 17;
    pThing->fX = '?';
    pThing->fD = 0.0;
    ...
    ...
    if (pThing->fNum < kLIM)
    ...;
    ...
    xv += pThing->fD;
}

```

The `->` operator takes the name of a (typed) pointer variable on its left (e.g. `pThing`, a `Thing*` pointer), and on its right it takes the name of a data member defined as part of that type (e.g. `fX`; the compiler checks that `fX` is the name of a data member of a `Thing`). The `->` operator produces an address: the address of the specified data member of a structure starting at the location held in the pointer. (So `pThing->fNum` would typically return the address held in `pThing`, `pThing->fD` would return an address value 4 greater, while `pThing->fX` would return an address value 12 greater than the starting address.)

If the expression involving the `->` operator is on the left side of an `=` assignment operator (i.e. it is an "lvalue"), the calculated address specifies where something is to be stored (e.g. as in `pThing->fX = '?'`, where the address calculated defines where the `'?'` is to be stored). Otherwise, the address is interpreted as the place from where a data value is to be fetched (e.g. as in `if (pThing->fNum ...)` or `+= pThing->fD;`).

There is a second less commonly used style. The same operations could be coded as follows: (*). *operator combination*

```

(*pThing).fNum = 17;
(*pThing).fX = '?';
(*pThing).fD = 0.0;
...
...
if ((*pThing).fNum < kLIM)
...;
...
xv += (*pThing).fD;

```

This uses `*` as a "dereferencing" operator. "Dereferencing" a pointer gives you the object pointed to. (Lots of things in C and C++ get multiple jobs to do; we've seen `'&'` work both as a bit wise "And" operator and as the "address of" operator. Now its `*`'s

turn; it may be an innocent multiply operator, but it can also work as a "dereferencing" operator.)

Dereferencing a pointer gives you a data object, in this case a `Thing` object. A `Thing` object has data members. So we can use the `.` data member selection operator to choose a data member. Hence the expressions like `(*pThing).fD`.

Calling member functions

If you have a pointer to an object that is an instance of a class, you can invoke any of its member functions as in the example in 20.2.1:

```
Job *my_next_task;
...
cout << "Now working on " << my_next_task->JobNumber() <<
endl;
```

Manipulating complete structures referenced by pointers

Though you usually want to access individual data members (or member functions) of an object, you sometimes need to manipulate the object as a whole.

You get the object by dereferencing the pointer using the `*` operator. Once you have the object, you can do things like assignments:

```
int main()
{
    Demo  *ptr1 = NULL;
    Demo  *ptr2 = NULL;
    ptr1 = new Demo(6, 'a');
    ptr2 = new Demo(7, 'b');

    // Change the contents of second Demo object to make it
    // identical to the first
    *ptr2 = *ptr1;
    ...
}
```

The `*ptr2` on the left side of the `=` operator yields an address that defines the target area for a copying (assignment) operation. The `*ptr1` on the right hand side of the `=` operator yields an address that is interpreted as the address of the source of the data for the copying operation.

**this*

In the example on class `Number`, we sometimes needed to initialize a new `Number` (result) to the same value as the current object. The code given in the last chapter used an extra `CopyTo()` member function. But this isn't necessary because we can code the required operation more simply as follows:

```
Number Number::Subtract(const Number& other) const
{
    Number result;
    result = *this;

    if(other.Zero_p()) return result;
```

```
    ...
}
```

The implicitly declared variable `this` is a `Number*`. It has been initialized to hold the address of the object that is executing the `Subtract()` function. If we want the object itself, we need to dereference `this` (hence `*this`). We can then directly assign its value to the other `Number` `result`.

Alternatively we could have used the copy constructor `Number(const Number&)`. This function has to be passed the `Number` that is to be copied. So we need to pass `*this`:

```
Number Number::Subtract(const Number& other) const
{
    Number result(*this);

    if(other.Zero_p()) return result;
    ...
}
```

Working with a pointer to an array

The following code fragment illustrates how you could work with an array allocated in the heap. The array in this example is an array of characters. As explained previously, the pointer variable that is to hold the address returned by the `new []` ("give me an array" operator) is just a `char*`. But once the array has been created, we can use the variable as if it were the name of a character array (i.e. as if it had been defined as something like `char ptrC[50]`).

```
#include <iostream.h>

int main()
{
    cout << "How big a string do you want? ";
    int len;
    cin >> len;

    char *ptrC = new char[len];
    for(int i = 0; i < len-1; i++)
        ptrC[i] = '!';
    ptrC[len - 1] = '\0';
    cout << "Change some letters:" << endl;
    for(;;) {
        int lnum;
        char ch;
        cout << "# ";
        cin >> lnum;
        if((lnum < 0) || (lnum >= len-1)) break;

        cout << "ch : ";
```

```

        cin >> ch;

        ptrC[lnum] = ch;
    }

    cout << "String is now ";
    cout << ptrC;
    cout << endl;

    delete [] ptrC;
    return 0;
}

```

Note that the programmer remembered to invoke the `delete []` operator to get rid of the array when it was no longer required.

***Awful warning on
dynamically allocated
arrays***

The code shown makes certain that character change operations are only attempted on characters that are in the range $0 \dots N-2$ for a string of length N (the last character in position $N-1$ is reserved for the terminating `'\0'`). What would happen if the checks weren't there and the code tried to change the -1 th element, or the 8th element of an array $0\dots7$?

The program would go right ahead and change these "characters". But where are they in memory?

If you look at Figure 20.2, you will see that these "characters" would actually be bytes that form the header or trailer housekeeping records of the memory manager. These records would be destroyed when the characters were stored.

Now it may be a long time before the memory manager gets to check its housekeeping records; but when it does things are going to start to fall apart.

Bugs related to overwriting the ends of dynamically allocated arrays are very difficult to trace. Quite apart from the delay before any error is detected, there are other factors that mean such a the bug will be intermittent!

It is rare for programs to have mistakes that result in negative array indices so overwriting the header of a block containing an array is not common. Usually, the errors relate to use of one too many data element (and hence overwriting of the trailer record). But often, the trailer record isn't immediately after the end of the array, there is a little slop of unused space.

The program will have asked for an array of 40 bytes; the memory manager may have found a free block with 48 bytes of space (plus header and trailer). This is close enough; there is no need to find something exactly the right size. So this block gets returned and used. An overrun of one or two bytes won't do any damage.

But the next time the program is run, the memory manager may find a free block of exactly the right size (40 bytes plus header and trailer). This time an overrun causes problems.

Be careful when using dynamically allocated arrays!

Deleting unwanted structures

If you have a pointer to a dynamically allocated struct or class instance you don't want, *delete operator* simply invoke the `delete` operator on that pointer:

```
Aircraft *thePlane;
...
if (thePlane->OutOfFuel())
    delete thePlane;
```

The `delete` operator passes details back to the memory manager which marks the space as free, available for reallocation in future.

The pointer isn't changed. It still holds the value of the now non-existent data structure. This is dangerous. If there is an error in the design of the code, there may be a situation where the data object is accessed after it is supposed to have been deleted. Such code will usually appear to work. Although the memory area occupied by the data object is now marked as "free" it is unlikely to be reallocated immediately; so it will usually contain the data that were last saved there. But eventually, the bug will cause problems.

It is therefore wise to change a pointer after the object it points to is deleted:

```
if (thePlane->OutOfFuel()) {
    delete thePlane;
    thePlane = NULL;
}
```

Setting the pointer to `NULL` is standard. Some programmers prefer to use an illegal address value:

```
thePlane = (Aircraft*)0xf5f5f5f5
```

Any attempt to reuse such a pointer will immediately kill the program with an address error; such an automatic kill makes it easier to find the erroneous code.

The `delete []` operator should be used to free an array created using the `new []` *delete [] operator* operator.

20.2.3 Strings and hash tables revisited

These new pointer types permit slightly better solutions to some of the examples that we have looked at previously.

Strings

Section 11.7 introduced the use of character arrays to hold constant strings and string variables. Things like `Message` were defined by typedefs:

```
typedef char Message[50];
```

(making `Message` a synonym for an array of 50 characters) and arrays of `Message` were used to store text:

```
Message gMessage[] {
    "Undefined symbol",
    "Illegal character",
    ...
    "Does not compute"
};
```

Similar constructs were used for arrays of keywords, or menu options, and similar components introduced in Chapter 12 and later chapters.

These things work. But they are both restricting and wasteful. They are restricting in that they impose a maximum length on the keywords or menu messages. They are wasteful in that usually the vast majority of the keywords are very much shorter than the specified size, but each occupies the same amount of memory.

Sometimes it is helpful to have all data elements the same size. For example, if we want to write some `Messages` to a binary disk file it is convenient to have them all the same size. This gives structure to the file (we can find a specific entry in the `Message` file) and simplifies input.

But if the data are only going to be used in main memory, then there is less advantage in having them the same size and the "wasted space" becomes a more important issue.

The following structure would take up much less overall memory than the `gMessage[]` table defined above:

```
char          *gMsgPtrs[] = {
    "Undefined symbol",
    "Illegal character",
    ...
    "Does not compute"
};
```

Figure 20.4 illustrates how this `gMsgPtrs[]` structure might be represented in memory. This representation has the overhead of a set of pointers (estimate at 4 bytes each) but each individual message string occupies only the amount of space that it needs instead of the 50 bytes previously allocated. Usually, this will result in a significant saving in space.

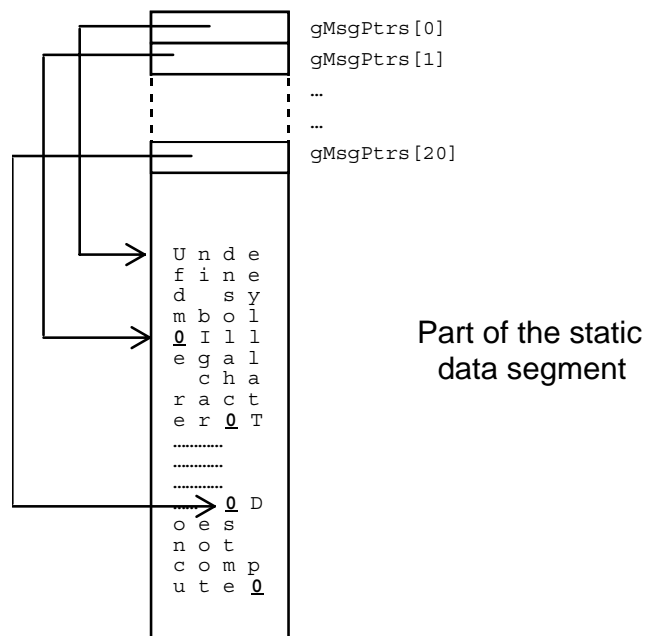


Figure 20.4 Representing an array of initialized character pointers.

The messages referred to by the `gMsgPtrs[]` array would probably be intended as *Pointers to const data* constants. You could specify this:

```
const char *gMsgPtrs[] = {
    "Undefined symbol",
    "Illegal character",
    ...
    "Does not compute"
};
```

This makes `gMsgPtrs[]` an array of pointers to constant character strings. There will be other examples later with definitions of pointers to constant data elements.

Many standard functions take pointers to constant data as arguments. This is very similar to `const` reference arguments. The function prototype is simply indicating that it doesn't change the argument that it can access by the pointer.

Fixed size character arrays are wasteful for program messages and prompts, and they may also be wasteful for character strings entered as data. If you had a program that had to deal with a large number of names (e.g. the program that sorted the list of pupils according to their marks), you could use a struct like the following to hold the data:

```

struct pupil_rec {
    int    fMark;
    char    fName[60];
};

```

but this has the same problem of wasting space. Most pupils will have names less than sixty characters (and you are bound to get one with more than 60).

You would be better off with the following:

```

struct Pupil_Rec {
    int    fMark;
    char    *fName;
};

```

The following example code fragment illustrates creation of structs and heap-based strings:

<p><i>Return NULL if no structure needed</i></p> <p><i>Create structure if necessary</i></p> <p><i>Read string into temporary "buffer"</i></p> <p><i>Make character array of required size and link to struct</i></p> <p><i>Fill in character array</i></p>	<pre> Pupil_Rec *GetPupilRec(void) { cout << "Enter mark, or -1 if no more records" << endl; int mark; cin >> mark; if (mark < 0) return NULL; Pupil_Rec *result = new Pupil_Rec; result->fMark = mark; cin.ignore(100, '\n'); cout << "Enter names, Family name, then given names" << endl; char buffer[200]; cin.getline(buffer, 199, '\n'); int namelen = strlen(buffer); result->fName = new char[namelen+1]; strcpy(result->fName, buffer); return result; } </pre>
---	---

This function is defined as returning a pointer to a `Pupil_Rec`; it will have to create this `Pupil_Rec` structure on the heap using the `new` operator. If the mark input is negative, it means there is no more input; in this case the function returns `NULL`. This is a very typical style in a program that needs to create a number of data records based on input data. The calling program can have a loop of the form `while((rec = GetPupilRec()) != NULL) { ... }`.

If the mark is not negative, another `Pupil_Rec` struct is created using the `new` operator and its `fMark` field is initialized. Any trailing input following the mark is removed by the call to `ignore()`. The function then prompts for the pupil's names.

There has to be some character array allocated into which the name can be read. This is the role of `buffer`. It gets filled with a complete line from the input.

The number of characters needed for the name is then determined via the call to `strlen()`. A character array is allocated on the heap using `new []` (the character array is one longer than the name so as to leave room for a terminating `'\0'` character). The name is then copied into this array.

The new structure has been built and so it can be returned. Note how every structure is represented by two separate blocks of information in the heap, with the character array linked to the main `Pupil_Rec` struct. In most of your later programs you will have hundreds if not thousands of separate objects each stored in some block of bytes on the heap. Although separately allocated in the heap, these objects are generally linked together to represent quite elaborate structural networks.

Hash table

Sections 18.2.2 and 18.2.3 contained variations on a simple hash table structure. One version had an array of with fixed 20 character words, the other used an array of structs incorporating fixed sized words and an integer count. These arrays again "wasted" space.

Ideally, a hash table should be not much more than half occupied; so many, perhaps almost half of the table entries will not be used even when all the data have been loaded. Most of the words entered into the hash table would have been less than twelve characters so much of the space allocated for each word is wasted (and of course the program can't deal with the few words that do exceed twenty characters).

A slightly more space-efficient variation would use a table of pointers to character arrays (for the version that is to store words) or pointers to structs for the version that needs words and associated integer data values). This arrangement for the table for words would be as shown in Figure 20. 5 (compare with version in Figure 18.1). Note that the words represented as separately allocated arrays in the heap do require those headers and trailers. The space costs of these cut into the savings made by only using the number of characters required for each word. The only significant saving is going to be that due to the fact that only half the table is populated.

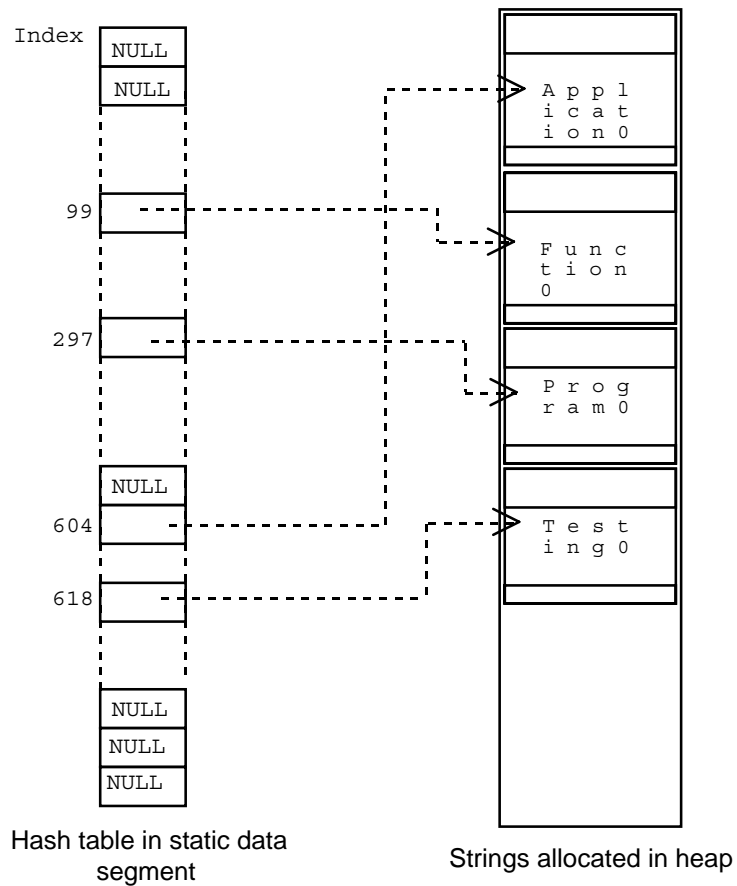


Figure 20.5 Hash table using pointers to strings.

The changes to the code needed to use the modified structure aren't substantial. The following are rewrites of a couple of the functions given in Section 18.2.

```
const int kTBLSIZE      = 1000;
char *theTable[kTBLSIZE];

void InitializeHashTable(void)
{
    for(int i=0; i< kTBLSIZE; i++)
        theTable[i] = NULL;
}

int NullEntry(int ndx)
```

```
{
    return (theTable[ndx] == NULL);
}
```

Functions `InitializeHashTable()` and `NullEntry()` both have minor changes to set pointers to `NULL` and to check for `NULL` pointers.

The code for `MatchEntry()` is actually unchanged! There is a subtle difference. Previously the argument was an array of characters, now it is a pointer to an array of characters. But because of the general equivalence of arrays and pointers these can be dealt with in the exact same manner.

```
int MatchEntry(int ndx, const char str[])
{
    return (0 == strcmp(str, theTable[ndx]));
}
```

The `InsertAt()` function has been changed so that it allocates an array on the heap to store the string that has to be inserted. The string's characters are copied into this new array. Finally, the pointer in the table at the appropriate index value is filled with the address of the new array.

```
void InsertAt(int ndx, const char str[])
{
    char *ptr;
    ptr = new char[strlen(str) + 1];
    strcpy(ptr, str);
    theTable[ndx] = ptr;
}
```

20.3 EXAMPLE: "AIR TRAFFIC CONTROLLER"

Problem

Implement the simulated "Air Traffic Control" (ATC) trainer/game described below.

The ATC trainer is to give users some feel for the problems of scheduling and routing aircraft that are inbound to an airport. Aircraft are picked up on radar at a range of approximately 150 miles (see Figure 20.6). They are identified by call sign, position, velocity, acceleration, and details of the number of minutes of fuel remaining. An aircraft's velocity and acceleration are reported in terms of x' , y' , z' and x'' , y'' , z'' components. The user (game-player, trainee air-controller, or whatever) must direct aircraft so that they land successfully on the single east-west runway (they must approach from the west).

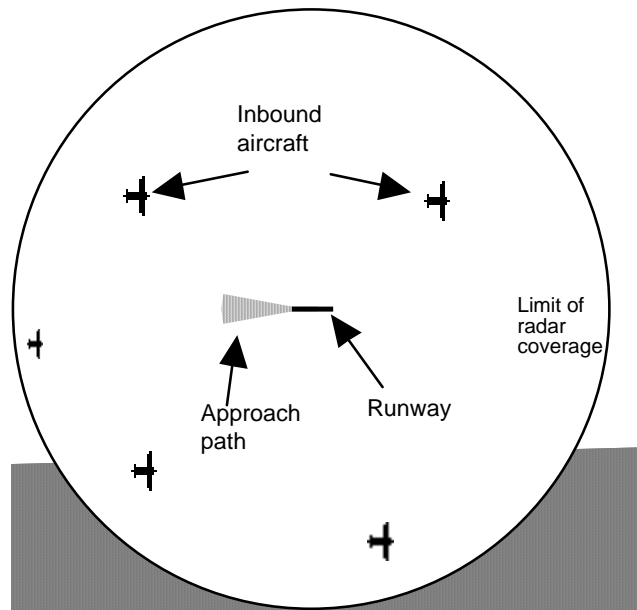


Figure 20.6 The "Air Traffic Control" problem.

There are a number of potential problems that the controller should try to avoid. If an aircraft flies too high, its engines stall and it crashes. If it flies too low it runs into one of the hills in the general vicinity of the airport. If it runs out of fuel, it crashes. If its (total) horizontal speed is too low, it stalls and crashes. If the horizontal speed is too high, or if it is descending too fast, the wings fall off and, again, it crashes.

At each cycle of the simulation, the controller may send new directives to any number of inbound aircraft. Each cycle represents one minute of simulated time. A directive specifies the accelerations (as x , y , z components) that should apply for a specified number of minutes. Subsequent directives will override earlier settings. The aircraft pilot will check the directive. If accelerations requested are not achievable, or it is apparent that they would lead to excessive velocities within a minute, the directive is ignored. An acceptable directive is acknowledged.

At each cycle of the simulation, every aircraft recomputes its position and velocity. Normally, one unit of fuel is burnt each minute. If an aircraft has a positive vertical acceleration ("*Climb!*", "*Pull out of that dive!*", ...) or its horizontal accelerations lead to an increase in its overall horizontal speed, it must have been burning extra fuel; in such cases, its remaining fuel has to be decremented by one extra unit.

Aircraft must be guided so that they line up with the approach path to the runway. If they are flying due east at an appropriate velocity, within certain height limits and with some remaining fuel, they may be handed over to the airport's automated landing

system. These aircraft are considered to have been handled successfully, and are deleted from the simulation.

Details

Controlling aircraft by specifying accelerations is unnatural, but it makes it easy to write code that defines how they move. You need simply remember your high school physics equations: *Equations of motion for aircraft*

```

u  initial velocity
v  final velocity
s  distance travelled
α  acceleration
t  time

v = u + α * t
s = u * t + 0.5 * α * t2

```

When recomputing the position and velocity of an aircraft, the x, y (and x', y') components can be treated separately. Just calculate the distance travelled along each axis and update the coordinates appropriately. The total ground speed is given by:

$$\text{speed} = \sqrt{(x')^2 + (y')^2}$$

The following limits can be used for aircraft performance (pure inventions, no claims to physical reality): *Aircraft limits*

Maximum height	40000 feet
Minimum safe height	450 feet
Maximum horizontal speed	12 miles per minute (mpm)
Minimum horizontal speed	2 mpm
Maximum rate of descent	600 feet per minute (fpm)
Maximum rate of ascent	1000 fpm
Maximum acceleration/ deceleration (horizontal)	2 miles per minute per minute
Maximum positive vertical acceleration	800 feet per minute per minute
Maximum negative vertical acceleration	400 fpm per minute

The maximum rate of ascent is simply a limit value, you cannot go faster even if you try, but nothing disastrous happens if you do try to exceed this limit. The other speed limits are critical, if you violate them the aircraft suffers.

Landing conditions

For an aircraft to achieve a safe landing, the controller must bring it into the pick up area of the automated landing system. The aircraft must then satisfy the following constraints:

```

Approach height          (600 ... 3000) feet
Distance west of the runway 4 miles ... 10 miles
Distance north/south of runway < 0.5 miles
Fuel remaining           > 4 minutes
x'                        +2 ... +3 mpm
    (i.e. flying east 120-180mph)
y'                        -0.1...+0.1 mpm
    (minimal transverse speed)
z'                        -500 ... 0 fpm
    (descending, but not too fast)
x'', y'', z''            • 0 (no accelerations)

```

Reports and command interface

A relatively crude status display, and command line interface for user input will suffice. For example, a status report listing details of all aircraft in the controlled space could be something like:

```

BA009  (...,...) (...,...) (...,...) fuel = ...
JL040  (...,...) (...,...) (...,...) fuel = ...

```

Each line identifies an aircraft's call sign; the first triple gives its x, y, z position (horizontal distances in miles from the airport, height in feet); the second triple gives the x', y', z' velocities (miles per minute, feet per minute); the third triple is the accelerations x'', y'', z'' (miles per minute per minute, feet per minute per minute); finally the number minutes of fuel remaining is given. For example:

```

BA009  (-128,4.5,17000) (4,-0.5,-1000) (0,0,0) fuel = 61

```

This aircraft is flying at 17000 feet. Currently it is 128 miles west of the airport and 4.5 miles north of the runway. Its total ground speed is approximately 4.03 miles per minute (241mph). It is descending at 1000 feet per minute. It is not accelerating in any way, and it has 61 minutes of fuel remaining.

The user has to enter commands that tell specific aircraft to accelerate (decelerate). These commands will have to include the aircraft's call sign, and details of the new accelerations. For example:

```

BA009  0 0 100 3

```

This command instructs plane BA009 to continue with no horizontal accelerations (so no change to horizontal velocities), but with a small positive vertical acceleration that is to apply for next three minutes (unless changed before then in a subsequent command). This will cause the aircraft to reduce its rate of descent.

Adding aircraft to the controlled airspace

Entry of planes into controlled air-space can be handled by a function that uses arrays of static data. One array of integers can hold the arrival times (given in minutes

from the start of simulation), another array of simple structs can hold details of flight names and the initial positions and velocities of the aircraft. If this function is called once on each cycle of the simulation, it can return details of any new aircraft (assume at most one aircraft arrives per minute).

Design

To start, what are the objects needed in this program?

The objects and their classes

Aircraft are obvious candidates. The program is all about aircraft moving around, responding to commands that change their accelerations, crashing into hills, and diving so fast that their wings fall off. Each aircraft owns some data such as its call sign, its current x, y, z position, its x', y', z' velocities. Each aircraft provides all services related to its data. For example, there has to be a way of determining when an aircraft can be handed over to the automated landing system. The controller shouldn't ask an aircraft for details of its position and check these, then ask for velocities and check these. Instead the controller should simply ask the aircraft "Are you ready to auto land?"; the aircraft can check all the constraints for itself.

So we can expect class Aircraft. Class Aircraft will own a group of data members and provide functions like "Update accelerations, change to these new values" and "Check whether ready to auto-land", "Fly for another minute and work out position".

class Aircraft

A class representing the "air traffic controller" is not quite so obvious. For a start, there will only ever be one instance of this class around. It gets created at the start of the game and is used until the game ends.

class Aircontroller?

When you've programmed many applications like this you find that usually it is helpful to have an object that owns most of the other program elements and provides the primary control functions.

In this example, the aircontroller will own the aircraft and will provide a main "run" function. It is in this "run" function that we simulate the passage of time. The "run" function will have a loop, each cycle of which represents one minute of elapsed time. The run-loop will organize things like letting each aircraft update its position, prompting the user to enter commands, updating some global timer, and checking for new aircraft arriving.

The main program can be simplified. It will create an "AirController" and tell it to "run".

The process of completing the design will be iterative. Each class will get considered and its details will be elaborated. The focus of attention may then switch to some other class, or maybe to a global function (a function that doesn't belong to any individual class). When other details have been resolved, a partial design for a class may get reassessed and then expanded with more detail.

class Aircontroller

What does an Aircontroller own?

Data owned

It is going to have to have some data members that represent the controlled airspace. These would probably include a count of the number of aircraft in the controlled space, and in some way will have to include the aircraft themselves. The aircraft are going to be class instances created in the heap, accessed by Aircraft* pointers. The Aircontroller object can have an array of pointers.

What does an Aircontroller do?

Services provided

It gets created by the main program and then runs. So it will need a constructor that initializes its records to show that the airspace is empty, and a "run" function that controls the simulation loop.

The main loop is going to involve the following steps:

```
Update a global timer
Let all planes move
Report current status, getting each plane to list its
details
Get all planes to check whether they've crashed or can
auto-land, remove those no longer present in the
airspace
if can handle more planes (arrays not full)
    check if any arrived
if airspace is not empty
    give user the chance to enter commands
    check for quit command
```

Naturally, most of these steps are going to be handled by auxiliary private member functions of the Aircontroller class.

The timer will be a global integer variable. It is going to get used by the function that adds aircraft to the airspace. (In this program, the timer could be a local variable of the "run" function and get passed as an argument in the call to the function that adds aircraft. However, most simulations require some form of global variable that represents the current time; so we follow the more general style.)

The array of pointers to aircraft will start with all the pointers NULL. When an aircraft is added, a NULL pointer is changed to hold the address of the new aircraft. When an aircraft crashes or lands, it gets deleted and the pointer with its address is reset to NULL. At any particular moment in the simulation, the non-NULL entries will be scattered through the array. When aircraft need to be activated, a loop like the following can be used:

```
for i = 0; i < max; i++
    if aircraft[i] != NULL
        aircraft[i] do something
```

Similar loop constructs will be needed in several of the auxiliary private member functions of class Aircontroller.

For example, the function that lets all the planes move is going to be:

```

move planes
for i = 0; i < max; i++
    if aircraft[i] != NULL
        aircraft[i] move

```

while that which checks for transfers to the auto lander would be:

```

check landings
for i = 0; i < max; i++
    if aircraft[i] != NULL
        if aircraft[i] can transfer
            report handoff to auto lander
            delete the aircraft
            aircraft[i] = NULL;
            reduce count of aircraft in controlled space

```

Another major role for the Aircontroller object will be to get commands from the user. The user may not want to enter any commands, or may wish to get one plane to change its accelerations, or may need to change several (or might even wish to quit from the game). It would be easiest to have a loop that kept prompting the user until some "ok no more commands" indicator was received. The following outline gives an idea for the structure:

Getting user commands

```

Get user commands
prompt for command entry
loop
    read word
    if( word is "Quit")
        arrange to terminate program
    if( word is "OK")
        return

    read accelerations and time

    if(any i/o problems) warn, and just ignore data
    else handle command

```

Handling commands will involve first finding the aircraft with the call sign entered, then telling it of its new accelerations:

```

Handle command
    identify target aircraft (one with call sign entered)
    if target not found
        warn of invalid call sign
    else tell target to adjust accelerations

```

The target can be found in another function that loops asking each aircraft in turn whether it has the call sign entered.

class Aircraft

Data owned What does an individual aircraft own?

There is a whole group of related data elements – the call sign, the coordinates, velocities. When an aircraft is created, these are to be filled in with predefined data from a static array that specifies the planes. It would actually be convenient to define an extra struct whose role is just to group most or all of these data elements. The predefined data used to feed aircraft into the airspace could be represented as an array of these structs.

So, we could have:

```
struct PlaneData {
    double    x, y, z;        // coords
    double    vx, vy, vz;    // velocities
    ...
    short     fuel;           // minutes remaining
    char      name[7];        // 6 character call name
};
```

An Aircraft object would have a PlaneData as a data member, and maybe some others.

What do Aircraft do?

Aircraft are going to be created; when created they will be given information to fill in their PlaneData data member. They will get told to move, to print their details for reports, to change their accelerations, to check whether they can land, and to check whether they are about to suffer misfortune like flying into the ground. There were quite a number of "terminating conditions" listed in the problem description; each could be checked by a separate private member function.

Some of the member functions will be simple:

```
toofast
    return true if speed exceeds safe maximum

tooslow
    return true if speed exceeds safe minimum
```

The overall horizontal speed would have to be a data member, or an extra auxiliary private member function should be added to calculate the speed.

The function that checks for transfer to auto lander will also be fairly simple:

```
making final approach
    if(not in range 4...10 miles west of runway)
        return false
    if(too far north or south)
        return false
    if( speed out of range)
        return false
    ...
```

```
return true
```

The "move" and "update" functions are more complex. The update function has to check that the new accelerations are reasonable:

```
update
    check time value (no negatives!)

    check each acceleration value against limits
    if any out of range ignore command

    calculate velocity components after one minute
    of new acceleration

    if any out range ignore command

    acknowledge acceptable command

    change data members
```

The move function has to recompute the x, y, z coordinates and the velocities. If the specified number of minutes associated with the last accelerations has elapsed, these need to be zeroed. The fuel left has to be reduced by an appropriate amount.

```
move

    calculate distance travelled
    assuming constant velocity

    if (command time > 0)
        /* Still accelerating */
        work out new velocities,
            (note: limit +ve vertical velocity)
        correct distances to allow for accelerations

    allow for extra fuel burn if accelerating
    decrement command time
        and if now zero clear those
        accelerations

    fix up coordinates
    allow for normal fuel usage
```

Aircraft generating function

This would use a global array with "arrival times" of aircraft, another global array with aircraft details, and an integer counter identifying the number of array entries already used. A sketch for the code is:

```
new arrivals function
    if time < next arrival time
        return null
```

```

        create new aircraft initializing it with data
        from PlaneData array

        update count of array entries processed

        return aircraft

```

Refining the initial designs

The initial designs would then be refined. For the most part, this would involve further expansion of the member functions and the identification of additional auxiliary private member functions. The process used to refine the individual functions would be the same as that illustrated in the examples in Part III.

The design process lead to the following class definitions and function specifications:

```

struct PlaneData {
    double x,y,z;
    double vx,vy,vz;
    double ax,ay,az;
    short fuel;
    char   name[7];
};

class Aircraft {
public:
    Aircraft(const PlaneData& d);
    void      PrintOn(ostream& os) const;
    void      Update(double a1,
                    double a2, double a3, short timer);
    void      Move();
    Boolean   MakingApproach() const;
    Boolean   Terminated() const;
    Boolean   CheckName(const char str[]) const;
    const char* ID() const;
private:
    Boolean   Stalled() const;
    Boolean   Pancaked() const;
    Boolean   NoFuel() const;
    Boolean   TooFast() const;
    Boolean   TooSlow() const;
    Boolean   CrashDiving() const;
    double    Speed() const;

    PlaneData fData;          // main data
    short     fTime;          // time remaining for command
};

```

(The implementation uses a typedef to define "Boolean" as a synonym for unsigned char and defines false and true.) The member functions would be further documented:

```
Aircraft(const PlaneData& d);  
    Initialize new Aircraft from PlaneData  
  
void      PrintOn(ostream& os) const;  
    Output details as needed in report  
  
void      Update(double a1,  
                 double a2, double a3, short timer);  
    Verify acceleration arguments, update data members if appropriate,  
    acknowledge.  
  
void      Move();  
    Recompute position and velocity.  
  
Boolean   MakingApproach() const;  
    Check whether satisfies landing conditions.  
  
Boolean   Terminated() const;  
    Check if violates any flight constraints.  
  
Boolean   CheckName(const char str[]) const;  
    Check str argument against name  
  
const char* ID() const;  
    Return name string (for reports like "XXX has landed/crashed")  
  
Boolean   Stalled() const;  
    Check if flown too high.  
  
Boolean   Pancaked() const;  
    Check if flown too low.  
  
Boolean   NoFuel() const;  
    Check fuel.  
  
Boolean   TooFast() const;  
    Check if flown too fast.  
  
Boolean   TooSlow() const;  
    Check if flown too slow.  
  
Boolean   CrashDiving() const;  
    Check if exceeding descent rate.  
  
double    Speed() const;  
    Check ground speed
```


The AirControlller class is:

```
class AirController {
public:
    AirController();
    void        Run();
private:
    void        Report();
    void        MovePlanes();
    void        AddPlanes();
    void        Validate();
    void        CheckCrashes();
    void        CheckLandings();

    Boolean     GetCommands();
    void        HandleCommands(const char name[],
                                double,double,double,short);
    Aircraft    *IdentifyByCall(const char name[]);

    Aircraft    *fAircraft[kMAXPLANES];
    int         fControlling;
};
```

The member functions are:

```
AirController();
    Constructor, initialize air space to empty.

void        Run();
    Run main loop moving aircraft, checking commands etc

void        Report();
    Get reports from each aircraft.

void        MovePlanes();
    Tell each aircraft in turn to move to new position.

void        AddPlanes();
    Call the plane generating function, if get plane returned add it to array
    and notify player of arrival.

void        Validate();
    Arrange checks for landings and crashes.

void        CheckCrashes();
    Get each plane in turn to check if any constraints violated.

void        CheckLandings();
    Get each plane in turn to check if it can autoland, sign off those
    that can autoland.
```

```

Boolean    GetCommands();
           Get user commands, returns true if "Quit"

void        HandleCommands(const char name[],
                           double, double, double, short);
           Pass a "change accelerations" command on to correct aircraft.

Aircraft    *IdentifyByCall(const char name[]);
           Identify aircraft from call sign.

```

(Really, the program should check for aircraft collisions as well as landings and crash conditions. Checking for collisions is either too hard or too boring. The right way to do it is to calculate the trajectories (paths) flown by each plane in the past minute and determine if any trajectories intersect. This involves far too much mathematics. The boring way of doing the check is to model not minutes of elapsed times but seconds. Each plane moves for one second, then distances between each pair of planes is checked. This is easy, but clumsy and slow.)

Implementation

The following are an illustrative sample of the functions. The others are either similar, or trivial, or easy to code from outlines given earlier.

The `main()` function is concise:

```

int main()
{
    AirController me;
    me.Run();
    return 0;
}

```

This is actually a fairly typical `main()` for an object-based program – create the principal object and tell it to run.

The file with the application code will have the necessary `#includes` and then start with definition of constants representing limits:

```

// Constants that define Performance Limits
const double kMaxHeight          =
40000.0;
const double kMinHeight          = 450.0;

...
const double kMaxVDownAcceleration = -400.0;

// Constants that define conditions for landing
const double      kminh           = 600.0;

```

```

const double    kmaxh                      = 3000.0;
const double    kinnerrange                 = -4.0;
...
const short     kminfuel                    = 4;

```

There are a couple of globals, the timer and a counter that is needed by the function that adds planes:

```

static long     PTimer = 0;
static short     PNum = 0;

```

The arrays with arrival times and prototype aircraft details would have to be defined:

```

short           PArrivals[] = {
    5, 19, 31, 45, 49,
    ...
};

PlaneData ExamplePlanes[] = {
    { -149.0, 12.0, 25000.0,
      7.0, 0.0, -400.0,
        0.0, 0.0, 0.0,
          120, "BA009" },
    { -144.0, 40.0, 25000.0,
      4.8, -1.4, 0.0,
        0.0, 0.0, 0.0,
          127, "QF040" },
    ...
};

static short NumExamples = sizeof(ExamplePlanes) /
                           sizeof(PlaneData);

```

When appropriate, function `NewArrival()` creates an Aircraft on the heap using the `new` operator; a `PlaneData` struct is passed to the constructor. The new Aircraft is returned as a result. The value `NULL` is returned if it was not time for a new Aircraft.

```

Aircraft *NewArrival(void)
{
    if(PNum==NumExamples) return NULL;

    if(PTimer < PArrivals[PNum]) return NULL;
    Aircraft *newPlane = new Aircraft(ExamplePlanes[PNum]);
    PNum++;
    return newPlane;
}

```

The constructor for controller simply sets all elements of the array `fAircraft` to `NULL` and zeros the `fControlling` counter. The main `Run()` function is:

```
void AirController::Run()
{
    for(Boolean done = false; !done;) {
        PTimer++;
        MovePlanes();
        Report();
        Validate();
        if(fControlling < kMAXPLANES)
            AddPlanes();
        if(fControlling>0)
            done = GetCommands();
    }
}
```

The `Report()` and `MovePlanes()` functions are similar; their loops involve telling each aircraft to execute a member function (`PrintOn()` and `Move()` respectively). Note the way that the member function is invoked using the `->` operator (`fAircraft[i]` is a pointer to an `Aircraft`).

```
void AirController::Report()
{
    if(fControlling < 1) cout << "Airspace is empty\n";
    else
        for(int i=0;i<kMAXPLANES; i++)
            if(fAircraft[i] != NULL)
                fAircraft[i]->PrintOn(cout);
}
```

The `AddPlanes()` function starts with a call to the plane generator function `NewArrival()`; if the result from `NewArrival()` is `NULL`, there is no new `Aircraft` and `AddPlanes()` can exit.

```
void AirController::AddPlanes()
{
    Aircraft* aPlane = NewArrival();
    if(aPlane == NULL)
        return;

    cout << "New Aircraft:\n\t";
    fControlling++;
    aPlane->PrintOn(cout);
    for(int i=0;i<kMAXPLANES; i++)
        if(fAircraft[i] == NULL) {
            fAircraft[i] = aPlane;
            aPlane = NULL;
            break;
        }
    if(aPlane != NULL) {
        cout << "??? Planes array full, program bug??"
            << endl;
    }
}
```

```

        delete aPlane;
    }
}

```

In other cases, the count of aircraft in the controlled space is increased, and an empty slot found in the `fAircraft` array of pointers. The empty slot is filled with the address of the newly created aircraft.

The `CheckCrashes()` and `CheckLanding()` functions called from `Validate()` are similar in structure. They both have loops that check each aircraft and dispose of those no longer required:

```

void AirController::CheckCrashes()
{
    for(int i=0;i<kMAXPLANES; i++)
        if((fAircraft[i] != NULL) &&
            (fAircraft[i]->Terminated())) {
                cout << fAircraft[i]->ID();
                cout << " CRASHED!" << endl;
                delete fAircraft[i];
                fAircraft[i] = NULL;
                fControlling--;
            }
}

```

Input to `GetCommands()` will start with a string; this should be either of the key words `OK` or `Quit` or the name of a flight. A buffer of generous size is allocated to store this string temporarily while it is processed.

The function checks for, and acts on the two key word commands. If the input string doesn't match either of these it is assumed to be the name of a flight and the function therefore tries to read three doubles (the new accelerations) and an integer (the time). Checks on input errors are limited but they are sufficient for this kind of simple interactive program. If data are read successfully, they are passed to the `HandleCommand()` function.

```

Boolean AirController::GetCommands()
{
    char buff[120];
    cout << "Enter Flight Commands:\n"; cout.flush();

    for(;;) {
        cin >> buff;
        if(0 == ::strcmp(buff,"Quit")) return true;
        if(0 == ::strcmp(buff,"OK")) {
            cin.ignore(SHRT_MAX, '\n');
            return false;
        }
        double a1,a2,a3;
        short t;
        cin >> a1 >> a2 >> a3 >> t;
        if(cin.fail()) {

```

```

        cout << "Bad input data\n";
        cin.clear();
        cin.ignore(SHRT_MAX, '\n');
    }
    else HandleCommands(buff, a1, a2, a3, t);
}
}

```

The `HandleCommands()` function uses the auxiliary function `IdentifyByCall()` to find the target. If target gets set to `NULL` it means that flight identifier code was incorrectly entered. If the target is found, it is asked to update its accelerations:

```

void AirController::HandleCommands(const char* call, double
a1, double a2, double a3, short n)
{
    Aircraft*    target = IdentifyByCall(call);

    if(target == NULL) {
        cout << "There is no aircraft with " << call
            << " as call sign.\n";
        return;
    }

    target->Update(a1, a2, a3, n);
}

```

Function `IdentifyByCall()` loops through the collection of aircraft asking each in turn whether its call sign matches the string entered by the user. The function returns a pointer to an `Aircraft` with a matching call sign, or `NULL` if none match.

```

Aircraft *AirController::IdentifyByCall(const char name[])
{
    for(int i=0; i < kMAXPLANES; i++)
        if((fAircraft[i] != NULL) &&
            (fAircraft[i]->CheckName(name)))
            return fAircraft[i];
    return NULL;
}

```

The constructor for `Aircraft` initializes its main data from the `PlaneData` struct passed as an argument and zeros the command timer. The `PrintOn()` function simply outputs the data members:

```

Aircraft::Aircraft(const PlaneData& d)
{
    fData = d;
    fTime = 0;
}

void Aircraft::PrintOn(ostream& os) const

```

```

{
    os.setf(ios::fixed,ios::floatfield);
    os.precision(2);
    os << fData.name;

    os << "\t(" << fData.x << "," << fData.y << ","
    << fData.z << ")\t(";
    ...
    os << endl;
}

```

An aircraft's name is needed at a couple of points in the code of `AirController`. The `ID()` function returns the name as a `char*` ("pointer to array of characters").

*Returning const something**

Now the address returned is that of the `fData.name` data member. If you return the address of a data member, the "wall around the data" can be breached. The data values can be changed directly instead of through member functions. Any pointers to data members that are returned should be specified as "pointers to constant data". So here the return type is `const char*` (i.e. a pointer to an array of characters that shouldn't be changed).

```

const char* Aircraft::ID() const
{
    return fData.name;
}

```

Function `Move()` sorts out the changes to position, velocity, and fuel:

```

void Aircraft::Move()
{
    double dx, dy, dz;
    dx = fData.vx;
    dy = fData.vy;
    dz = fData.vz;

    double oldspeed = Speed();

    if(fTime>0) {
        /* Still accelerating */
        fData.vx += fData.ax; dx += fData.ax * 0.5;
        fData.vy += fData.ay; dy += fData.ay * 0.5;
        fData.vz += fData.az; dz += fData.az * 0.5;

        if((fData.az>0) || (Speed() > oldspeed))
            fData.fuel--;

        if(fData.vz>kMaxAscentRate) {
            fData.vz = kMaxAscentRate;
            fData.az = 0.0;
        }

        fTime--;
        if(fTime==0)
            fData.az = fData.ay = fData.ax = 0.0;
    }
}

```

```

    }
    fData.x += dx;
    fData.y += dy;
    fData.z += dz;
    fData.fuel--;
}

```

The `Update()` function performs a series of checks on the new acceleration values. If all checks are passed, the values in the data members are changed and an acknowledgment is printed:

```

void Aircraft::Update(double a1, double a2,
    double a3, short timer)
{
    /* validate input */
    if(timer < 1) return;

    /* accelerations, x and y limited by Horizontal
    acceleration */
    if(fabs(a1) > kMaxHAcceleration) return;
    if(fabs(a2) > kMaxHAcceleration) return;

    /* Vertical bracketed in range. */
    if(a3 > kMaxVUpAcceleration) return;
    if(a3 < kMaxVDownAcceleration) return;

    /* check that new velocities not excessive */
    double newvertvelocity = fData.vz + a3;
    if(newvertvelocity > kMaxAscentRate) return;
    if(newvertvelocity < kMaxDescentRate) return;

    double newvx, newvy, newtotal;
    newvx = fData.vx + a1;
    newvy = fData.vy + a2;
    newtotal = sqrt(newvx*newvx + newvy*newvy);
    if(newtotal > kMaxSpeed) return;
    if(newtotal < kMinSpeed) return;

    cout << fData.name << ": Roger\n";
    fData.ax = a1;
    fData.ay = a2;
    fData.az = a3;
    fTime = timer;
}

```

Function `MakingApproach()` involves a series of checks against the constraints that define the "auto landing" conditions:

```

Boolean Aircraft::MakingApproach() const
{
    /* in pick up range of auto lander? */
    if((fData.x < kouterrange) ||
        (fData.x > kinnerrange)) return false;
}

```



```

    /* Aligned with runway? */
    if(fabs(fData.y) > koffline) return false;

    /* In height bracket? */
    if((fData.z < kminh) ||
        (fData.z > kmaxh)) return false;

    /* In velocity bracket? */
    if((fData.vx < kxprimelow) ||
        (fData.vx > kxprimehigh)) return false;

    ...

    /* and no real accelerations? */
    if(fabs(fData.ax) > kxalphalimit) return false;
    ...

    /* and sufficient fuel? */
    if(fData.fuel < kminfuel) return false;

    return true;
}

```

Function `Terminated()` uses the various auxiliary member functions to check for terminating conditions:

```

Boolean Aircraft::Terminated() const
{
    return
        Stalled() || Pancaked() || NoFuel() ||
        TooFast() || TooSlow() || CrashDiving();
}

```

The remaining functions of class `Aircraft` are all simple; representative examples are:

```

Boolean Aircraft::CheckName(const char str[]) const
{
    return (0 == strcmp(fData.name, str));
}

Boolean Aircraft::Stalled() const
{
    return fData.z > kMaxHeight;
}

double Aircraft::Speed() const
{
    return sqrt(fData.vx*fData.vx + fData.vy*fData.vy);
}

```

The program runs OK but as a game it is a little slow (and directing an aircraft to land safely is surprisingly hard). The following is a fragment from a recording of the game:

```
Airspace is empty
New Aircraft:
BA009 (-149,12,25000) (7,0,-400) (0,0,0) fuel: 120
Enter Flight Commands:
BA009 is west of the runway, a little to the north and
coming in much too fast; slow it down, make it edge south a
bit
BA009 -1 -0.2 0 3
BA009: Roger
OK
BA009 (-142.5,11.9,24600) (6,-0.2,-400) (-1,-0.2,0)
fuel: 119
...
BA009 (-124.5,9.9,23000) (4,-0.6,-400) (0,0,0)
fuel: 115
Enter Flight Commands:
Slow up a little more, increase rate of descent
BA009 -0.5 0 -100 2
BA009: Roger
OK
BA009 (-120.75,9.3,22550) (3.5,-0.6,-500) (-0.5,0,-100)
fuel: 114
...
BA009 (-15.5,0,3350) (3,-0,-500) (0,0,0) fuel: 78
QF040 (-52.77,0.18,11600) (3.01,-0.05,-600) (0,0,0) fuel:
100
NZ164 (-71,-45.95,20500) (1,5.3,-500) (0,0,0) fuel: 70
CO1102 (116,-87,32500) (-5,1,-500) (0,0,0) fuel: 73
Enter Flight Commands:
BA009 0 0 150 1
BA009: Roger
OK
BA009 (-12.5,0,2925) (3,-0,-350) (0,0,0) fuel: 76
QF040 (-49.76,0.13,11000) (3.01,-0.05,-600) (0,0,0) fuel:
99
NZ164 (-70,-40.65,20000) (1,5.3,-500) (0,0,0) fuel: 69
CO1102 (111,-86,32000) (-5,1,-500) (0,0,0) fuel: 72
Enter Flight Commands:
OK
BA009 (-9.5,0,2575) (3,-0,-350) (0,0,0) fuel: 75
QF040 (-46.75,0.08,10400) (3.01,-0.05,-600) (0,0,0) fuel:
98
NZ164 (-69,-35.35,19500) (1,5.3,-500) (0,0,0) fuel: 68
CO1102 (106,-85,31500) (-5,1,-500) (0,0,0) fuel: 71
BA009 transferred to airport traffic control, bye!
One landed safely, QF040 on course as well
Enter Flight Commands:
```

20.4 & : THE "ADDRESS OF" OPERATOR

The primary reason for having pointers is to hold addresses of data objects that have been dynamically allocated in the heap, i.e. for values returned by the `new` operator.

You need pointers to heap based structures when representing simple data objects with variable lifetimes like the `Aircraft` of the last example. You also need pointers when building up complex data structures that represent networks of different kinds. Networks are used for all sorts of purposes; they can represent electrical circuits, road maps, the structure of a program, kinship relations in families, the structure of a molecule, and thousands of other things. Networks are built up at run-time by using pointers to thread together different dynamically created component parts.

Why would you want pointers to data in the static data segment or to automatics on the stack? After all, if such variables are "in scope" you can use them directly, you don't have to work through pointer intermediaries. You should NEVER incorporate the address of an automatic (stack based) data item in an elaborate network structure. It is rare to want to incorporate the address of a static data item.

Really, you shouldn't be writing code that needs addresses of things other than heap-based objects. Usually you just want the address of an entire heap based object (the value returned by `new`), though sometimes you may need the address of a specific data member within a heap based object.

But C and C++ have the `&` "address of" operator. This allows you to get the address of any data element. Once an address has been obtained it can be used as the value to be stored in a pointer of the appropriate type. If you look at almost any C program, and most C++ programs, you will see `&` being used all over the place to get addresses of automatics and statics. These addresses are assigned to pointers. Pointers are passed as arguments. Functions with pointer arguments have code that has numerous expression using pointer dereferencing (`*ptr`). Why are all these addresses needed?

Reference arguments and pointer arguments for functions

No "pass by reference" in C

For the most part, the addresses are needed because the C language does not support pass by reference. In C, arguments for functions are passed by value. So, simple variables and struct instances are copied onto the stack (ignore arrays for now, they are discussed in the next section). The called function works with a copy of the original data. There are sound reasons for designing a language that way. If you are trying to model the mathematical concept of a function, it is something that simply computes a value. An idealized, mathematical style function should not have side effects; it shouldn't go changing the values of its arguments.

Uses of "pass by reference"

Function arguments that are "passed by reference" have already been used for a number of reasons. Thus, in the example in section 12.8.3, there was a function that had to return a set of values (actually, an `int` and a `double`) rather than a single value. Since a function can only return a single value, it was necessary to have reference arguments. The example function had an integer reference and a double reference as arguments; these allowed the function to change the values of the caller's variables.

Other examples have used pass by reference for structures. In the case of const reference arguments, this was done to avoid unnecessary copying of the struct that would occur if it were passed by value. In other cases, the structs are passed by reference to allow the called function to modify the caller's data.

As previously explained, a compiler generates different code for value and reference arguments. When a function needs a value argument, the compiler generates code that, at the point of call, copies the value onto the stack; within the function, the code using the "value" variable will have addresses that identify it by its position in the current stack frame. Reference arguments are treated differently. At the point of call their addresses are loaded onto the stack. Within the function, the address of a reference argument is taken from the stack and loaded into an address register. When the code needs to access the actual data variable, it uses indirect addressing via this address register; which is the same way that code uses pointers.

*Implementation of
pass by value and
pass by reference*

Consider the `Exchange()` function in the following little program:

```
void Exchange(double& data1, double& data2)
{
    double temp;
    temp = data1;
    data1 = data2;
    data2 = temp;
}

int main()
{
    cout << "Enter two numbers : ";
    double a, b;
    cin >> a >> b;
    if (b < a)
        Exchange(a, b);
    cout << "In ascending order: " << a << ", " << b <<
endl;
    return 0;
}
```

Function `Exchange()` swaps the values in the two variables it is given as arguments; since they are reference variables, the function changes data in the calling environment. The code generated for this program passes the addresses of variables `a` and `b`. The value in `a` gets copied into `temp`, replaced by the value in `b`, then the value of `temp` is stored in `b`. These data movements are achieved using indirect addressing through address registers loaded with the address of `a` and `b`.

Pass by reference really involves working with addresses and pointers. But the compiler takes care of the details. The compiler arranges to get the addresses of the arguments at the point of call. The compiler generates code that uses indirection, pointer style, for the body of the function.

Programs need the "pass by reference" mechanism. The C language didn't have it. But C allowed programmers to hand code the mechanism everywhere it was needed.

*Compiler uses
addresses and
pointers for pass by
reference*

*Programmer uses
addresses and
pointers*

*Faking pass by
reference using
explicit addresses and
pointers*

C has pass by value. An address is a value and can be passed as an argument if the function specifies that it wants a pointer as an argument. It is easy to use the & operator to get an address, so at the point of call the addresses can be determined and their values put onto the stack.

The code of the function has to use pointers. So, for example, a function that needs to change a double argument will need a double* pointer (i.e. its argument list will have to include something like double *dptr). When the code of the function needs the value of that double, it will have to access it indirectly via the pointer, i.e. it will have to use the value *dptr.

Using explicit pointers and address, the previous program becomes:

```
void Exchange(double *dptr1, double *dptr2)
{
    double temp;
    temp = *dptr1;
    *dptr1 = *dptr2;
    *dptr2 = temp;
}

int main()
{
    cout << "Enter two numbers : ";
    double a, b;
    cin >> a >> b;
    if(b < a)
        Exchange(&a, &b);
    cout << "In ascending order: " << a << ", " << b <<
endl;
    return 0;
}
```

The function prototype specified pointer arguments, Exchange(double *dptr1, double *dptr2). The call, Exchange(&a, &b), has the appropriate "address of operations" getting the addresses of the actual arguments a and b.

The statement:

```
temp = *dptr1;
```

gets the double value accessed indirectly via dptr1 and saves it in the double temp.
The statement

```
*dptr1 = *dptr2;
```

gets the value accessed indirectly via dptr2 and uses it to overwrite the double in the memory location referenced via dptr1. The final statement changes the value in the location reference by dptr2 to be the value stored temporarily in temp.

High level source code using explicit pointers and addresses corresponds almost directly to the actual machine instruction sequences generated by the compiler. Identical instruction sequences are generated for the high level code that uses references; it is just that the code with references leaves the mechanism implicit. There are programmers who prefer the pointer style, arguing that "it is better to see what is really going on". (Of course, if you follow such arguments to their logical conclusion, all programs should be written in assembly language because in assembler you can see the instructions and really truly know what is going on.)

Explicit pointers and addresses represent the underlying mechanism

There are a few situations where the pointer style is more natural. The low level `read()` and `write()` functions are good examples. These need to be given the address of a block of bytes; the code of the function uses a byte pointer to access the first byte, then the second, and so on as each byte gets transferred to/from file.

Pointer style or reference style?

In most other cases, reference style is easier to read. The reference style leaves the mechanisms implicit, allowing the reader to focus on the data transformations being performed. The use of explicit pointers, and the continual pointer dereferencing that this necessitates, makes it harder to read the code of functions written using the pointer style.

While you can adopt the style you prefer for the functions that you write for yourself, you are often constrained to using a particular style. All the old C libraries still used from C++ will use pointers and will need to be passed addresses. Because of the C heritage, many C++ programmers write code with pointer arguments; consequently, many of the newer C++ libraries use pointers and pointer dereferencing.

Getting the addresses of program elements

You can get the address of any program element. You automatically get the address of any data object you create in the heap (the return value from `new` is the address); in most other cases you need to use the `&` operator. The following contrived code fragment illustrates how addresses of a variety of different data elements can be obtained and used in calls to `write()`. The prototype for function `write()` specifies either a `char*` or a `void*` (depending on the particular version of the `iostream` library used in your IDE); so at the point of call, a value representing an address has to be specified.

```
struct demo { int f1; double f2; };

// Some data in the static data segment
long array[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
demo d1 = { 1, 2.2 };
demo d2 = { 2, 4.4 };

void WriteJunk(ofstream& out)
{
    double auto1 = 3.142;
    demo *demoptrA = new demo;
```

```

    demoptrA->f1 = -7; demoptrA->f2 = 6.8;
    demo *demoptrB = new demo;
    demoptrB->f1 = 11; demoptrB->f2 = 81.8;
    ...
    // The version of iostream and the compiler on your IDE
    // may require an explicit (char*) cast in all these
calls
    // e.g. the first might have to be
    //    out.write((char*) &auto1, sizeof(double));
    out.write(&auto1, sizeof(double));
    out.write(&array[4], sizeof(long));
    out.write(&d1, sizeof(demo));
    out.write(&(d2.f1), sizeof(int));
    out.write(demoptrA, sizeof(demo)); // No & needed!
    out.write(&(demoptrB->f2), sizeof(double));
}

```

As shown in this fragment, you can get the addresses of:

- local, automatic (stack-based) variables (&auto1);
- specifically chosen array elements (automatic or global/filescope) (&array[4]);
- global/filescope variables (&d1);
- a chosen data member of an auto or static struct (&(d2.f1)).
- a chosen data member of a struct accessed by a pointer (&(demoptrB->f2)).

The call

```
out.write(demoptrA, sizeof(demo));
```

doesn't need an & "get address of"; the pointer demoptrA contains the address needed, so it is the value from demoptrA has to be passed as the argument.

Addresses of

functions and arrays

There are two kinds of program element whose address you can get by just using their names. These are functions and arrays. You shouldn't use the & address of operator with these.

The use of function addresses, held in variables of type "pointer to function", is beyond the scope of this book. You will learn about these later.

As explained more in the next section, C (and hence C++) regards an array name as having an address value; in effect, an array name is a pointer to the start of the array. So, if you needed to pass an array to the write() function, your code would be:

```

long Array2[] = { 100, -100, 200, 300, 567 };
...
out.write(Array2, sizeof(Array2));

```

Pointers as results from functions

A function that has a pointer return type will return an address value. This address value will get used after exit from the function. The address is going to be that of some data element; this data element had better still be in existence when the address gets used!

This should be obvious. Surprisingly often, beginners write functions that return addresses of things that will disappear at the time the functions exits. A simple example is:

```
char *GetName()  
{  
    char buff[100];  
    cout << "Enter name";  
    cin.getline(buff, 99, '\n');  
    return buff;  
}
```

buggy code!

This doesn't work. The array `buff` is an automatic, it occupies space on the stack while the function `GetName()` is running but ceases to exist when the function returns.

The compiler will let such code through, but it causes problems at run time. Later attempts to use the pointer will result in access to something on the stack, but that something won't be the character buffer.

A function should only return the address of an object that outlives it; which means an object created in the heap. The following code will work:

```
char *GetName()  
{  
    char buff[100];  
    cout << "Enter name";  
    cin.getline(buff, 99, '\n');  
    char *ptr = new char[strlen(buff) + 1];  
    strcpy(ptr, buff);  
    return ptr  
}
```

20.5 POINTERS AND ARRAYS

When arrays were introduced in Chapter 11, it was noted that C's model for an array is relatively weak. This model, which of course is shared by C++, really regards an array as little more than a contiguous block of memory.

The instruction sequences generated for code that uses an array have the following typical form:

```
load address register with the address of the start of the  
array  
calculate offset for required element  
add offset to address register
```



```
load data value using indirect address via address register
```

which is somewhat similar to the code for using a pointer to get at a simple data element

```
load address register from pointer variable
load data value using indirect address via address register
```

and really very similar to code using a pointer to access a data member of a struct:

```
load address register with the address of the start of the
struct
calculate offset for required data member
add offset to address register
load data value using indirect addressing via address
register
```

If you think about things largely in terms of the instruction sequences generated, you will tend to regard arrays as similar to pointer based structs. Hence you get the idea of the name of the array being a pointer to the start of that array.

Once you start thinking mainly about instructions sequences, you do tend to focus on different issues. For example, suppose that you have an array of characters that you need to process in a loop, you know that if you use the following high level code:

```
char msg[50];
...
for(int i=0; i < len; i++) {
    ... = msg[i];
    ...
}
```

then the instruction sequence for accessing the *i*th character will be something like

```
load an address register with the address of msg[0]
add contents of integer register that holds i to address
register
load character indirectly using address in address register
```

which is a little involved. (Particularly as on the very first machine used to implement C, a PDP-7, there was only one register so doing subscripting involved a lot of shuffling of data between the CPU and memory.)

Still thinking about instructions, you would know that the following would be a more efficient way of representing the entire loop construct:

```
load address register with address of msg[0]
load integer register with 0
loop
    compare contents of integer register and value len
```

```

    jump if greater to end_loop
    ...
    load character indirectly using address in address
register
    add 1 to address register // ready for next time
    ...

```

This instruction sequence, which implements a pointer based style for accessing the array elements, is more "efficient". This approach needs at most two instructions to get a character from the array where the other scheme required at least three instructions. (On many machines, the two operations "load character indirectly using address in address register" and "add 1 to address register" can be combined into a single instruction.)

C was meant to compile to efficient instruction sequences. So language constructs were adapted to make it possible to write C source code that would be easy to compile into the more efficient instruction sequence:

```

char msg[50];
char *mptr;
...
mptr = msg;
for(int i=0; i < len; i++) {
    ...
    ... = *mptr;
    mptr++;
    ...
}

```

This initializes a pointer with the address of the first array element:

```
mptr = msg;
```

(which could also be written as `mptr = &(msg[0])`). When the character is needed, pointer dereferencing is used:

```
... = *mptr;
```

Finally, "pointer arithmetic" is performed so as to make the pointer hold the address of the next character from the array.

C (and C++) have to allow arithmetic operations to be done on pointers. Once you've allowed operations like `++`, you might as well allow other addition and subtraction operations (I don't think anyone has ever found much application for pointer multiplication or division). For example, if you didn't have the `strlen()` function, you could use the following:

```

int StrLen(char *mptr)
{
    char *tmp;
    tmp = mptr;

```

Pointer arithmetic

```

        while( *tmp) tmp++;
        return tmp - mptr;
    }

```

This function would be called with a character array as an argument (e.g. `StrLen("Hello");`). The while loop moves the pointer `tmp` through the array until it is pointing to a null character. The final statement:

```
return tmp - mptr;
```

subtracts the address of the start of the array from the address where the `'\0'` character is located.

Arithmetic operations on pointers became a core part of the language. Many of the C libraries, still used from C++, are written in the expectation that you will be performing pointer arithmetic. For example, the string library contains many functions in addition to `strlen()`, `strcmp()`, and `strcpy()`; one of the other functions is `strchr()`:

```
char *strchr(const char *s, int c)
```

this finds the first occurrence of a character `c` in a string `s`. It returns a pointer to the position where the character occurs (or `NULL` if it isn't there). Usually, you would want to know which array element of the character array contained the character, but instead of an integer index you get a pointer. Of course, you can use pointer arithmetic to get the index:

```

char word[] = "Hello";
char *ptr = strchr(word, 'e');
int pos = ptr - word;
cout << "e occurred at position " << pos << endl;

```

Obviously, character arrays can not be a special case. What works for characters has to work for other data types. This requirement has ramifications. If `++` changes a `char*` pointer so that it refers to the next element of a character array, what should `++` do for an array of doubles?

The following program fragment illustrates the working of `++` with different data types (all pointer values are printed as long integers using decimal output to make things clearer):

```

struct zdemo { int f1; char f2; double f3 ;};

int main()
{
    char cArray[10];
    short sArray[10];
    long lArray[10];
    double dArray[10];
    zdemo zArray[10];

```

```

char *cptr = cArray;
short *sptr = sArray;
long *lptr = lArray;
double *dptr = dArray;
zdemo *zptr = zArray;

for(int i=0; i < 5; i++) {
    cout << long(cptr) << ", " << long(sptr) << ", "
        << long(lptr) << ", " << long(dptr) << ", " <<
            long(zptr) << endl;
    cptr++; sptr++, lptr++, dptr++, zptr++;
}
cout << cptr - cArray << ", ";
cout << sptr - sArray << ", ";
cout << lptr - lArray << ", ";
cout << dptr - dArray << ", ";
cout << zptr - zArray << endl;

return 0;
}

22465448, 22465460, 22465480, 22465520, 22465600
22465449, 22465462, 22465484, 22465528, 22465616
22465450, 22465464, 22465488, 22465536, 22465632
22465451, 22465466, 22465492, 22465544, 22465648
22465452, 22465468, 22465496, 22465552, 22465664
5, 5, 5, 5, 5

```

Output

The arithmetic operations take account of the size of the data type to which the pointer refers. Increment a `char*` changes it by 1 (a `char` occupies one byte), incrementing a `double*` changes it by 8 (on the machine used, a `double` needs 8 bytes). Similarly the value `zptr` (22465680) - `zArray` (22465600) is 5 not 80, because this operation on `zdemo` pointers involves things whose unit size is 16 bytes not one byte.

A lot of C code uses pointer style for all operations on arrays. While there can be advantages in special cases like working through the successive characters in a string, in most cases there isn't much benefit. Generally, the pointer style leads to code that is much less easy to read (try writing a matrix multiply function that avoids the use of the `[]` operator).

Although "pointer style" with pointer dereferencing and pointer arithmetic is a well established style for C programs, it is not a style that you should adopt. You have arrays because you want to capture the concept of an indexable collection of data elements. You lost the benefit of this "indexable collection" abstraction as soon as you start writing code that works with pointers and pointer arithmetic.

20.6 BUILDING NETWORKS

The Air Traffic Controller program represented an extreme case with respect to dynamically created objects – its `Aircraft` were all completely independent, they didn't relate in a specific ways.

There are other programs where the structures created in the heap are components of a larger construct. Once created, the individual separate structures are linked together using pointers. This usage is probably the more typical. Real examples appear in the next chapter with standard data structures like "lists" and "trees".

In future years, you may get to build full scale "Macintosh" or "Windows" applications. In these you will have networks of collaborating objects with an "application" object linked (through pointers) to "document" objects; the documents will have links to "views" (which will have links back to their associated documents), and other links to "windows" that are used to frame the views.

The data manipulated by such programs are also likely to be represented as a network of components joined via pointers. For example, a word processor program will have a data object that has a list of paragraphs, tables, and pictures. Each paragraph will have links to things such as structures that represent fonts, as well as to lists of sentences.

Each of the components in one of these complex structures will be an instance of some struct type or a class. These various structs and classes will have data members that are of pointer types. The overall structure is built up by placing addresses in the pointer data members.

Figure 20.7 illustrates stages in building up a simple kind of list or queue. The overall structure is intended to keep a collection of structures representing individual data items. These structures would be called "list cells" or "list nodes". They would be instances of a struct or class with a form something like the following:

```
struct list_cell {
    data_type1  data_member_1;
    data_type2  data_member_2;
    ...;
    list_cell   *fNext;
};
```

Pointer data member for link Most of the data members would be application specific and are not of interest here. But one data member would be a pointer; its type would be "pointer to list cell". It is these pointer data members, the `fNext` in the example, that are used to link the parts together to form the overall structure.

Head pointer A program using a list will need a pointer to the place where it starts. This will also be a pointer to `list_cell`. It would be a static data segment variable (global or filescope) or might be a data member of yet another more complex structure. For the example in Figure 20.7, it is assumed that this "head pointer" is a global:

```
list_cell *Head;
```

Initially, there would be no `list_cells` and the head pointer would be `NULL` (stage 1 in Figure 20.7). User input would somehow cause the program to create a new `list_cell`:

```
list_cell *MakeCell()
{
```

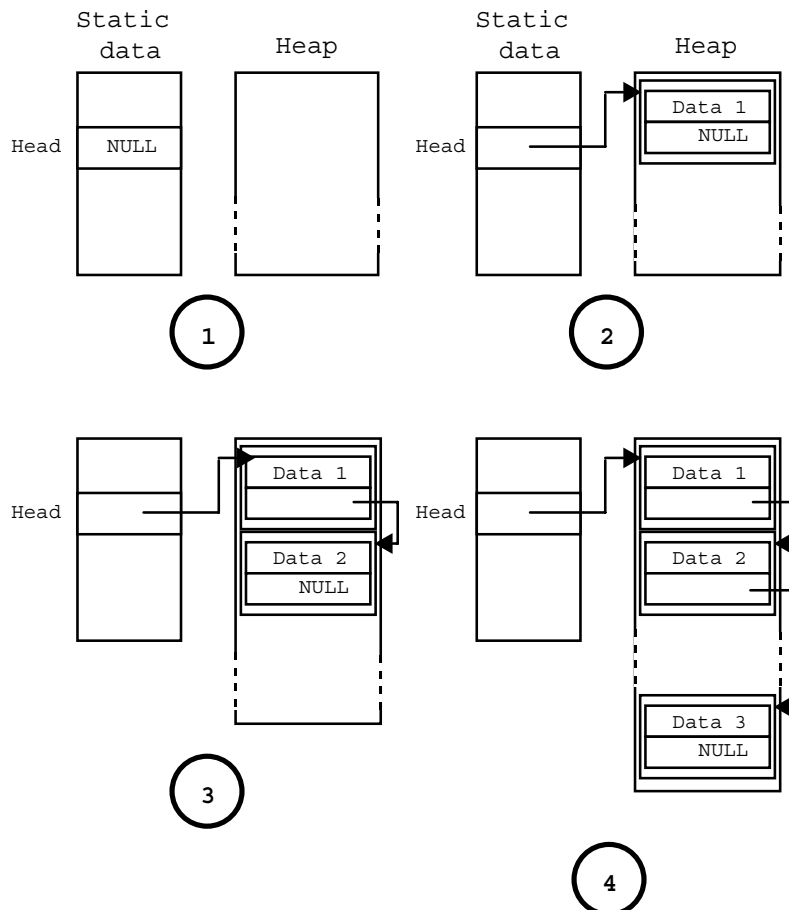


Figure 20.7 Building a "list structure" by threading together "list nodes" in the heap.

```
list_cell *res = new list_cell;
cout << "Enter data for ...";
...;
list_cell->data_member_1 = something;
...
list_cell->fNext = NULL;
}
```

The `MakeCell()` function would obviously fill in all the various specialized data members; the only general one is the link data member `fNext` which has to be initialized to `NULL`.

The function that created the new `list_cell` would add it to the overall structure:

*Setting the Head
pointer to the first
cell*

```
...
list_cell *newdata = MakeCell();
if (Head == NULL)
    Head = newdata;
else ...;
...
```

As this would be the first `list_cell`, the `Head` pointer is changed to point to it. This results in the situation shown as stage 2, in Figure 20.7.

When, subsequently, other `list_cells` get created, the situation is slightly different. The `Head` pointer already points to a `list_cell`. Any new `list_cell` has to be attached at the end of the chain of existing `list_cell(s)` whose start is identified by the `Head` pointer.

*Finding the end of a
list*

The correct place to attach the new `list_cell` will be found using a loop like the following:

```
list_cell *ptr = Head;
while(ptr->fNext != NULL)
    ptr = ptr->fNext;
```

*Attaching a new cell
at the end of the list*

The `list_cell*` variable `ptr` starts pointing to the first `list_cell`. The while loop moves it from `list_cell` to `list_cell` until a `list_cell` with a `NULL fNext` link is found. Such a `list_cell` represents the end of the current list and is the place where the next `list_cell` should be attached. The new `list_cell` is attached by changing the `fNext` link of that end `list_cell`:

```
ptr->fNext = newdata;
```

Stages 3 and 4 shown in Figure 20.7 illustrate the overall structure after the addition of the second and third `list_cells`.

A lot of the code that you will be writing over the next few years will involve building up networks, and chasing along chains of pointers. Two of the examples in the next chapter look at standard cases of simple pointer based composite structures. The first of these is a simple `List` class that represents a slight generalization of the idea of the list as just presented. The other example illustrates a "binary search tree". The code for these examples provides a more detailed view of how such structures are manipulated.

21 Collections of data

As illustrated in the "Air Traffic Controller" example in Chapter 20, arrays can be used to store collections of data. But arrays do have limitations (e.g. you have to specify a maximum size), and they don't provide any useful support function that might help manage a data collection. Over the years, many different structures have been devised for holding collections of data items. These different structures are adapted to meet specific needs; each has associated functionality to help manage and use the collection.

Standard "collection classes"

For example, a simple Queue provides a structure where new data items can be added "at the rear"; the data item "at the front" of the Queue can be removed when it is time for it to be dealt with. You probably studied queues already today while you stood in line to pay at the college cafeteria.

Queue

A "Priority Queue" has different rules. Queued items have associated priorities. When an item is added to the collection, its place is determined by its priority. A real world example is the queue in the casualty department of a busy hospital. Car crash and gun shot patients are priority 1, they go straight to the front of the queue, patients complaining of "heavy" pains in their chests - mild heart attacks – are priority 2 etc.

Priority Queue

"Dynamic Arrays" and "Lists" provide different solutions to the problem of keeping a collection of items where there are no particular access rules like those of Queue. Items can be added at the front or rear of a list, and items can be removed from the middle of lists. Often, lists get searched to find whether they contain a specific item; the search is "linear" (start at the front and keep checking items until either you find the one that is wanted or you get to the end of the list).

Dynamic arrays and lists

A "Binary Tree" is more elaborate. It provides a means for storing a collection of data items that each include "key" values. A binary tree provides a fast search system so you can easily find a specific data element if you know its "key".

Binary tree

Sections 21.1 and 21.2 illustrate a simple Queue and a "Priority Queue". These use fixed size arrays to store (pointers to) data elements. You have to be able to plan for a maximum number of queued items. With the "Dynamic Array", 21.3, things get a bit more complex. A "Dynamic Array" supports "add item", "search for item", and "remove item" operations. Like a simple Queue, the Dynamic Array uses an array of pointers to the items for which it is responsible. If a Dynamic Array gets an "add item"

Data storage

request when its array is full, it arranges to get additional memory. So it starts with an initial array of some default size, but this array can grow (and later shrink) as needed. With Lists, 21.4, you abandon arrays altogether. Instead, a collection is made up using auxiliary data structures ("list cells" or "list nodes"). These list cells hold the pointers to the data items and they can be linked together to form a representation of the list. Binary trees, 21.5, also utilize auxiliary structures ("tree nodes") to build up a representation of an overall data structure.

Data items? What data items get stored in these lists, queues and trees?

Ideally, it should be any data item that a programmer wants to store in them. After all, these Queue, List, and Binary Tree structures represent general concepts (in the preferred jargon, they are "abstract data types"). You want to be able to use code implementing these types. You don't want to have to take outline code and re-edit it for each specific application.

"Generic code" There are at least three ways in C++ in which you can get some degree of generality in your code (code that is independent of the type of data manipulated is "generic"). The examples in this chapter use the crudest of these three approaches. Later examples will illustrate more sophisticated techniques such as the use of (multiple) inheritance and the use of templates. The approach used here is an older, less sophisticated (and slightly more error prone) technique but it is also the simplest to understand.

The data items that are to be in the Queues, Lists, and Dynamic Arrays are going to be structures that have been created in the heap. They will be accessed via pointers. We can simply use `void*` pointers (pointers to data items of unspecified type). Code written to use `void*` pointers is completely independent of the data items accessed via those pointers.

The Priority Queue and Binary Tree both depend on the data items having associated "key values" (the "keys" of the Binary Tree, the priorities of the Priority Queue). These key values will be long integers. The interface for these classes require the user to provide the integer key and a pointer to the associated data object.

21.1 CLASS QUEUE

When on earth would you want to model a "queue" in a computer program?

Well, it is not very often! Most of the places where queues turn up tend to be in the underlying operating system rather than in typical applications programs. Operating systems are full of queues of many varied kinds.

Queues in operating systems

If the operating system is for a shared computer (rather than a personal computer) it will have queues for jobs waiting to run. The computer might have one CPU and one hundred terminals; at any particular time, there may be "requests" for CPU cycles from many of the terminals. The operating system maintains a queue of requests and gets the CPU to process each in turn.

Elsewhere in the operating system, there will be queues of requests for data transfers to and from disk. There will also be queues of characters waiting to be transmitted

down modem lines to remote terminals or other machines, and queues of characters already typed but not yet processed.

Outside of operating systems, queues most frequently turn up in simulations. The real world has an awful lot of queues, and real world objects spend a lot of time standing in queues. Naturally, such queues appear in computer simulations.

What does a queue own? A queue owns "something" that lets it keep pointers to the queued items; these are kept in "first come, first served" order. (The "something" can take different forms.) *A queue owns ...*

A queue can respond to the following requests: *A queue does ...*

- First
Remove the front element from the queue and return it.
- Add (preferred name Append)
Add another element at the rear of the queue.
- Length
Report how many items are queued.
- Full
If the queue only has a finite amount of storage, it may get to be full, a further Add operation would then cause some error. A queue should have a "Full" member function that returns true if the queue is full.
- Empty
Returns true if there are no data elements queued; an error would occur if a First operation was performed on an empty queue.

Several different data structures can be used to represent a queue in a program. This example uses a structure known as a "circular buffer". This version of a queue is most commonly used for tasks such as queuing characters for output devices.

*Representation:
"circular buffer"*

The principal of a circular buffer is illustrated in Figure 21.1. A fixed sized array, the "buffer", is used to store the data values. In this implementation, the array stores `void*` pointers to other data structures that have been allocated in the heap; but if a circular buffer were being used to queue characters going to a modem, the array would hold the actual characters. (In Figure 21.1, the data items are shown as characters). As well as the array, there are two integers used to index into the array; these will be referred to as the "get-index" and the "put-index". Finally, there is a separate integer that maintains a count of the number of items currently stored.

An Append ("put") operation puts a data item at "the end of the queue". Actually the Append operation puts it in the array at the index position defined by `fp` (the put index) and then increments the `fp` index and the counter (`fcount`). So, successive characters fill in array elements 0, 1, 2, In the example shown in Figure 21.1, the array `fdata` has six elements. The `fp` index is incremented modulo the length of the array. So, the next element after `fdata[5]` is `fdata[0]` (this makes the index "circle round" and hence the name "circular buffer").

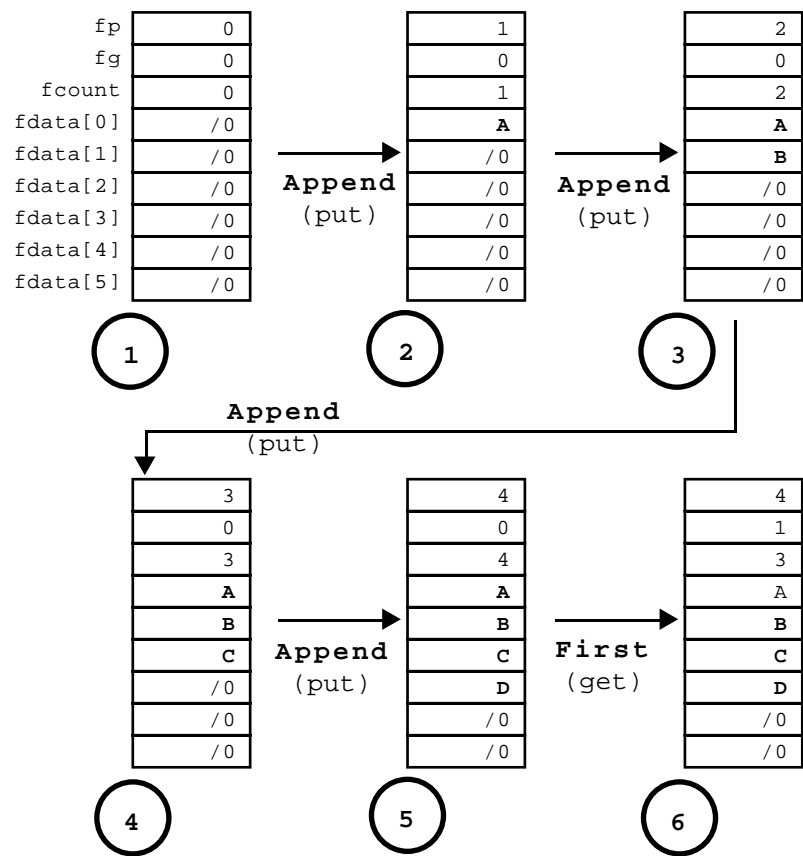


Figure 21.1 Queue represented as a "circular buffer" and illustrative Append (put) and First (get) actions.

The First (get) operation that removes the item from "the front of the queue" works in a similar fashion. It uses the "get index" (`fg`). "First" takes the data item from `fdata[fg]`, updates the value of `fg` (modulo the array length so it too "circles around"), and reduces the count of items stored. (The data value can be cleared from the array, but this is not essential. The data value will be overwritten by some subsequent Append operation.)

Error conditions Obviously there are a couple of problem areas. A program might attempt a First (get) operation on an empty queue, or might attempt to add more data to a queue that is already full. Strictly a program using the queue should check (using the "Empty" and "Full" operations) before performing a First or Append operation. But the code implementing the queue still has to deal with error conditions.

Exceptions When you are programming professionally, you will use the "Exception" mechanism (described in Chapter 26) to deal with such problems. The queue code can "throw an

exception"; this action passes information back to the caller identifying the kind of error that has occurred. But simpler mechanisms can be used for these introductory examples; for example, the program can be terminated with an error message if an "Append" operation is performed on a full queue.

The queue will be represented by a class. The declaration for this class (in a header file Q.h) specifies how this queue can be used: *Class declaration*

```
#ifndef __MYQ__
#define __MYQ__

#define QSIZE 6

class Queue {
public:
    Queue();

    void    Append(void* newitem);
    void    *First(void);

    int     Length(void) const;
    int     Full(void) const;
    int     Empty(void) const;

private:
    void    *fdata[QSIZE];
    int     fp;
    int     fg;
    int     fcount;
};

inline int Queue::Length(void) const { return fcount; }
inline int Queue::Full(void) const { return fcount == QSIZE; }
inline int Queue::Empty(void) const { return fcount == 0; }
#endif
```

The public interface specifies a constructor (initializor), the two main operations (*public* Append() and First()) and the three functions that merely ask questions about the state of the queue.

The private part of the class declaration defines the data structures used to represent a queue. A program can have any number of Queue objects; each has its own array, count, and array indices. *private*

The three member functions that ask about the queue status are all simple, and are good candidates for being inline functions. Their definitions belong in the header file with the rest of the class declaration. These are const functions; they don't change the state of the queue. *inline functions*

The other member functions would be defined in a separate Q.cp file:

```
#include <iostream.h>
```

```

#include <stdlib.h>
#include "Q.h"

Handling errors void QueueStuffed()
{
    cout << "Queue structure corrupted" << endl;
    cout << "Read instructions next time" << endl;
    exit(1);
}

```

The `QueueStuffed()` function deals with cases where a program tries to put too many data items in the queue or tries to take things from an empty queue. (Error messages like these should really be sent to `cerr` rather than `cout`; but on many IDEs, `cout` and `cerr` are the same thing.)

```

Constructor Queue::Queue()
{
    fp = fg = fcount = 0;
}

```

The constructor has to zero the count and the pointers. There is no need to clear the array; any random bits there will get overwritten by "append" operations.

```

Append void Queue::Append(void* newitem)
{
    if(Full())
        QueueStuffed();
    fdata[fp] = newitem;
    fp++;
    if(fp == QSIZE)
        fp = 0;
    fcount++;
    return;
}

```

The `Append()` function first checks that the operation is legal, using the `QueueStuffed()` function to terminate execution if the queue is full. If there is an empty slot in the array, it gets filled in with the address of the data item that is being queued. Then the `fcount` counter is incremented and the `fp` index is incremented modulo the length of the array. The code incrementing `fp` could have been written:

```

fp++;
fp = fp % QSIZE;

```

The code with the `if()` is slightly more efficient because it avoids the divide operation needed for the modulo (%) operator (divides are relatively slow instructions).

```

void *Queue::First(void)

```

```

{
    if(Empty())
        QueueStuffed();
    void* temp = fdata[fg];
    fg++;
    if(fg == QSIZE)
        fg = 0;
    fcount--;
    return temp;
}

```

The First() (get) operation is very similar.

Test program

Naturally, a small test program must be provided along with a class. As explained in the context of class `Bitmap`, the test program is part of the package. You develop a class for other programmers to use. They may need to make changes or extensions. They need to be able to retest the code. You, as a class implementor, are responsible for providing this testing code.

The test program is part of the package!

These test programs need not be anything substantial. You just have to use all the member functions of the class, and have some simple inputs and outputs that make it easy to check what is going on. Usually, these test programs are interactive. The tester is given a repertoire of commands; each command invokes one of the member functions of the class under test. Here we need "add", "get", and "length" commands; these commands result in calls to the corresponding member functions. The implementation of "add" and "get" can exercise the `Full()` and `Empty()` functions.

We need data that can be put into the queue, and then later get removed from the queue so that they can be checked. Often, it is necessary to have the tester enter data; but here we can make the test program generate the data objects automatically.

The actual test program creates "Job" objects (the program defines a tiny class `Job`) and adds these to, then removes these from the queue.

```

#include <stdlib.h>
#include <iostream.h>
#include "Q.h"

class Job {
public:
    Job(int num, char sym);
    void PrintOn(ostream& out);
private:
    int fN;
    char fC;
};

```

Include the standard header

Class Job: just for testing the queue

```

Job::Job(int num, char sym)
{
    fN = num; fC = sym;
}

void Job::PrintOn(ostream& out)
{
    out << "Job #" << fN << ", activity " << fC << endl;
}

The test program
int main()
{
    int n = 0;
    Job *j = NULL;
    Job *current = NULL;
Make a Queue object
    Queue theQ;

    Loop until a quit
command entered
    for(int done = 0; !done ; ) {
        char command;
        cout << ">";
        cin >> command;
        switch(command) {
Test Length()
            case 'q':
                done = 1;
                break;
            case 'l' :
                cout << "Queue length now " <<
                    theQ.Length() << endl;
                break;
Test Full() and
Append()
            case 'a' :
                j = new Job(++n, (rand() % 25) + 'A');
                cout << "Made new job "; j->PrintOn(cout);
                if(theQ.Full()) {
                    cout << "But couldn't add it to queue"
                        " so got rid of it";
                    cout << endl;
                    delete j;
                }
                else {
                    theQ.Append(j);
                    cout << "Queued" << endl;
                }
                j = NULL;
                break;
Test Empty() and
First()
            case 'g':
                if(theQ.Empty())
                    cout << "Silly, the queue is empty"
                        << endl;
                else {
                    if(current != NULL)
                        delete current;
                    current = (Job*) theQ.First();
                    cout << "Got ";

```



```

                                current->PrintOn(cout);
                                }
                                break;
case 'c':
    if(current != NULL) {
        cout << "Current job is ";
        current->PrintOn(cout);
    }
    else cout << "No current job" << endl;
    break;
case '?':
    cout << "Commands are:" << endl;
    cout << "\tq Quit\n\ta Add to queue\t" << endl;
    cout << "\tc show Current job\n"
        "\tg Get job at front of queue" << endl;
    cout << "\tl Length of queue\n" << endl;
    break;
default:
    ;
    }
    }
    return EXIT_SUCCESS;
}

```

A quick test produced the following:

```

>a
Made new job Job #1, activity T
Queued
>a
Made new job Job #2, activity N
Queued
>c
No current job
>g
Got Job #1, activity T
>l
Queue length now 1
>a
Made new job Job #3, activity G
Queued

```

Recording of test run

21.2 CLASS PRIORITYQUEUE

Priority queues are almost as rare as simple queues. They turn up in similar applications – simulations, low-level operating system's code etc. Fortuitously, they have some secondary uses. The priority queue structure can be used as the basis of a sorting mechanism that is quite efficient; it is a reasonable alternative to the Quicksort

function discussed in Chapter 13. Priority queues also appear in the implementation of some "graph" algorithms such as one that finds the shortest path between two points in a network (such graph algorithms are outside the scope of this text, you may meet them in more advanced courses on data structures and algorithms).

As suggested in the introduction to this chapter, a priority queue would almost certainly get used in a simulation of a hospital's casualty department. Such simulations are often done. They allow administrators to try "What if ...?" experiments; e.g. "What if we changed the staffing so there is only one doctor from 8am to 2pm, and two doctors from 2pm till 9pm?". A simulation program would be set up to represent this situation. Then a component of the program would generate "incoming patients" with different problems and priorities. The simulation would model their movements through casualty from admission through queues, until treatment. These "incoming patients" could be based on data in the hospital's actual records. The simulation would show how long the queues grew and so help determine whether a particular administrative policy is appropriate. (The example in Chapter 27 is a vaguely similar simulation, though not one that needs a priority queue.)

The objects being queued are again going to be "jobs", but they differ slightly from the last example. These jobs are defined by a structure:

```
struct Job {
    long    prio;
    char    name[30];
};
```

(this simple struct is just to illustrate the working of the program, the real thing would be much larger with many more data fields). The `prio` data member represents the priority; the code here will use small numbers to indicate higher priority.

A over simple priority queue!

You can implement a very simple form of priority queue. For example:

```
class TrivialPQ {
public:
    TrivialPQ();
    Job    First();
    void    Insert(const Job& j);
private:
    Job    fJobs[kMAXSIZE];
    int    fcount;
};
```

A `TrivialPQ` uses an array to store the queued jobs. The entries in this array are kept ordered so that the highest priority job is in element 0 of the array.

If the entries are to be kept ordered, the `Insert()` function has to find the right place for a new entry and move all less urgent jobs out the way:

```
void TrivialPQ::Insert(const Job& j)
{
```

```

    int    pos = 0;
    while((pos < fcount) && (fJobs[pos].prio < j.prio)) pos++;
    // j should go at pos
    // less urgent jobs move up to make room
    for(int i = fcount; i > pos; i--)
        fJobs[i] = fJobs[i-1];
    fJobs[pos] = j;
    fcount++;
}

```

Similarly, the `First()` function can shuffle all the lower priority items up one slot after the top priority item has been removed:

```

Job TrivialPQ::First()
{
    Job j = fJobs[0];
    for(int i = 1; i < fcount; i++)
        fJobs[i-1] = fJobs[i];
    fcount--;
    return j;
}

```

The class `TrivialPQ` does define a workable implementation of a priority queue. *Cost is $O(N)$* The trouble is the code is rather inefficient. The cost is $O(N)$ where N is the number of jobs queued (i.e. the cost of operations like `First()` and `Insert()` is directly proportional to N). This is obvious by inspection of `First()`; it has to move the $N-1$ remaining items down one slot when the front element is removed. On average, a new Job being inserted will go somewhere in the middle. This means the code does about $N/2$ comparisons while finding the place, then $N/2$ move operations to move the others out of the way.

You could obviously reduce the cost of searching to find the right place. Since Jobs in the `fJobs` array are in order, a binary search mechanism could be used. (Do you still remember binary search from Chapter 13?) This would reduce the search costs to $O(\lg N)$ but the data shuffling steps would still be $O(N)$.

If you could make the data shuffling steps as efficient as the search, then the overall cost would be introduced to $\lg N$. It doesn't make that much difference if the queues are short and the frequency of calls is low (when $N \approx 10$, then $O(\lg N)/O(N) \approx 3/10$.) If the queues are long, or operations on the queue are very frequent, then changing to a $O(\lg N)$ algorithm gets to be important (when $N \approx 1000$, then $O(\lg N)/O(N) \approx 10/1000$.)

There is an algorithm that makes both searching and data shuffling operations have costs that are $O(\lg N)$. The array elements, the Jobs, are kept "partially ordered". The highest priority job will be in the first array element; the next two most urgent jobs will be in the second and third locations, but they won't necessarily be in order. When the most urgent job is removed, the next most urgent pops up from the second or the third location; in turn, it is replaced by some other less urgent job. Once all the changes have been made, the partial ordering condition will still hold. The most urgent of the

An alternative $O(\lg N)$ algorithm

remaining jobs will be in the first array element, with the next two most urgent jobs in the successive locations.

The algorithm is quite smart in that it manages to keep this partial ordering of the data with a minimum number of comparison and data shuffling steps. Figures 21.2 illustrates the model used to keep the data "partially ordered".

Trees, roots, and tree-nodes

You should imagine the data values placed in storage elements that are arranged in a branching "tree"-like manner. Like most of the "trees" that appear in illustrations of data structures, this tree "grows" downwards. The topmost "node" (branch point) is the "root". It contains the data value with the most urgent priority (smallest "prio" value).

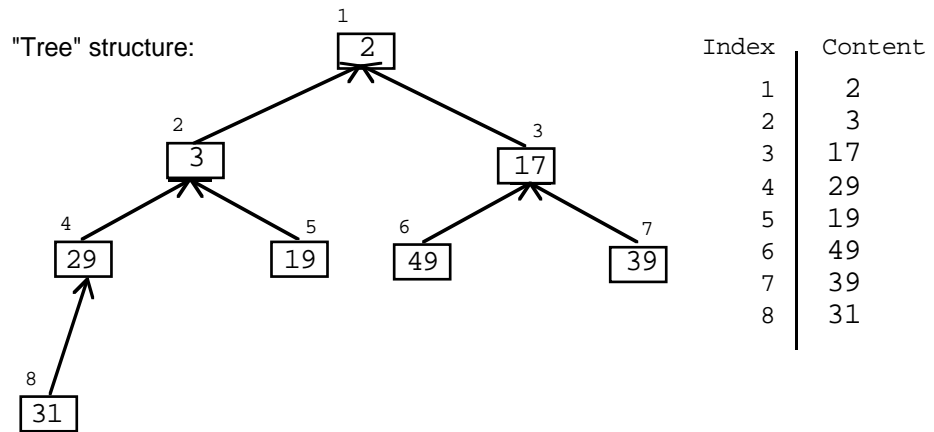


Figure 21.2 "Tree" of partially ordered values and their mapping into an array.

Child nodes and leaves

Each "node" in the tree can have up to two nodes in the level below it. (These are referred to as its "child nodes"; the terminology is very mixed up!) A node with no children is a "leaf".

At every level, the data value stored in a node is smaller than the data values stored in its children. As shown in Figure 21.2, you have the first node holding the value 2, the second and third nodes hold the values 3 and 17. The second node, the one with the 3 has children that hold the values 29 and 19 and so on.

This particular tree isn't allowed to grow arbitrarily. The fourth and fifth nodes added to the tree become children of the second node; the sixth and seventh additions are children of the third node. The eight to fifteenth nodes inclusive form the next level of this tree; with the 8th and 9th "children" of node 4 and so on.

"Tree" structure can be stored in an array

Most of the "trees" that you will meet as data structures are built by allocating separating node structs (using the new operator) and linking them with pointers. This particular form of tree is exceptional. The restrictions on its shape mean that it can be mapped onto successive array elements, as also suggested in Figure 21.2

The arrangement of data values in successive array entries is:

(#1, 2) (#2, 3) (#3, 17) (#4, 29) (#5, 19) (#6, 49) (#7, 39)
 (#8, 31)

The values are "partially sorted", the small (high priority) items are near the top of the array while the large values are at the bottom. But the values are certainly not in sorted order (e.g. 19 comes after 29).

The trick about this arrangement is that when a new data value is added at the end, it doesn't take very much work to rearrange data values to restore the partial ordering. This is illustrated in Figure 21.3 and 21.4.

Figure 21.3 illustrates the situation immediately after a new data value, with priority 5, has been added to the collection. The collection is no longer partially ordered. The data value in node 4, 29, is no longer smaller than the values held by its children.

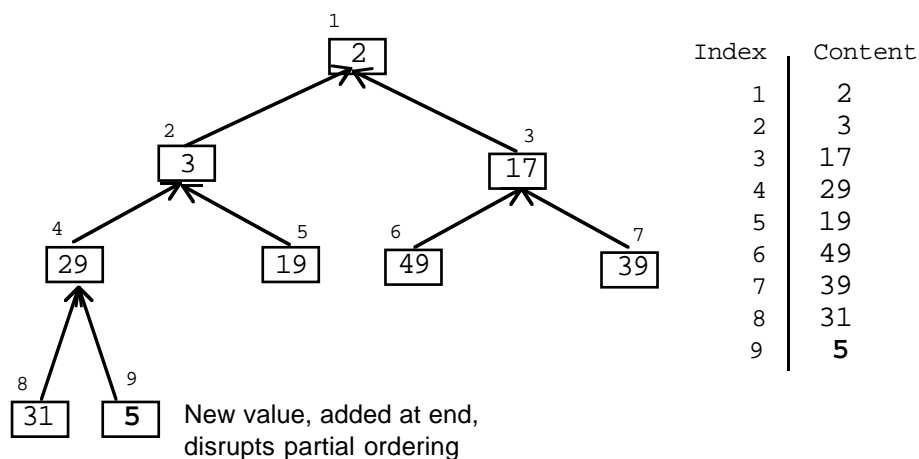


Figure 21.3 Addition of data element disrupts the partial ordering.

The partial ordering condition can be restored by letting the new data value work its way up the branches that link its starting position to the root. If the new value is smaller than the value in its "parent node", the two values are exchanged.

In this case, the new value in node 9 is compared with the value in its parent node, node 4. As the value 5 is smaller than 29, they swap. The process continues. The value 5 in node 4 is compared with the value in its parent node (node 2). Here the comparison is between 3 and 5 and the smaller value is already in the lower node, so no changes are necessary and the checking procedure terminates. Figure 21.4 illustrates the tree and array with the partial ordering reestablished.

Note that the newly added value doesn't get compared with all the other stored data values. It only gets compared with those along its "path to root".

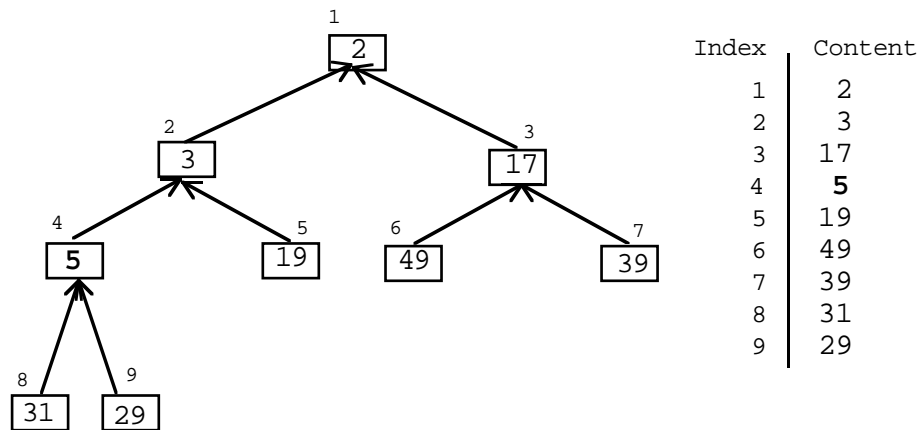


Figure 21.4 Restoration of the partial ordering.

If the newly added value was smaller than any already in the storage structure, e.g. value 1, it would work its way all the way up until it occupied the root-node in array element 1. In this "worst case", the number of comparison and swap steps would then be the same as the number of links from the root to the leaf.

O(lgN) algorithm Using arguments similar to those presented in Chapter 13 (when discussing binary search and Quicksort), you can show that the maximum number of links from root to leaf will be proportional to lgN where N is the number of data values stored in the structure. So, the Insert operation has a O(lgN) time behaviour.

Removing the top priority item When the "first" (remove) operation is performed, the top priority item gets removed. One of the other stored data values has to replace the data value removed from the root node. As illustrated in Figure 21.5; the restoration scheme starts by "promoting" the data value from the last occupied node.

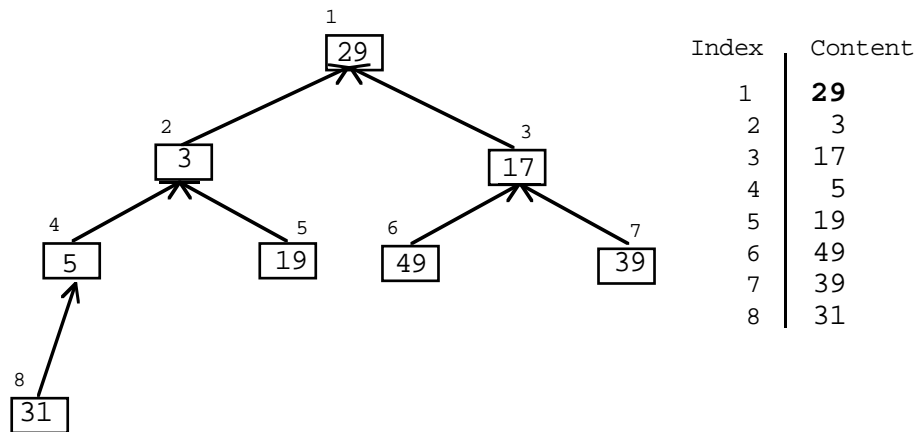
Of course, this destroys the partial ordering. The value 29 now in the root node (array element 1) is no longer smaller than the values in both its children.

Tidying up after removing the first element So, once again a process of rearrangement takes place. Starting at the root node, the data value in a node is compared with those in its children. The data values in parent node and a child node are switched as needed. So, in this case, the value 29 in node 1 switches places with the 3 in node 2. The process is then repeated. The value 29 now in node 2 is switched for the value 5 in node 4. The switching process can then terminate because partial ordering has again been restored.

As in the case of insert, this tidying up after a removal only compares values down from root to leaf. Again its cost is O(lgN).

Although a little more involved than the algorithms for TrivialPQ, the functions needed are not particularly complex. The improved O(lgN) (as opposed to O(N)) performance makes it worth putting in the extra coding effort.

Highest priority item removed (2)



Partial ordering restored

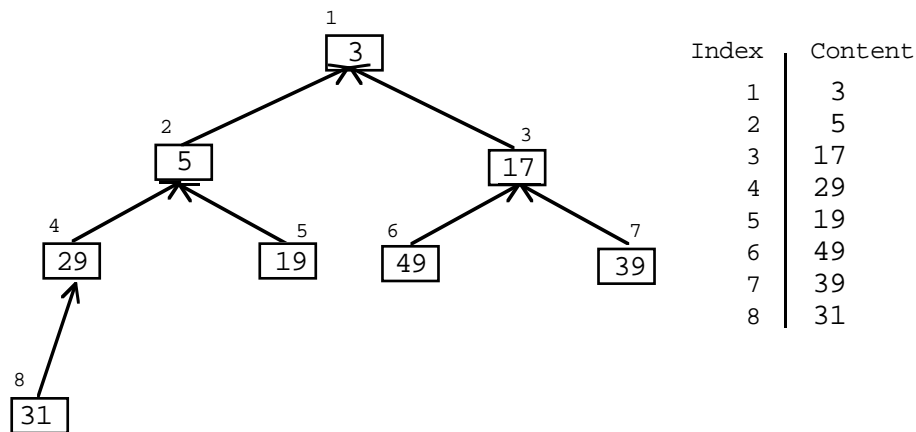


Figure 21.5 Removal of top priority item followed by restoration of partial ordering.

Values get entered in random order, but they will be removed in increasing order. *"Heapsort"*
 You can see that the priority queue could be used to sort data. You load up the priority queue with all the data elements (using for "priorities" the key values on which you want the data elements sorted). Once they are all loaded, you keep removing the first element in the priority queue until it becomes empty. The values come out sorted.

This particular sorting algorithm is called "heapsort". Its efficiency is $O(N \lg N)$ (roughly, you are doing $O(\lg N)$ insert and remove operations for each of N data elements). Heapsort and Quicksort have similar performances; different degrees of

ordering in the initial data may slightly favour one algorithm relative to the other. Of course, heapsort has the extra space overhead of the array used to represent the queue.

The name "heap" in heapsort is somewhat unfortunate. It has nothing to do with "the Heap" used for free storage allocation.

Design and implementation of class PriorityQ

<i>A priority queue owns ...</i>	What does a priority queue own? This priority queue owns an array in which it keeps pointers to the queued items along with details of their priorities; there will also be an integer data member whose value defines the number of items queued.
<i>A priority queue does ...</i>	<p>A priority queue can respond to the following requests:</p> <ul style="list-style-type: none"> • First Remove the front element from the queue and return it. • Add (preferred name Insert) Insert another item into the queue at a position determined by its priority • Length Report how many items are queued. • Full As the queue only has a finite amount of storage, it may get to be full, a further Insert operation would then cause some error. The queue has to have a "Full" member function which returns true it is full. • Empty Returns true if there are no data elements queued; an error would occur if a First operation was performed on an empty queue.
<i>and supplementary debugging functions</i>	In cases like this where the algorithms are getting a little more complex, it often helps to have some extra functions to support debugging. When building any kind of tree or graph, you will find it useful to have some form of "print" function that can provide structural information. As this is not part of the interface that would normally be needed by clients, it should be implemented as conditionally compiled code.
<i>Auxiliary private member functions</i>	The <code>Insert()</code> and <code>First()</code> member functions make initial changes to the data in the array, then the "partial ordering" has to be reestablished. With <code>Insert()</code> , a data element gets added at the bottom of the array and has to find its way up through the conceptual tree structure until it gets in place. With <code>First()</code> a data element gets put in at the root and has to drop down until it is in place. These rearrangements are best handled by auxiliary private member functions (they could be done in <code>Insert()</code> and <code>First()</code> but it is always better to split things up into smaller functions that are easier to analyze and understand).
<i>A struct to store details of queued objects and priorities</i>	The storage array used by a <code>PriorityQ</code> will be an array of simple structs that incorporate a <code>void*</code> pointer to the queued object and a long integer priority. This can mean that the priority value gets duplicated (because it may also occur as an actual data member within the queued object). Often though, the priority really is something that

is only meaningful while the object is associated with the priority queue; after all, casualty patients cease to have priorities if they get to be admitted as ward patients.

The type of struct used to store these data is only used within the priority queue code. Since nothing else in the program need know about these structs, the struct declaration can be hidden inside the declaration of class `PriorityQ`.

The "pq.h" header file with the class declaration is:

```
#ifndef __MYPQ__
#define __MYPQ__

#define DEBUG

#define k_PQ_SIZE 50
class PriorityQueue {
public:
    PriorityQueue();

    void    Insert(void* newitem, long priority);
    int     Length(void) const;
    int     Full(void) const;
    int     Empty(void) const;

    void    *First(void);
#ifdef DEBUG
    void    PrintOn(ostream& out) const;
#endif
private:
    void    TidyUp(void);
    void    TidyDown(void);

    struct keyeddata { long fK; void *fd; };
    keyeddata    fQueue[k_PQ_SIZE+1];
    int          fcount;
};

inline int PriorityQueue::Length(void) const { return fcount; }
inline int PriorityQueue::Full(void) const
{ return fcount == k_PQ_SIZE; }
inline int PriorityQueue::Empty(void) const { return fcount == 0; }

#endif
```

Class declaration

Here, the token `DEBUG` is defined so the `PrintOn()` function will be included in the generated code. (Coding is slightly easier if element zero of the array is unused; to allow for this, the array size is `k_PQ_SIZE+1`.)

As in the case of class `Queue`, the simple member functions like `Length()` can be defined as "inlines" and included in the header file.

The constructor has to zero out the count; it isn't really necessary to clear out the array but this is done anyway: *Constructor*

```

PriorityQ::PriorityQ()
{
    for(int i=0; i<= k_PQ_SIZE; i++) {
        fQueue[i].fd = NULL;
        fQueue[i].fK = 0;
    }
    fcount = 0;
}

```

The `Insert()` function increments the count and adds the new queued item by filling in the next element in the `fQueue` array. Function `TidyUp()` is then called to restore partial ordering. `First()` works in an analogous way using `TidyDown()`. Note that neither includes any checks on the validity of the operations; it would be wiser to include `Full()` and `Empty()` checks with calls to an error function as was done for class `Queue`.

```

Insert    void PriorityQ::Insert(void* newitem, long priority)
            {
                fcount++;
                fQueue[fcount].fd = newitem;
                fQueue[fcount].fK = priority;
                TidyUp();
            }

```

```

First    void *PriorityQ::First(void)
            {
                void *chosen = fQueue[1].fd;
                fQueue[1] = fQueue[fcount];
                fcount--;
                TidyDown();
                return chosen;
            }

```

The `PrintOn()` routine has to be in "conditionally compiled" brackets. It simply runs through the array identifying the priority of the data object in each position. This sort of information can help when checking out the implementation:

```

#ifdef DEBUG
void PriorityQ::PrintOn(ostream& out) const
{
    if(fcount == 0) {
        cout << "Queue is empty" << endl;
        return;
    }

    cout << "Position      Item Priority" << endl;
    for(int i = 1; i<= fcount; i++)
        cout << i << "          " << fQueue[i].fK << endl;
}

```

```

}
#endif

```

The `TidyUp()` function implements that chase up through the "branches of the tree" comparing the value of the data item in a node with the value of the item in its parent node. The loop stops if a data item gets moved up to the first node, or if the parent has a data item with a lower `fK` value (higher priority value). Note that you find the array index of a node's parent by halving its own index; e.g. the node at array element 9 is at $9/2$ or 4 (integer division) which is as shown in the Figures 21.3 etc.

```

void PriorityQueue::TidyUp(void)
{
    int          k = fcount;
    keyeddata    v = fQueue[k];
    for(;;) {
        if(k==1) break;
        int nk = k / 2;
        if(fQueue[nk].fK < v.fK) break;
        fQueue[k] = fQueue[nk];
        k = nk;
    }
    fQueue[k] = v;
}

```

*Restoring partial
ordering*

The code for `TidyDown()` is generally similar, but it has to check both children (assuming that there are two child nodes) because either one of them could hold a data item with a priority value smaller than that associated with the item "dropping down" through levels of the "tree". (The code picks the child with the data item with the smaller priority key and pushes the descending item down that branch.)

```

void PriorityQueue::TidyDown(void)
{
    int          Nelems = fcount;
    int          k = 1;
    keyeddata    v = fQueue[k];
    for(;;) {
        int nk;
        if(k > (Nelems / 2)) break;
        nk = k + k;
        if((nk <= Nelems-1) &&
           (fQueue[nk].fK > fQueue[nk+1].fK)) nk++;

        if(v.fK <= fQueue[nk].fK) break;
        fQueue[k] = fQueue[nk];
        k = nk;
    }
    fQueue[k] = v;
}

```

Test program

Once again, a small test program must be provided along with the class. This test program first checks out whether "heapsort" works with this implementation of the priority queue. The queue is loaded up with some "random data" entered by the user, then the data values are removed one by one to check that they come out in order. The second part of the test is similar to the test program for the ordinary queue. It uses an interactive routine with the user entering commands that add things and remove things from the queue.

The test program starts by including appropriate header files and defining a `Job` struct; the priority queue will store `Jobs` (for this test, a `Job` has just a single character array data member):

```
#include <stdlib.h>
#include <iostream.h>
#include "pq.h"

struct Job {
    char    name[30];
};

Job* GetJob()
{
    cout << "job name> ";
    Job *j = new Job;
    cin >> j->name;
    return j;
}

main()
{
    PriorityQ    thePQ;
    int    i;

    /*
    Code doing a "heapsort"
    */
    cout << "enter data for heap sort test; -ve prio to end data"
          << endl;
    for(i = 0; i++)
    {
        int prio;
        cout << "job priority ";
        cin >> prio;
        if(prio < 0)
            break;
        Job* j = GetJob();
        thePQ.Insert(j, prio);
    }
}
```

**Load the queue with
some data**

```

        if(thePQ.Full())
            break;
    }

    while(!thePQ.Empty()) {
        Job *j = (Job*) thePQ.First();
        cout << j->name << endl;
        delete j;
    }

    for(int done = 0; !done ; ) {
        Job *j;
        char ch;
        cout << ">";
        cin >> ch;

        switch(ch) {
        case 'q': done = 1;
                  break;
        case 'l' : cout << "Queue length now " << thePQ.Length()
                    << endl;
                  break;
        case 'a':
            if(thePQ.Full()) cout << "Queue is full!" << endl;
            else {
                int prio;
                cout << "priority "; cin >> prio;
                j = GetJob();
                thePQ.Insert(j,prio);
            }
            break;
        case 's':
            thePQ.PrintOn(cout);
            break;
        case 'g':
            if(thePQ.Empty()) cout << "Its empty!" << endl;
            else {
                j = (Job*) thePQ.First();
                cout << "Removed " << j->name << endl;
                delete j;
            }
            break;
        case '?':
            cout << "Commands are:" << endl;
            cout << "\tq Quit\n\ta Add to queue\t" << endl;
            cout << "\ts show queue status\n\tg"
                << "Get job at front of queue" << endl;
            cout << "\tl Length of queue\n" << endl;
            break;
        default:
            ;
        }
    }
}

```

Pull the data off the queue, hope it comes out sorted

Interactive part adding, removing items etc

Type case from void to Job**

```

    }

    return 0;
}

```

Note the `(Job*)` type casts. We know that we put pointers to `Jobs` into this priority queue but when they come back they are `void*` pointers. The type cast is necessary, you can't do anything with a `void*`. (The type cast is also safe here; nothing else is going to get into this queue). The code carefully deletes `Jobs` as they are removed from the priority queue; but it doesn't clean up completely (there could be `Jobs` still in the queue when the program exits).

Test output

```

...
job priority 5
job name> data5
job priority 107
job name> data107
job priority -1
data5
data11
data17
data35
data55
data92
data107
data108
>a
priority 18
job name> drunk
>a
priority 7
job name> spider-venom
>a
priority 2
job name> pain-in-chest
>a
priority 30
job name> pain-in-a---
>a
priority 1
job name> gunshot
>g
Removed gunshot
>g
Removed pain-in-chest
>g
Removed spider-venom
>g
Removed drunk
>q

```

21.3 CLASS DYNAMICARRAY

It is surprising how rarely dynamic arrays feature in books on data structures. A dynamic array looks after a variable size collection of data items; thus, it performs much the same sort of role as does a List. In many situations, dynamic arrays perform better than lists because they incur less overhead costs.

Both "list" and "dynamic array" allow "client programs" (i.e. programs that use instances of these classes) to:

- add data objects to the collection (as in the rest of this chapter, the data objects are identified by their addresses and these are handled as `void*` pointers);
- ask for the number of objects stored;
- ask whether a particular object is in the collection;
- remove chosen objects from the collection.

In these two kinds of collection there is an implied ordering of the stored data items. There is a "first item", a "second item" and so forth. In the simple versions considered here, this ordering doesn't relate to any attributes of the data items themselves; the ordering is determined solely by the order of addition. As the items are in sequence, these collections also allow client programs to ask for the first, second, ..., n -th item.

Figure 21.6 illustrates a dynamic array. A dynamic array object has a main data block which contains information like the number of items stored, and a separate array of pointers. The addresses of the individual data items stored in the collection are held in this array. This array is always allocated in the heap (or simplicity, the header and trailer information added by the run-time memory manager are not shown in illustrations). The basic data block with the other information may be in the heap, or may be an automatic in the stack, or may be a static variable.

As shown in Figure 21.6, the array of pointers can vary in size. It is this variation in size that makes the array "dynamic".

When a dynamic array is created it gets allocated an array with some default number of entries; the example in Figure 21.6 shows a default array size of ten entries. As data objects get added to the collection, the elements of the array get filled in with data addresses. Eventually, the array will become full.

When it does become full, the `DynamicArray` object arranges for another larger array to be allocated in the heap (using operator `new []`). All the pointers to stored data items are copied from the existing array to the new larger array, and then the existing array is freed (using operator `delete []`). You can vary the amount of growth of the array, for example you could make it increase by 25% each time. The simplest implementations of a dynamic array grow the array by a specified amount, for example the array might start with ten pointers and grow by five each time it gets filled.

"Growing the array"

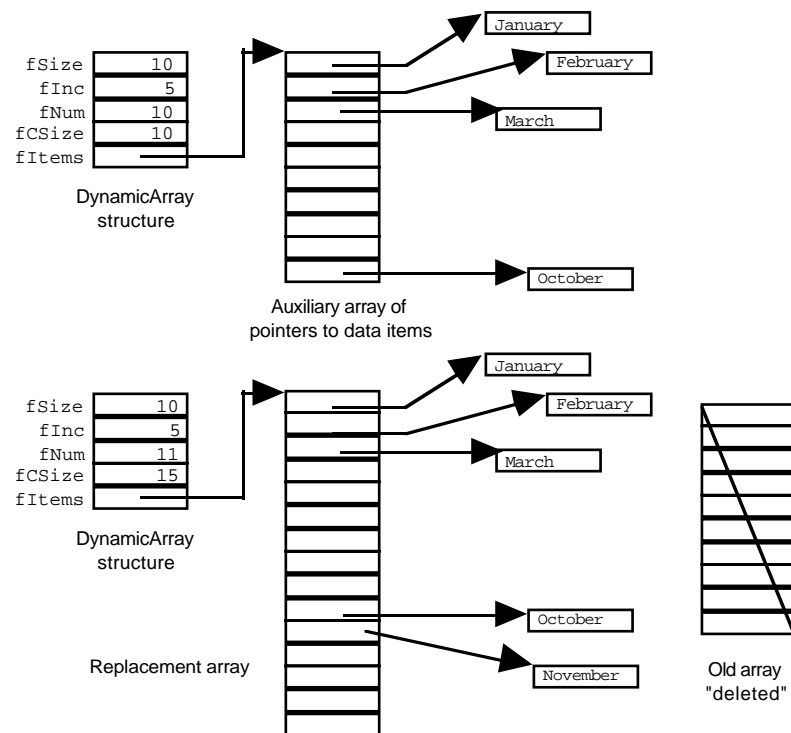


Figure 21.6 A "Dynamic Array".

**Removing items and
"Shrinking" the
array**

Stored items do get removed from the array. It isn't like a queue where only the first item can be removed, a "remove" request will identify the item to be removed. When an item is removed, the "gap" in the array is filled up by moving up the data items from the later array elements. So, if the array had had seven elements, removal of the fifth would result in the pointers in the sixth and seventh elements each moving up one place. If lots of items are removed, it may be worth shrinking the array. "Shrinking" is similar to "growing". The `DynamicArray` object again creates a new array (this time smaller) in the heap and copies existing data into the new array. The old large array is deleted. Typically, the array is only shrunk when a remove operation leaves a significant number of unused elements. The array is never made smaller than the initially allocated size.

**What does a dynamic
array own?**

The main thing a `DynamicArray` object owns is the array of pointers to the data items. In addition, it needs to have integer data members that specify: initial size (this is also the minimum size), the size of the increment, the current array size, and the number of data elements currently stored.

**A `DynamicArray`
does ...**

You can vary the behaviours of `DynamicArrays` and `Lists` a little. These classes have to have functions to support at least some version of the following behaviours:

- **Length**
Report how many items are stored in the collection.
- **Add** (preferred name **Append**)
Add another item to the collection, placing it "after" all existing items.
- **Position**
Find where an item is in the collection (returning its sequence number); this function returns an error indicator if the item is not present. (This function may be replaced by a **Member** function that simply returns a true or false indication of whether an item is present).
- **Nth**
Get the item at a specified position in the collection (the item is not removed from the collection by this operation).
- **Remove**
Removes a specified item, or removes the item at a specified position in the collection.

Note that there is neither a "Full" nor an "Empty" function. DynamicArrays and Lists are never full, they can always grow. If a client program needs to check for an "empty" collection, it can just get the length and check whether this is zero.

You have to choose whether the sequence numbers for stored items start with 0 or 1. Starting with 0 is more consistent with C/C++ arrays, but usually a 1-base is more convenient for client applications. So, if there are N elements in the collection, the "Position" function will return values in the range $1 \dots N$ and functions like "Nth", and "Remove" will require arguments in the range $1 \dots N$.

The remove function should return a pointer to the item "removed" from the collection.

If desired, a debugging printout function can be included. This would print details like the current array size and number of elements. It is also possible for the function to print out details of the addresses of stored data items, but this information is rarely helpful.

A declaration for a simple version of the DynamicArray class is:

```
#ifndef __MYDYNAM__
#define __MYDYNAM__

class DynamicArray {
public:
    DynamicArray(int size = 10, int inc = 5);

    int    Length(void) const;
    int    Position(void *item) const;
    void   *Nth(int n) const;

    void   Append(void *item);

    void   *Remove(void *item);
```

*File "D.h" with the
declaration of class
DynamicArray*

Default parameters

```

        void    *Remove(int itempos);

Auxiliary "Grow"   private:
function as private void    Grow(int amount);
member             int     fNum;
                   int     fSize;
                   int     fCSize;
                   int     fInc;
The "pointer to    void    **fItems;
pointer"           };

inline int DynamicArray::Length(void) const { return fNum; }

#endif

```

For a class like this, it is reasonable for the class declaration to provide default initial values for parameters like the initial array size and the increment, so these appear in the declaration for the constructor.

The next group of functions are all const functions; they don't change the contents of a `DynamicArray`. Once again, the `Length()` function is simple and can be included in the header file as an inline function.

Along with the `Append()` function, there are two overloaded versions of `Remove()`; one removes an item at a specified position in the collection, the other finds an item and (if successful) removes it. The `Append()` and `Remove()` functions require auxiliary "grow" and "shrink" functions. Actually, a single `Grow()` function will suffice, it can be called with a positive increment to expand the array or a negative increment to contract the array. The `Grow()` function is obviously private as it is merely an implementation detail.

void ! ?** Most of the data members are simple integers. However, the `fItems` data member that holds the address of the array of pointers is of type `void**`. This seemingly odd type requires a little thought. Remember, a pointer like `char*` can hold the address of a `char` variable or the address of the start of an array of `chars`; similarly an `int*` holds the address of an `int` variable or the address of an array of `ints`. A `void*` holds an address of something; it could be interpreted as being the address of the start of "an array of `voids`" but you can't have such a thing (while there are `void*` variables you can't have `void` variables). Here we need the address of the start of an array of `void*` variables (or a single `void*` variable if the array size is 1). Since `fItems` holds the address of a `void*` variable, its own type has to be `void**` (pointer to a pointer to something).

The code implementing the class member functions would be in a separate "D.cp" file. The constructor fills in the integer data members and allocates the array:

```

#include <stdlib.h>
#include <assert.h>
#include "D.h"

```

```

const int kLARGE = 10000;
const int kINCL = 5000;

DynamicArray::DynamicArray(int size, int inc)
{
    assert((size > 1) && (size < kLARGE));
    assert((inc > 1) && (inc < kINCL));

    fNum = 0;
    fCSize = fSize = size;
    fInc = inc;

    fItems = new void* [size];
}

```

Constructor

Note that the definition does not repeat the default values for the arguments; these only appear in the initial declaration. (The `assert()` checks at the start protect against silly requests to create arrays with -1 or 1000000 elements.) The `new []` operator is used to create the array with the specified number of pointers.

Function `Length()` was defined as an inline. The other two const access functions involve loops or conditional tests and so are less suited to being inline.

```

int DynamicArray::Position(void *item) const
{
    for(int i = 0; i < fNum; i++)
        if(fItems[i] == item) return i+1;
    return 0;
}

void *DynamicArray::Nth(int n) const
{
    if((n < 1) || (n > fNum))
        return NULL;
    n--;
    return fItems[n];
}

```

The class interface defines usage in terms of indices 1...N. Of course, the implementation uses array elements 0...N-1. These functions fix up the differences (e.g. `Nth()` decrements its argument `n` before using it to access the array).

Both these functions have to deal with erroneous argument data. Function `Position()` can return zero to indicate that the requested argument was not in the collection. Function `Nth()` can return `NULL` if the index is out of range. Function `Position()` works by comparing the address of the item with the addresses held in each pointer in the array; if the addresses match, the item is in the array.

The `Append()` function starts by checking whether the current array is full; if it is, a call is made to the auxiliary `Grow()` function to enlarge the array. The new item can

then be added to the array (or the new enlarged array) and the count of stored items gets incremented:

```
void DynamicArray::Append(void *item)
{
    if(fNum == fCSize)
        Grow(fInc);
    fItems[fNum] = item;
    fNum++;
}
```

The `Grow()` function itself is simple. The new array of pointers is created and the record of the array size is updated. The addresses are copied from the pointers in the current array to the pointers in the new array. The old array is freed and the address of the new array is recorded.

"Growing" the array

```
void DynamicArray::Grow(int delta)
{
    int newsize = fCSize + delta;
    void **temp = new void* [newsize];
    fCSize = newsize;
    for(int i=0; i < fNum; i++)
        temp[i] = fItems[i];
    delete [] fItems;
    fItems = temp;
}
```

The `Remove(void *item)` function has to find the data item and then remove it. But function `Position()` already exists to find an item, and `Remove(int)` will remove an item at a known position. The first `Remove()` function can rely on these other functions:

```
void *DynamicArray::Remove(void *item)
{
    int where = Position(item);
    return Remove(where);
}
```

The `Remove(int)` function verifies its argument is in range, returning `NULL` if the element is not in the array. If the required element is present, its address is copied into a temporary variable and then the array entries are "closed up" to remove the gap that would otherwise be left.

The scheme for shrinking the array is fairly conservative. The array is only shrunk if remove operations have left it less than half full and even then it is only shrunk a little.

```
void *DynamicArray::Remove(int itempos)
{

```

```

        if((itempos < 1) || (itempos > fNum))
            return NULL;
        itempos--;
        void *tmp = fItems[itempos];
        for(int i = itempos + 1; i < fNum; i++)
            fItems[i-1] = fItems[i];
        fNum--;

        if((fNum > fSize) && (fNum < (fCSize / 2)))
            Grow(-fInc);
        return tmp;
    }

```

Test program

The test program for this class creates "book" objects and adds them to the collection. For simplicity, "books" are just character arrays allocated in the heap. Rather than force the tester to type in lots of data, the program has predefined data strings that can be used to initialize the "books" that get created.

```

#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include "D.h"

typedef char *Book;
Book BookStore[] = {
    "The C++ Programming Language",
    "Developing C++ Software",
    "Algorithms in C++",
    ...
    "Effective C++",
    "Object Models, Strategies, Patterns, and Applications"
};

int Numbooks = sizeof(BookStore) / sizeof(char*);

Book PickABook(void)
{
    int num;
    cout << "Pick a book, any book (#1 ... "
        << Numbooks << "): " << endl;
    cin >> num;
    while((num < 1) || (num > Numbooks)) {
        cout << "Try again, enter number ";
        cin.clear();
        cin.ignore(100, '\n');
        cin >> num;
    }
}

```

*Predefined data to
reduce amount of
data entry in test
program*

*Function that creates
a "Book" on the
heap*

```

        num--;
        cout << "You picked : " << BookStore[num] << endl;
        Book ptr;
        ptr = new char[strlen(BookStore[num]) + 1];
        strcpy(ptr, BookStore[num]);
        return ptr;
    }

```

Function `PickABook()` allows the tester to simply enter a number; this selects a string from the predefined array and uses this string to initialize a dynamically created `Book`.

Function `GetPos()` is another auxiliary function used to allow the tester to enter the sequence number of an item in the collection:

```

int GetPos(int max)
{
    if(max < 1) {
        cout << "The collection is empty!" << endl;
        return 0;
    }
    cout << "Which item from collection? ";
    int temp;
    cin >> temp;
    while((temp < 1) || (temp > max)) {
        cout << "Position must be in range 1..." <<
            max << endl;
        cin.clear();
        cin.ignore(100, '\n');
        cin >> temp;
    }
    return temp;
}

```

Like most such test programs, this one consists of a loop in which the user is prompted for commands. These commands exercise the different member functions of the class:

```

int main()
{
    Book lastadded = NULL;
    DynamicArray c1;

    for(int done = 0; !done ; ) {
        char command;
        Book ptr;
        int pos;
        cout << ">";
        cin >> command;
        switch(command) {
            case 'q': done = 1; break;

```

```

case 'l' :
    cout << "Collection now contains "
          << cl.Length()
          << " data items" << endl;
    break;
case 'a' :
    ptr = PickABook();
    if(cl.Position(ptr)) {
        // Note this will never happen,
        // see discussion in text
        cout << "You've already got that one!" << endl;
        delete ptr;
    }
    else {
        cl.Append(ptr);
        cout << "Added" << endl;
        lastadded = ptr;
    }
    break;
case 'f':
    pos = GetPos(cl.Length());
    if(pos > 0) {
        ptr = (char*) cl.Nth(pos);
        cout << pos << " : " << ptr << endl;
    }
    break;
case 'F':
    if(lastadded == NULL)
        cout << "Got to add something first" << endl;
    else {
        cout << "Last book added was "
              << lastadded << endl;
        pos = cl.Position(lastadded);
        if(pos)
            cout << "That's still in the collection"
                  << " at position "
                  << pos << endl;
        else
            cout << "You seem to have got rid"
                  << " of it." << endl;
    }
    break;
case 'r':
    pos = GetPos(cl.Length());
    if(pos > 0) {
        ptr = (Book) cl.Remove(pos);
        cout << "Removed " << ptr << " from collection"
              << endl;
        cout << "Collection now has " <<
              cl.Length() << " items" << endl;
        delete ptr;
    }

```

Check Length()

Check Append()

Check Nth()

Check Position()

Check Remove(int)

```

        break;

    case '?':
        cout << "Commands are:" << endl;
        cout << "\tq Quit\n\tA Add to collection\t" << endl;
        cout << "\tf Find item in collection (use number)"
            << endl;
        cout << "\tF Find last added item in collection"
            << endl;
        cout << "\tr Remove item from collection"
            << endl;
        cout << "\tl Length (size) of collection\n" << endl;
        break;

    default:
        ;
    }
}
return EXIT_SUCCESS;
}

```

Most of the testing code should be easy to understand.

Note that function `PickABook()` creates a new data structure in the heap for each call. So, if you choose book 1 ("C++ Programming Language") twice, you will get two separate data structures in the heap, both of which contain this string. These structures are at different addresses. The check `c1.Position(ptr)` in case 'a': will always fail (get a zero result) because the newly allocated book isn't in the collection (even though the collection may already contain a book with similar content). In contrast, the test `c1.Position(lastadded);` in case 'F': may succeed (it will succeed if the last book added is still in the collection because then the address in `lastadded` will be the same as the address in the last of the used pointers in the array).

Function `Position(void*)` implements an identity check. It determines whether the specific item identified by the argument is in the array. It does not check for an equal item. In this context, an identity test is what is required, but sometimes it is useful to have an equality test as well. An equality test would allow you to find whether the array contained an item similar to the one referred to in the query.

In this simplified implementation of class `DynamicArray`, it is not possible to have an equality test. You can only test two data objects for equality if you know their structure or know of a routine for comparing them. You never know anything about a data item pointed to by a `void*` so you can't do such tests.

On the whole, a `DynamicArray` is pretty efficient at its job. Remove operations do result in data shuffling. Some implementations cut down on data shuffling by having an auxiliary array of booleans; the true or false setting of a boolean indicates whether a data element referenced by a pointer is still in the collection. When an item is removed, the corresponding boolean is set to false. Data are only reshuffled when the array has a significant number of "removed" items. This speeds up removals, but makes operations

like `Nth()` more costly (as it must then involve a loop that finds the *n*-th item among those not yet removed).

21.4 CLASS LIST

A list is the hardy perennial that appears in every data structures book. It comes second in popularity only to the "stack". (The "stack" of the data structures books is covered in an exercise. It is not the stack used to organize stackframes and function calls; it is an impoverished simplified variant.)

There are several slightly different forms of list, and numerous specialized list classes that have extra functionality required for specific applications. But a basic list supports the same functionality as does a dynamic array:

- Length
- Append
- Position
- Nth
- Remove

The data storage, and the mechanisms used to implement the functions are however quite different.

The basics of operations on lists were explained in Section 20.6 and illustrated in Figure 20.7. The list structure is made up from "list cells". These are small structs, allocated on the heap. Each "list cell" holds a pointer to a stored data element and a pointer to the next list cell. The list structure itself holds a pointer to the first list cell; this is the "head" pointer for the list.

List cells, next pointers, and the head pointer

```
struct ListCell { void *fData; ListCell *fNext; };

class List {
...
private:
    ListCell    *fHead;
    ...
};
```

This arrangement of a "head pointer" and the "next" links makes it easy to get at the first entry in the list and to work along the list accessing each stored element in turn.

An "append" operation adds to the end of the list. The address of the added list cell has to be stored in the "next" link of the cell that was previously at the end of the list. If you only have a head pointer and next links, then each "append" operation involves searching along the list until the end is found. When the list is long, this operation gets to be a little slow. Most implementations of the list abstract data type avoid this cost by having a "tail" pointer in addition to the head pointer. The head pointer holds the

A "tail" pointer

address of the first list cell, the tail pointer holds the address of the last list cell; they will hold the same address if the list only has one member. Maintaining both head and tail pointers means that there is a little more "housekeeping" work to be done in functions that add and remove items but the improved handling of append operations makes this extra housekeeping worthwhile.

"Previous" links in the list cells

Sometimes it is worth having two list cell pointers in each list cell:

```
struct D_ListCell {
    void      *fData;
    D_ListCell *fNext;
    D_ListCell *fPrev;
};
```

These "previous" links make it as easy to move backwards toward the start of the list as it is to move forwards toward the end of the list. In some applications where lists are used it is necessary to move backwards and forwards within the same list. Such applications use "doubly linked" lists. Here, the examples will consider just the singly linked form.

The class declaration for a simple singly linked list is:

File "List.h" with class declaration

```
#ifndef __MYLIST__
#define __MYLIST__

class List {
public:
    List();

    int    Length(void) const;
    int    Position(void *item) const;

    void    *Nth(int n) const;

    void    Append(void *item);

    void    *Remove(void *item);
    void    *Remove(int itempos);

private:
    struct ListCell { void *fData; ListCell *fNext; };
    int      fNum;
    ListCell *fHead;
    ListCell *fTail;
};

inline int List::Length(void) const { return fNum; }

#endif
```

Apart from the constructor, the public interface for this class is identical to that of class `DynamicArray`.

The struct `ListCell` is declared within the private part of class `List`. `ListCells` are an implementation detail of the class; they are needed only within the `List` code and shouldn't be used elsewhere.

The data members consist of the two `ListCell*` pointers for head and tail and `fNum` the integer count of the number of stored items. You could miss out the `fNum` data member and work out the length each time it is needed, but having a counter is more convenient. The `Length()` function, which returns the value in this `fNum` field, can once again be an inline function defined in the header file.

The remaining member functions would be defined in a separate `List.cp` implementation file.

The constructor is simple, just zero out the count and set both head and tail pointers to `NULL`:

```
#include <stdlib.h>
#include <assert.h>
#include "List.h"

List::List()
{
    fNum = 0;
    fHead = fTail = NULL;
}
```

Figure 21.7 illustrates how the first "append" operation would change a `List` object. The `List` object could be a static, an automatic, or a dynamic variable; that doesn't matter. The data item whose address is to be stored is assumed to be a dynamic (heap based) variable. Initially, as shown in part 1 of Figure 21.7, the `List` object's `fNum` field would be zero, and the two pointers would be `NULL`. Part 2 of the figure illustrates the creation of a new `ListCell` struct in the heap; its `fNext` member has to be set to `NULL` and its `fData` pointer has to hold the address of the data item.

Finally, as shown in part 3, both head and tail pointers of the `List` object are changed to hold the address of this new `ListCell`. Both pointers refer to it because as it is the only `ListCell` it is both the first and the last `ListCell`. The count field of the `List` object is incremented.

The code implementing `Append()` is:

```
void List::Append(void *item)
{
    ListCell *lc = new ListCell;
    lc->fData = item;
    lc->fNext = NULL;
```

*Create new `ListCell`
and link to data item*

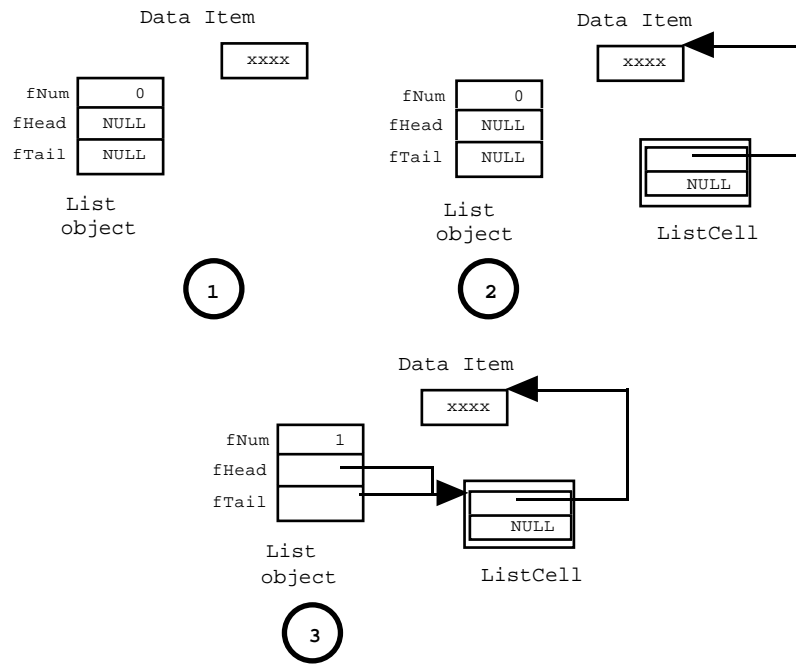


Figure 21.7 The first append operation changing a List object.

```

Add to an empty list      if(fHead == NULL)
                          fHead = fTail = lc;

Add to end of an         else {
existing list             fTail->fNext = lc;
                          fTail = lc;
                          }

Update member count      fNum++;
                          }

```

Figure 21.8 illustrates the steps involved when adding an extra item to the end of an existing list.

Part 1 of Figure 21.8 shows a `List` that already has two `ListCells`; the `fTail` pointer holds the address of the second cell.

An additional `ListCell` would be created and linked to the data. Then the second cell is linked to this new cell (as shown in Part 2):

```
fTail->fNext = lc;
```

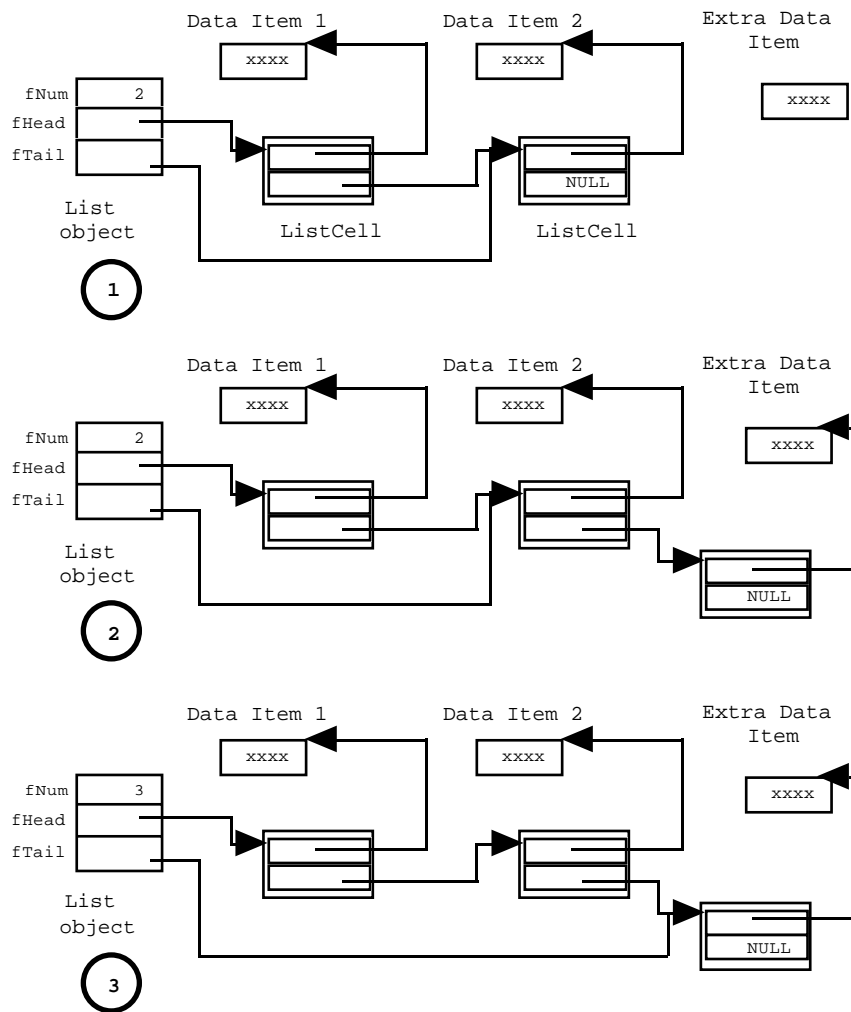


Figure 21.8 Adding an element at the tail of an existing list.

Finally, the `fTail` pointer and `fNum` count are updated as shown in Part 3 of the figure.

The `Position()` function, that finds where a specific data element is stored, involves "walking along the list" starting at the cell identified by the head pointer. The `ListCell*` variable `ptr` is initialized with the address of the first list cell and the position counter `pos` is initialized to 1. The item has been found if its address matches the value in the `fData` member of a `ListCell`; when found, the value for `pos` is returned. If the data address in the current list cell does not match, `ptr` is changed so

that it holds the address of the next list cell. The item will either be found, or the end of the list will be reached. The last list cell in the list will have the value `NULL` in its `fNext` link, this gets assigned to `ptr`. The test in the `while` clause will then terminate the loop. As with the `DynamicArray`, a return value of 0 indicates that the sought item is not present in the list.

```
int List::Position(void *item) const
{
    ListCell *ptr = fHead;
    int pos = 1;
    while(ptr != NULL) {
        if(ptr->fData == item) return pos;
        pos++;
        ptr = ptr->fNext;
    }
    return 0;
}
```

The function `Nth()` is left as an exercise. It also has a `while` loop that involves walking along the list via the `fNext` links. This `while` loop is controlled by a counter, and terminates when the desired position is reached.

***Removing an item
from the middle of a
list***

The basic principles for removing an item from the list are illustrated in Figure 21.9. The `ListCell` that is associated with the data item has first to be found. This is done by walking along the list, in much the same way as was done the `Position()` function. A `ListCell*` pointer, `tmp`, is set to point to this `ListCell`.

There is a slight complication. You need a pointer to the previous `ListCell` as well. If you are using a "doubly linked list" where the `ListCells` have "previous" as well as "next" links then getting the previous `ListCell` is easy. Once you have found the `ListCell` that is to be unhooked from the list, you use its "previous" link to get the `ListCell` before it. If you only have a singly linked list, then it's slightly more complex. The loop that walks along the list must update a "previous" pointer as well as a pointer to the "current" `ListCell`.

Once you have got pointer `tmp` to point to the `ListCell` that is to be removed, and `prev` to point to the previous `ListCell`, then unhooking the `ListCell` from the list is easy, you simply update `prev's fNext` link:

```
prev->fNext = tmp->fNext;
```

Parts 1 and 2 of Figure 21.9 shows the situation before and after the resetting of `prev's fNext` link.

When you have finished unhooking the `ListCell` that is to be removed, you still have to tidy up. The unhooked `ListCell` has to be deleted and a pointer to the associated data item has to be returned as the result of the function.

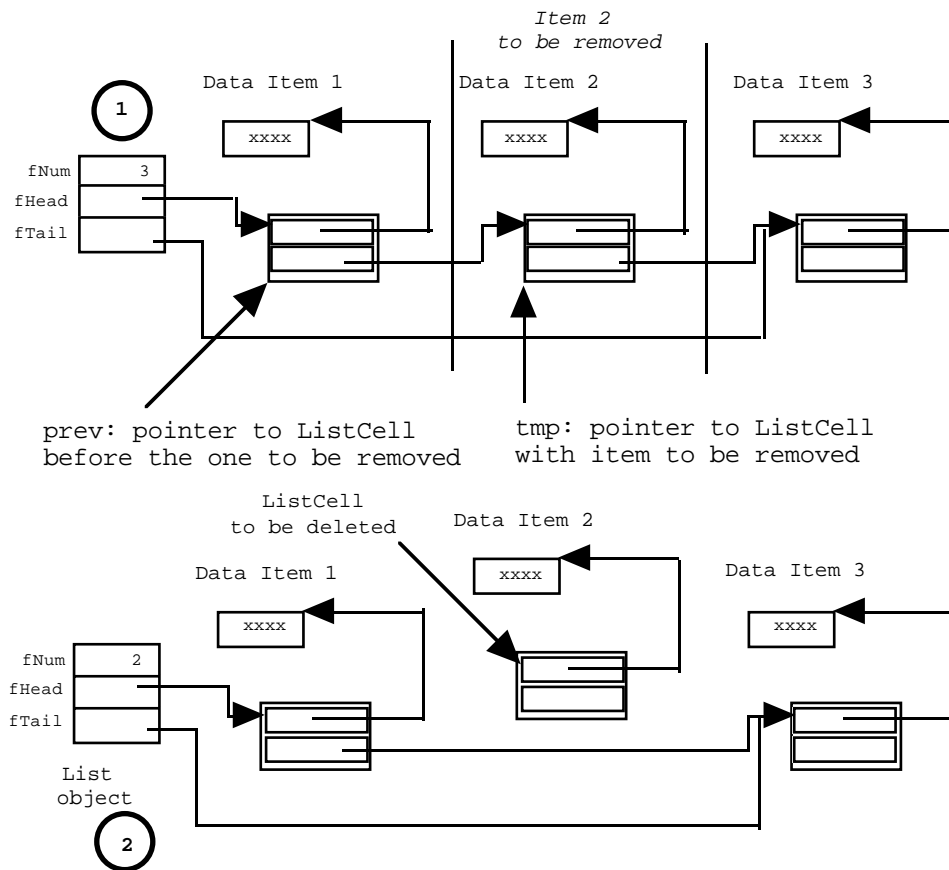


Figure 21.9 Removing an item from the middle of a list.

A `Remove()` function has to check for special cases where the `ListCell` that gets removed is the first, or the last (or the only) `ListCell` in the list. Such cases are special because they make it necessary to update the List's `fHead` and/or `fTail` pointers.

Complications at the beginning and end of a list

If the `ListCell` is the first in the list (`fHead == tmp`) then rather than "unhooking" it, the `fHead` pointer can just be set to hold the address of the next `ListCell`:

Removing the first entry

```
if (fHead == tmp)
    fHead = tmp->fNext;
```

If the `ListCell` should happen to be the last one in the current list (`fTail == tmp`), the List's `fTail` pointer needs to be moved back to point to the previous entry:

Removing the last entry

```
if (fTail == tmp)
```

```
fTail = prev;
```

The two versions of `Remove()` are similar. The functions start with the loop that searches for the appropriate `ListCell` (and deal with the `ListCell` not being present, a `List` may get asked to remove an item that isn't present). Then there is code for unhooking the `ListCell` and resetting `fHead` and `fTail` pointers as required. Finally, there is code that tidies up by deleting the discarded `ListCell` and returning a pointer to the associated data item.

The following code illustrates the version that finds an item by matching its address. The version removing the item at a specified position is left as another exercise.

	<pre>void *List::Remove(void *item) { ListCell *tmp = fHead; ListCell *prev = NULL; while(tmp != NULL) { if(tmp->fData == item) break; prev = tmp; tmp = tmp->fNext; } if(tmp == NULL) return NULL; if(fHead == tmp) fHead = tmp->fNext; else prev->fNext = tmp->fNext; if(fTail == tmp) fTail = prev; fNum--; void *dataptr = tmp->fData; delete tmp; return dataptr; }</pre>
<i>Loop to find the right ListCell</i>	
<i>Item to be removed was not present anyway ListCell is first</i>	
<i>ListCell in middle</i>	
<i>Fix up if last ListCell</i>	
<i>Tidy up</i>	

Go through the code and convince yourself that it does deal correctly with the case of a `List` that only has one `ListCell`.

Test program

You've already seen the test program! Apart from two words, it is identical to the test program for class `DynamicArray`. One word that gets changed is the definition of the datatype for the collection. This has to be changed from


```
DynamicArray    cl;  
  
to  
  
List            cl;
```

The other word to change is the name of the header file that has to be `#included`; this changes from "D.h" to "List.h".

The test program has to be the same. Lists and dynamic arrays are simply different implementations of the same "collection" abstraction.

Overheads with the `List` implementation are significantly higher. If you just look at the code, the overheads may not be obvious. The overheads are the extra space required for the `ListCells` and the time required for calls to `new` and `delete`.

With the `ListCells`, a `List` is in effect using two pointers for each data item while the `DynamicArray` managed with one. Further, each `ListCell` incurs the space overhead of the storage manager's header and trailer records. In the `DynamicArray`, the array allocated in the heap has a header and a trailer, but the cost of these is spread out over the ten or more pointers in that array.

The two implementations have similar costs for their `Position()` functions. The `Append()` for a `List` always involves a call to `new`, whereas this is relatively rare for a `DynamicArray`. A `DynamicArray` has to reshuffle more data during a `Remove()`; but the loop moving pointers up one place in an array is not particularly costly. Further, every (successful) `Remove()` operation on a `List` involves a call to `delete` while `delete` operations are rare for a `DynamicArray`. The difference in the pattern of calls to `new` and `delete` will mean that, in most cases, processing as well as space costs favour the `DynamicArray` implementation of a collection.

Although `DynamicArrays` have some advantages, `Lists` are more generally used and there are numerous special forms of list for specific applications. You had better get used to working with them.

21.5 CLASS BINARYTREE

Just as there are many different kinds of specialized list, there are many "binary trees". The binary tree illustrated in this section is one of the more common, and also one of the simplest. It is a "binary search tree".

A "binary search tree" is useful when:

- you have data items (structs or instances of some class) that are characterized by unique "key" values, e.g. the data item is a "driver licence record" with the key being the driver licence number;

When to use a binary search tree

- you need to maintain a collection of these items, the size of the collection can not be readily fixed in advanced and may be quite large (but not so large that the collection can't fit in main memory);
- items are frequently added to the collection;
- items are frequently removed from the collection;
- the collection is even more frequently searched to find an item corresponding to a given key.

The "keys" are most often just integer values, but they can be things like strings. The requirement that keys be unique can be relaxed, but this complicates searches and removal operations. If the keys are not unique, the search mechanism has to be made more elaborate so that it can deal in turn with each data item whose key matches a given search key. For simplicity, the rest of this section assumes that the keys are just long integers and that they will be unique.

Alternatives: a simple array? Bit slow.

You could just use an array to store such data items (or a dynamic array so as to avoid problems associated with not knowing the required array size). Addition of items would be easy, but searches for items and tidying up operations after removal of items would be "slow" (run times would be $O(N)$ with N the number of items in the collection). The search would be $O(N)$ because the items wouldn't be added in any particular order and so to find the one with a particular key you would have to start with the first and check each in turn. (You could make the search $O(\lg N)$ by keeping the data items sorted by their keys, but that would make your insertion costs go up because of the need to shuffle the data.) Removal of an item would require the movement of all subsequent pointers up one place in the array, and again would have a run time $O(N)$.

Alternatives: a hash table?? Removals too hard.

If you simply needed to add items, and then search for them, you could use a hash table. (Do you still remember hash tables from Chapter 18?) A "search" of a hash table is fast, ideally it is a single operation because the key determines the location of the data item in the table (so, ideally, search is $O(1)$). The keys associated with the data items would be reduced modulo the size of the hash table to get their insertion point. As explained in Chapter 18, key collisions would mean that some data items would have to be inserted at places other than the location determined simply by their key. It is this factor that makes it difficult to deal with removals. A data item that gets removed may have been one that had caused a key collision and a consequent shift of some other data item. It is rather easy to lose items if you try doing removals on a hash table.

Binary search tree performance

Binary trees offer an alternative where insertions, searches, and removals are all practical and all are relatively low cost. On a good day, a binary tree will give you $O(\lg N)$ performance for all operations; the performance does deteriorate to $O(N)$ in adverse circumstances.

A binary search tree gets its $O(\lg N)$ performance in much the same way as we got $O(\lg N)$ performance with binary search in a sorted array (Chapter 13). The data are going to be organized so that testing a given search key against a value from one of the

data items is going to let you restrict subsequent search steps to either one of two subtrees (subsets of the collection). One subtree will have all the larger keys, the other subtree will have all the smaller keys. If your search key was larger than the key just tested you search the subtree with the large keys, if your key is smaller you search the other subtree. (If your search key was equal to the value just checked, then you've found the data item that you wanted.)

If each such test succeeded in splitting the collection into equal sized subtree, then the search costs would be $O(\lg N)$. In most cases, the split results in uneven sized subtree and so performance deteriorates.

Tree structure

A binary search tree is built up from "tree-node" structures. In this section, "tree-nodes" (tn) are defined roughly as follows: *Simple "tree-nodes"*

```
struct tn {
    long    key;
    void    *datalink;
    tn      *leftlink;
    tn      *rightlink;
};
```

The tree-node contains a key and a pointer to the associated data record (as in the rest of this chapter, it is assumed that the data records are all dynamic structures located in the heap). Having the key in the tree-node may duplicate information in the associated data record; but it makes coding these initial examples a little simpler. The other two pointers are links to "the subtree with smaller keys" (leftlink) and "the subtree with larger keys" (rightlink). (The implementation code shown later uses a slightly more sophisticated version where the equivalent of a "tree-node" is defined by a class that has member functions and private data members.)

The actual binary tree object is responsible for using these tn structures to build up an overall structure for the data collection. It also organizes operations like searches. It keeps a pointer, the "root" pointer, to the "root node" of the tree. *BinaryTree object*

Figure 21.10 illustrates a simple binary search tree. The complete tree start starts with a pointer to the "root" tree-node record. The rest of the structure is made up from separate tree-nodes, linked via their "left link" and "right link" pointers.

The key in the root node is essentially arbitrary; it will start by being the key associated with the first data record entered; in the example, the key in the root node is 1745.

All records associated with keys less than 1745 will be in the subtree attached via the root node's "left link". In Figure 21.10, there is only one record associated with a smaller key (key 1642). The tree node with key 1642 has no subtrees, so both its left link and right link are NULL.

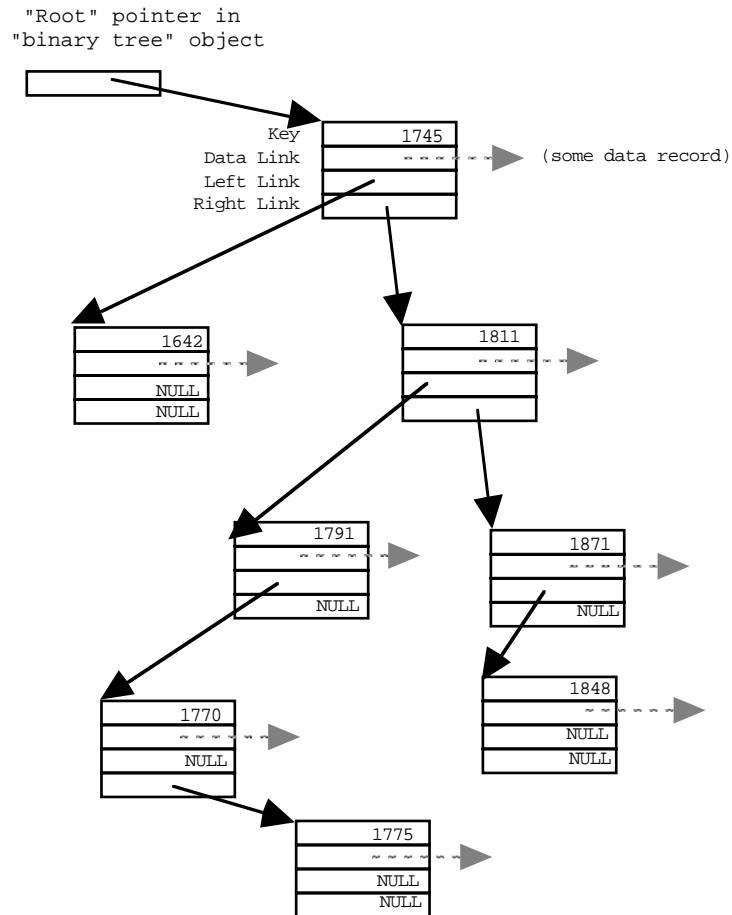


Figure 21.10 A "binary search tree".

All records having keys greater than 1745 must be in the right subtree. The first entered might have been a record with key 1811. The left subtree of the tree node associated with key 1811 will hold all records that are less than 1811 but greater than 1745. Its right subtree will hold all records associated with keys greater than 1811.

The same structuring principles apply "recursively" at every level in the tree.

The shape of the tree is determined mainly by the order in which data records are added. If the data records are added in some "random" order, the tree will be evenly balanced. However, it is more usual for the addition process to be somewhat orderly, for example records entered later might tend to have larger keys than those entered earlier. Any degree of order in the sequence of addition results in an unbalanced tree.

Thus, if the data records added later do tend to have larger keys, then the "right subtrees" will tend to be larger than the left subtrees at every level.

Searching

You can find a data record associated with a given "search key" (or determine that there is no record with that key) by a simple recursive procedure. This procedure takes as arguments a pointer to a subtree and the desired key value:

```
recursive_search(tn *sub_tree_ptr, Key search_key)
    if sub_tree_ptr is NULL
        return NULL;

    compare_result = Compare search_key and sub_tree_ptr->Key

    if(compare_result is EQUAL)
        return sub_tree_ptr->Data;

    else
    if(compare_result is LESS)
        return recursive_search(
            sub_tree_ptr->left_link, search_key)
    else
        return recursive_search(
            sub_tree_ptr->right_link, search_key)
```

*Pseudo-code of a
recursive search
function*

This function would be called with the "root" pointer and the key for the desired record.

Like all recursive functions, its code needs to start by checking for termination conditions. One possible terminating condition is that there is no subtree to check! The function is meant to chase down the left/right links until it finds the specified key. But if the key is not present, the function will eventually try to go down a NULL link. It is easiest to check this at the start of the function (i.e. at the next level of recursion). If the subtree is NULL, the key is not present so the function returns NULL.

*Termination test of
recursive search
function*

If there is a subtree to check, the search key should be compared with the key in the tree-node record at the "root of the current subtree". If these keys are equal, the desired record has been found and can be returned as the result of the function.

Comparison of keys

Otherwise the function has to be called recursively to search either the left or the right subtree as shown.

*Recursive call to
search either the left
or the right subtree*

Two example searches work as follows:

recursive_search(root, 1770)	recursive_search(root, 1795)
sub_tree_ptr != NULL	sub_tree_ptr != NULL
compare 1770, 1745	compare 1795, 1745
result is GREATER	result is GREATER
recursive_search(recursive_search(

Program call

*Execution at first
level of recursion*

	right_subtree, 1770)	right_subtree, 1795)
Execution at second level of recursion	sub_tree_ptr != NULL compare 1770, 1811 result is LESS recursive_search(left_subtree, 1770)	sub_tree_ptr != NULL compare 1795, 1811 result is LESS recursive_search(left_subtree, 1795)
Execution at third level of recursion	sub_tree_ptr != NULL compare 1770, 1791 result is LESS recursive_search(left_subtree, 1770)	sub_tree_ptr != NULL compare 1795, 1791 result is GREATER recursive_search(right_subtree, 1795)
Execution at fourth level of recursion	sub_tree_ptr != NULL compare 1770, 1770 result is EQUALS return data for key 1770	sub_tree_ptr == NULL return NULL

Addition of a record

The function that adds a new record is somewhat similar to the search function. After all, it has to "search" for the point in the tree structure where the new record can be attached.

Of course, there is a difference. The addition process has to change the tree structure. It actually has to replace one of the `tn*` (pointers to `treenode`) pointers.

Changing an existing pointer to refer to a new treenode

A pointer has to be changed to hold the address of a new tree node associated with the new data record. It could be the root pointer itself (this would be the case when the first record is added to the tree). More usually it would be either the left link pointer, or the right link pointer of an existing tree-node structure that is already part of the tree.

The recursive function that implements the addition process had better be passed a reference to a pointer. This will allow the pointer to be changed.

The addition process is illustrated in Figure 21.11. The figure illustrates the addition of a new `treenode` associated with key 1606. The first level of recursion would have the addition function given the root pointer. The key in the record identified by this pointer, 1745, is larger than the key of the record to be inserted (1606), so a recursive call is made passing the left link of the first tree node.

The `treenode` identified by this pointer has key 1642, again this is larger than the key to be inserted. So a further recursive call is made, this time passing the left link of the `treenode` with key 1642.

The value in this pointer is `NULL`. The current tree does not contain any records with keys less than 1642. So, this is the pointer whose value needs to be changed. It is to point to the subtree with keys less than 1642, and there is now going to be one – the new `treenode` associated with key 1606.

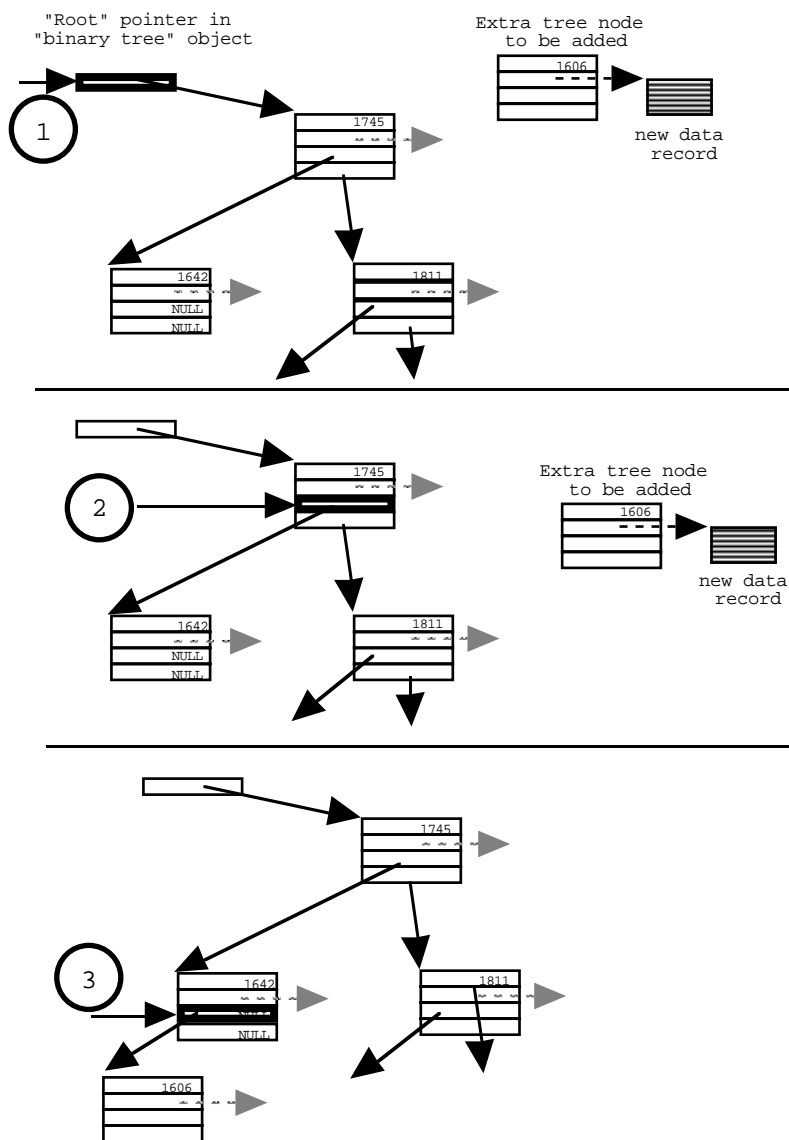


Figure 21.11 Inserting a new record.

A pseudo-code outline for the addition function is:

```
recursive_add(a treenode pointer passed by reference ('c'),
              pointer to new data item, key)
  if c contains NULL
```

*Pseudo-code of a
recursive addition
function*

```

        make a new treenode and fill in its
            pointer to data and its key
        change contents of pointer 'c' to hold the address
            of this new treenode
        return

compare_result = Compare key and c->Key

if(compare_result is EQUAL)
    warn user that duplicate keys are not allowed
    return without making any changes

else
    if(compare_result is LESS)
        return recursive_add(
            sub_tree_ptr->left_link, data item, key)
    else
        return recursive_add(
            sub_tree_ptr->right_link, data item, key)

```

Recursion terminates if the treenode pointer argument contains NULL. A NULL pointer is the one that must be changed; so the function creates the new treenode, fills in its data members, changes the pointer, and returns.

If the pointer argument identifies an existing treenode structure, then its keys must be compared with the one that is to be inserted. If they are equal, you have a "duplicate key"; this should cause some warning message (or maybe should throw an exception). The addition operation should be abandoned.

In other cases, the result of comparing the keys determines whether the new record should be inserted in the left subtree or the right subtree. Appropriate recursive calls are made passing either the left link or the right link as the argument for the next level of recursion.

Removal of a record

Removing a record can be trickier. As shown in Figures 21.12 and 21.13, some cases are relatively easy.

If the record that is to be removed is associated with a "leaf node" in the tree structure, then that leaf can simply be cut away (see Figure 21.12).

If the record is on a vine (only one subtree below its associated treenode) then its treenode can be cut out. The treenode's only subtree can be reattached at the point where the removed node used to be attached (Figure 21.13).

Things get difficult with "internal nodes" that have both left and right subtrees. In the example tree, the treenodes associated with keys 1645 and 1811 are both "internal" nodes having left and right subtrees. Such nodes cannot be cut from the tree because this would have the effect of splitting the tree into two separate parts.

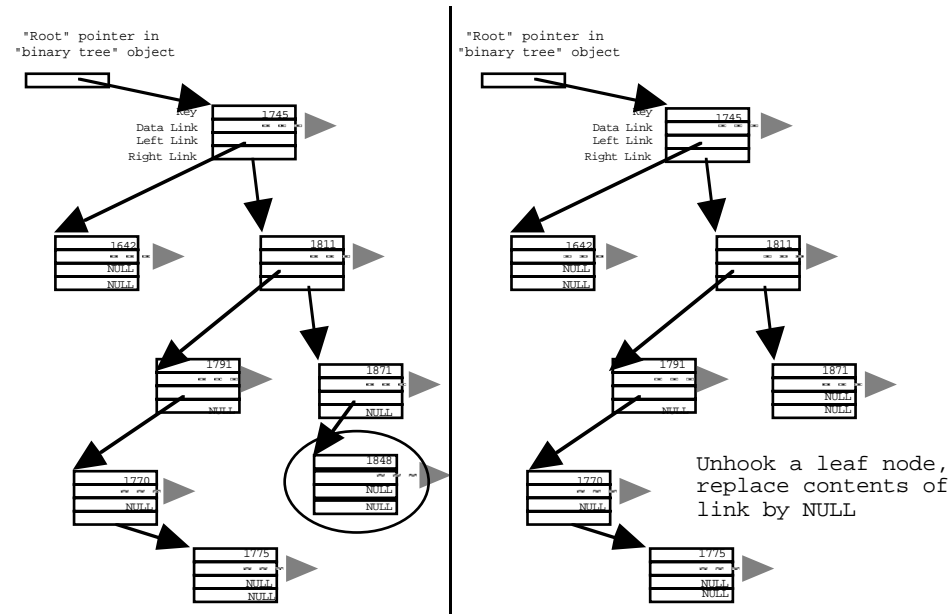


Figure 21.12 Removing a "leaf node".

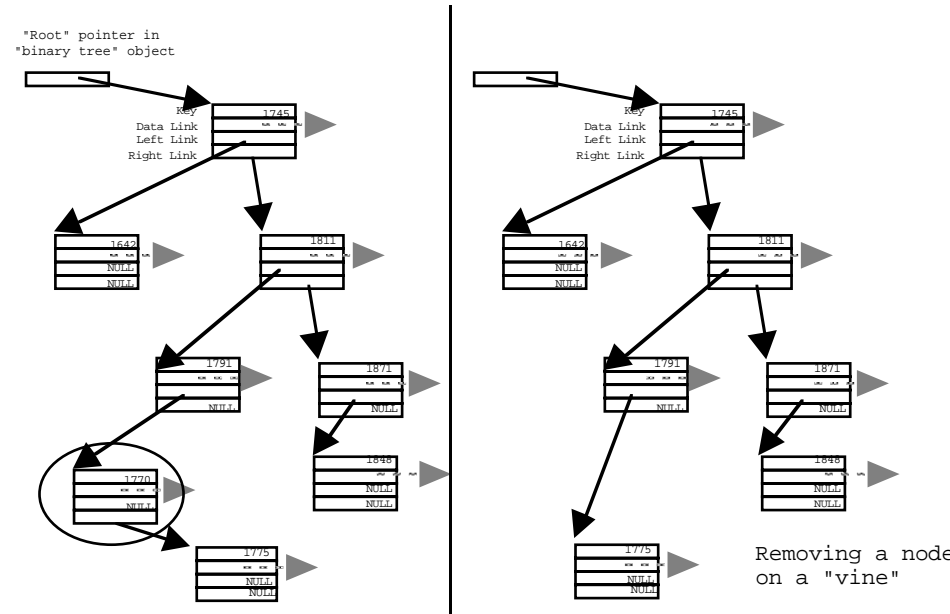


Figure 21.13 Removing a "vine" node.

The key, and pointer to data record, can be removed from a treenode provided that they are replaced by other appropriate data. Only the key and the pointer to data record get changed; the existing left link and right link from the treenode are not to be changed. The replacement key and pointer have to be data that already exist somewhere in the tree. They have to be chosen so that the tree properties are maintained. The replacement key must be such that it is larger than all the keys in the subtree defined by the existing left link, and smaller than all the keys in the subtree defined by the existing right link.

Promoting a "successor"

These requirements implicitly define the key (and associated pointer to data record) that must be used to replace the data that are being removed. They must correspond to that data record's "successor". The successor will be the (key, data record) combination that has the smallest key that is larger than the key being deleted from an internal node.

Find the (key, data) combination to be deleted

Figure 21.14 illustrates the concept of deletion with promotion of a successor. The (key, data) combination that is to be deleted is the entry at the root, the one with key 1745. The key has first to be found; this would be done in the first step of the process.

Find the successor

The next stage involves finding the successor – the (key, data) combination with the smallest key greater than 1745. This will be the combination with key 1770. You find it by starting down the right subtree (going to the treenode with key 1811) and then heading down left links for as far as you can go.

Promoting the successor

The key and pointer to data record must then be copied from the successor treenode. They overwrite the values in the starting treenode, replacing the key and data pointer that are supposed to get deleted. Thus, in the example in Figure 21.14, the "root" treenode gets to hold the key 1770 and a pointer to the data item associated with this key.

Removal of duplicate entry

Of course, key 1770 and its associated data record are now in the tree twice! They exist in the treenode where the deleted (key, data record) combination used to be, and they are still present in their original position. The treenode where they originally occurred must be cut from the tree. If, as in the case shown in Figure 21.14, there is a right subtree, this replaces the "successor treenode". This is shown in Figure 21.14, where the treenode with key 1775 becomes the new "left subtree" of the treenode with key 1791.

If you had to invent such an algorithm for yourself, it might be hard. But these algorithms have been known for the last forty years (and for most of those forty years, increasing numbers of computing science students have been given the task of coding the algorithms again). Coding is not hard, provided the process is broken down into separate functions.

One function will find the (key, data record) combination that is to be deleted; the algorithm used will be similar to that used for both the search and insert operations. Once the (key, data) combination has been found, a second function can sort out whether the deletion involves cutting off a leaf, excising a node on a vine, or replacing the (key, data record) by a successor. Cutting off a leaf and excising a vine node are both straightforward operations that don't require further subroutines. If a successor has to be promoted, it must be found and removed from its original position in the tree.

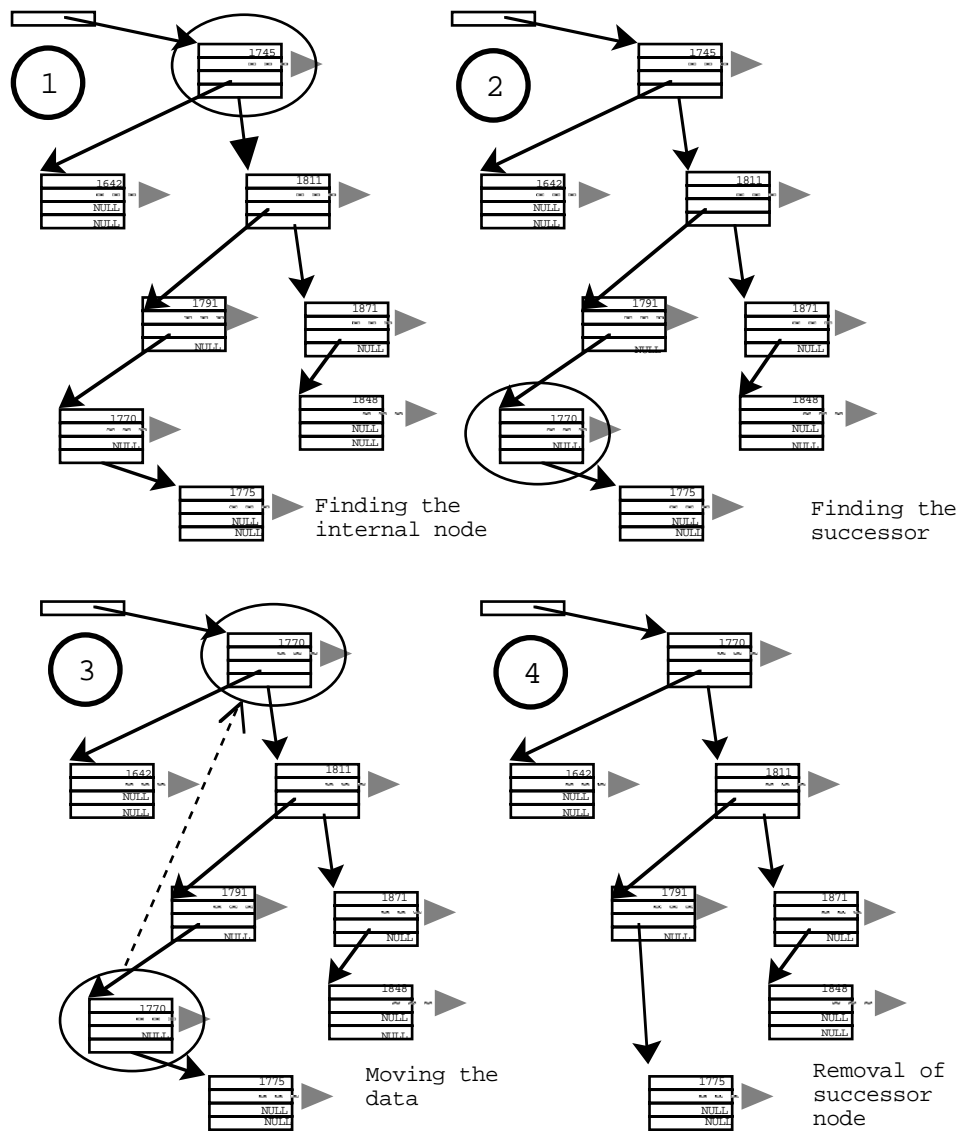


Figure 21.14 Removing a "internal" node and promoting a successor.

Pseudo-code for the initial search step is as follows:

```
recursive_remove(a treenode pointer passed by reference ('c'), key)
    if c contains NULL
        return NULL
```

Pseudo-code of a recursive removal function

```

compare_result = Compare key and c->Key

if(compare_result is EQUAL)
    return result of Delete function applied 'c'

else
    if(compare_result is LESS)
        return recursive_remove(
            sub_tree_ptr->left_link, data item, key)
    else
        return recursive_remove(
            sub_tree_ptr->right_link, data item, key)

```

The function should return a pointer to the data record associated with the key that is to be removed (or NULL if the key is not found). Like the search function, the remove function has to check whether it has run off the end of the tree looking for a key that isn't present; this is done by the first conditional test.

Passing a pointer by reference

The rest of code is similar to the recursive-add function. Once again, a pointer has to be passed by reference. The actual pointer variable passed as a reference argument could be the root pointer for the tree (if we are deleting the root node of a tree with one or two records); more usually, it will be either the "left link" or "right link" data member of another treenode. The contents of this pointer may get changed. At the stage where the search process is complete, the argument pointer will hold the address of the treenode with the key that is to be removed. If the treenode with the "bad" key is a leaf, the pointer should be changed to NULL. If the treenode with the bad key is a vine node, the pointer should be changed to hold the address of that treenode's only child. The actual changes are made in the auxiliary function Delete. But the pointer argument has to be passed by reference to permit changes to its value.

Determining the deletion action

Pseudo code for the Delete function is as follows:

```

delete(a treenode pointer passed by reference ('c'))
    save pointer to data record associated with bad key

    if treenode pointed to by c is a leaf (both subtree
        links being NULL)
        delete that treenode,
        set c to NULL
    else
        if treenode pointed to by c is a vine node with only a
            right subtree
            set c to hold address of right subtree
            delete discarded treenode;
        else
            similar code for c being a vine node with only a
                left subtree
        else
            get successor (removing it from tree when found)
            replace key and data pointers with values from

```

```

        successor
    delete the successor treenode

    return pointer to data record associated with bad key

```

Most of the code of Delete is made up by the conditional construct that checks for, and deals with the different possible situations.

A pseudo code outline for the function to find the successor treenode and remove it from its current position in the tree is:

```

successor(a treenode pointer passed by reference ('c'))
    if c->treenode has a subtree attached to its left link
        call successor recursively passing left link
    else
        have found successor treenode, unhook it
        c changed to point to treenode's right subtree
    return treenode

```

The BinaryTree class

More detailed outlines of the search, add, and remove operations can be found in numerous text books; many provide Pascal or C implementations. Such implementations have the functions as separate, independent global functions. We need these functions packaged together as part of a BinaryTree class.

A BinaryTree object doesn't have much data, after all most of the structure representing the tree is made up from treenodes. A BinaryTree object will own a pointer to the "root" treenode; it is useful if it also owns a count of the number of (key, data record) combinations currently stored in the tree.

A BinaryTree should do the following:

- "Num items" (equivalent to Length of lists etc)
Report how many items are stored in the collection.
- Add
Add another (key, data record) combination to the tree, placing it at the appropriate position (the Add operation should return a success/failure code, giving a failure return for a duplicate key).
- Find
Find the data record associated with a given key, returning a pointer to this record (or NULL if the specified key is not present).
- Remove
Removes the (key, data record) combination associated with a specified key, returning a pointer to the data record (or NULL if the specified key is not present).

*What does a
BinaryTree object
own?*

*What does a
BinaryTree object
do?*

Because the structure is moderately complex, and keeps getting rearranged, it would be worth including a debugging output function that prints a representation of the structure of the tree.

The discussion of operations like remove identified several additional functions, like the one to find a treenode's successor. These auxiliary functions would be private member functions of the class.

The declaration for the class would be in a header file:

```
File "biny.h"  #ifndef __MYBINY__
                 #define __MYBINY__

                 #define DEBUG

                 class TreeNode;

                 class BinaryTree
                 {
                 public:
                     BinaryTree();

                     int    NumItems() const;

                     int    Add(void* nItem, long key);
                     void    *Find(long key);
                     void    *Remove(long key);
                 #ifdef DEBUG
                     void    PrintOn(ostream& out) const;
                 #endif
                 private:
                     int    AuxAdd(TreeNode*& c, void* nItem, long key);
                     void    *AuxFind(TreeNode* c, long key);
                     void    *AuxRemove(TreeNode*& c, long key);

                     void    *Delete(TreeNode*&c);
                     TreeNode *Successor(TreeNode*& c);

                 #ifdef DEBUG
                     void    AuxPrint(TreeNode* c, ostream& out, int depth)
                     const;
                 #endif
                     TreeNode    *fRoot;
                     int          fNum;
                 };

                 inline int BinaryTree::NumItems(void) const { return fNum; }

                 #endif
```

Note the declaration:

```
class TreeNode;
```

This needs to appear before class `BinaryTree` to allow specification of function arguments and variables of type `TreeNode*` (pointer to `TreeNode`). The full declaration of class `TreeNode` does not need to appear in the header file. Client's using class `BinaryTree` will know that the tree is built up using auxiliary `TreeNode` objects. However, clients themselves never need to use `TreeNodes`, so the declaration of the `TreeNode` class can be "hidden" in the implementation file of class `BinaryTree`.

Some of the functions have odd looking arguments:

Those `TreeNode&`
arguments*

```
void      *Delete(TreeNode*&c);
```

An expression like `TreeNode*&` is a little unpronounceable, but that really is the hardest part about it. Reference arguments, like `int&`, were illustrated in many earlier examples (starting with the example in section 12.8.3). We use references when we need to pass a variable to a function that may need to change the value in the variable. In this case the variable is of type "pointer to `TreeNode`" or `TreeNode*`. Hence, a "pointer to a `TreeNode` being passed by reference" is `TreeNode*&`.

As is illustrated below in the implementation, the public interface functions like `Find()` do very little work. Almost all the work is done by an associated private member function (e.g. `AuxFind()`). The public function just sets up the initial call, passing the root pointer to the recursive auxiliary function.

As usual, trivial member functions like `NumItems()` can be defined as inlines. Their definitions go at the end of the header file.

Implementation

The implementation code would be in a separate file, `biny.cp`.

After the usual `#includes`, this file would start with the declaration of the auxiliary `TreeNode` class and the definition of its member functions.

```
class TreeNode {
public:
    TreeNode(long k, void *d);
    TreeNode*& LeftLink(void);
    TreeNode*& RightLink(void);
    long Key(void) const;
    void *Data(void) const;
    void Replace(long key, void *d);
private:
    long      fKey;
    void      *fData;
    TreeNode  *fLeft;
    TreeNode  *fRight;
};
```

Class `TreeNode` is simply a slightly fancy version of the `tn_treenode` struct illustrated earlier. It packages the key, data pointer, and links and provides functions that the `BinaryTree` code will need to access a `TreeNode`.

The function definitions would specify most (or all) as "inline":

```
TreeNode::TreeNode(long k,void *d)
{
    fKey = k;
    fData = d;
    fLeft = NULL;
    fRight = NULL;
}

inline long TreeNode::Key(void) const { return fKey; }
inline void *TreeNode::Data(void) const { return fData; }
inline void TreeNode::Replace(long key, void *d)
    { fKey = key; fData = d; }
```

When `TreeNode`s are created, the key and data link values will be known so these can be set by the constructor. `TreeNode`s always start as leaves so the left and right links can be set to `NULL`.

The `Key()` and `Data()` functions provide access to the private data. The `Replace()` function is used when the contents of an existing `TreeNode` have to be replaced by data values promoted from a successor `TreeNode` (part of the deletion process explained earlier).

The `LeftLink()` and `RightLink()` functions return "references" to the left link and right link pointer data members. They are the first examples that we have had of functions that return a reference data type. Such functions are useful in situations like this where we need in effect to get the address of a data member of a struct or class instance.

```
inline TreeNode*& TreeNode::LeftLink(void) { return fLeft; }
inline TreeNode*& TreeNode::RightLink(void) { return fRight; }
```

The code for class `BinaryTree` follows the declaration and definition of the auxiliary `TreeNode` class. The constructor is simple; it has merely to set the root pointer to `NULL` and zero the count of data records.

```
BinaryTree::BinaryTree()
{
    fRoot = NULL;
    fNum = 0;
}
```


The public functions `Find()`, `Add()`, and `Remove()` simply set up the appropriate recursive mechanism, passing the root pointer to the auxiliary recursive function. Thus, `Find()` is:

```
void *BinaryTree::Find(long key)
{
    return AuxFind(fRoot, key);
}
```

the other functions are similar in form.

The `AuxFind()` function implements the recursive mechanism exactly as explained earlier:

```
void *BinaryTree::AuxFind(TreeNode* c, long key) Search
{
    if(c == NULL)
        return NULL;

    int compare = key - c->Key();

    if(compare == 0)
        return c->Data();

    if(compare < 0)
        return AuxFind(c->LeftLink(), key);
    else
        return AuxFind(c->RightLink(), key);
}
```

The `AuxAdd()` function implements the "recursive_add" scheme outlined above, and includes necessary details such as the update of the count of stored data records, and the return of a success or failure indicator:

```
int BinaryTree::AuxAdd(TreeNode*& c, void* nItem, long key) Addition
{
    if(c==NULL) {
        c = new TreeNode(key, nItem);
        fNum++;
        return 1;
    }

    int compare = key - c->Key();

    if(compare == 0) {
        cout << "Sorry, duplicate keys not allowed" << endl;
        return 0;
    }

    if(compare < 0)
```

```

        return AuxAdd(c->LeftLink(), nItem, key);
    else
        return AuxAdd(c->RightLink(), nItem, key);
}

```

The following three functions implement the remove mechanism. First there is the `AuxRemove()` function that organizes the search for the `TreeNode` with the "bad" key:

Removal

```

void *BinaryTree::AuxRemove(TreeNode*& c, long key)
{
    if(c == NULL)
        return NULL;

    int compare = key - c->Key();

    if(compare == 0)
        return Delete(c);

    if(compare < 0)
        return AuxRemove(c->LeftLink(), key);
    else
        return AuxRemove(c->RightLink(), key);
}

```

The `Delete()` function identifies the type of deletion action and makes the appropriate rearrangements:

```

void *BinaryTree::Delete(TreeNode*& c)
{
    void *deaddata = c->Data();
    if((c->LeftLink() == NULL) && (c->RightLink() == NULL))
        { delete c; c = NULL; }
    else
    if(c->LeftLink() == NULL) {
        TreeNode* temp = c;
        c = c->RightLink();
        delete temp;
    }
    else
    if(c->RightLink() == NULL) {
        TreeNode* temp = c;
        c = c->LeftLink();
        delete temp;
    }
    else {
        TreeNode* temp = Successor(c->RightLink());
        c->Replace(temp->Key(), temp->Data());
        delete temp;
    }
    return deaddata;
}

```

```
}
```

The `Successor()` function finds and unhooks a successor node:

```
TreeNode *BinaryTree::Successor(TreeNode*& c)
{
    if(c->LeftLink() != NULL)
        return Successor(c->LeftLink());
    else {
        TreeNode *temp = c;
        c = c->RightLink();
        return temp;
    }
}
```

The print function is only intended for debugging purposes. It prints a rough representation of a tree. For example, if given the tree of Figure 21.10, it produces the output:

```
      1848
     1811
    1791
   1775
  1770
 1745
1642
```

With a little imagination, you should be able to see that this does represent the structure of the tree (the less imaginative can check by reference to Figure 21.15).

Given that everything else about this tree structure is recursive, it is inevitable that the printout function is too! The public interface function sets up the recursive process calling the auxiliary print routine to do the work. The root pointer is passed in this initial call.

```
#ifdef DEBUG
void BinaryTree::PrintOn(ostream& out) const
{
    AuxPrint(fRoot, out, 0);
}
```

The (recursive) auxiliary print function takes three arguments – a pointer to treenode to process, an indentation level (and also an output stream on which to print). Like all recursive functions, it starts with a check for a terminating condition; if the function is called with a `NULL` subtree, there is nothing to do and an immediate return can be made.

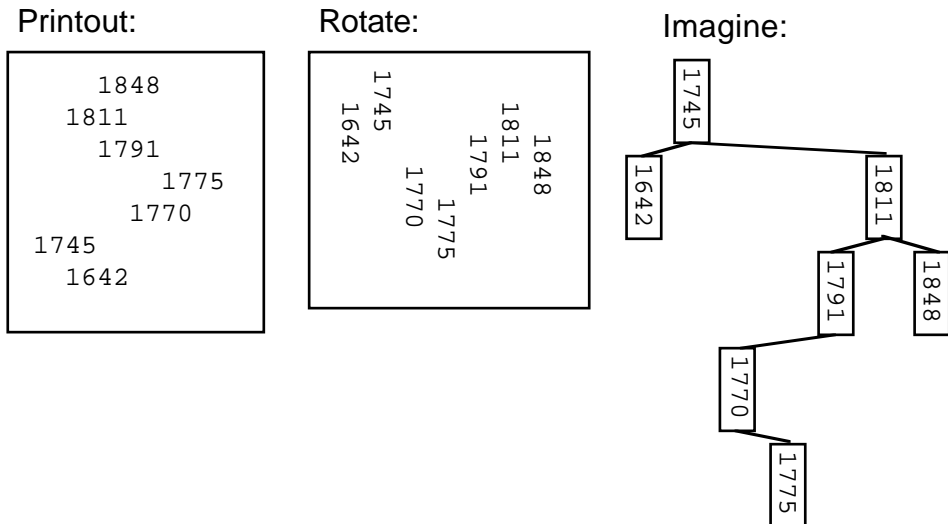


Figure 21.15 Printed representation of binary tree.

```

void BinaryTree::AuxPrint(TreeNode* c, ostream& out,
int depth) const
{
    if(c == NULL)
        return;

    AuxPrint(c->RightLink(), out, depth + 2);

    for(int i=0; i< depth; i++)
        cout << " ";
    cout << c->Key();
    cout << endl;

    AuxPrint(c->LeftLink(), out, depth + 2);
}
#endif
  
```

Deal with right subtree

Deal with data in this treenode

Deal with left subtree

If the function has been passed a pointer to a `TreeNode`, processing starts by a recursive call to deal with the *right* subtree (with an increased indentation specified in the `depth` argument). This recursive call will get all higher valued keys printed before the current entry. (The higher valued keys are printed first so that when you rotate the printout they are on the right!) Once the right subtree has been dealt with, the key in the current `TreeNode` is printed (the `for` loop arranges for the appropriate indentation on the line). Finally, the left subtree is processed.

The recursive `AuxPrint()` function gets to process every node in the tree – it is said to perform a "tree traversal". There are many circumstances where it is useful to visit every node and so traversal functions are generally provided for trees and related data structures. Such functions are described briefly in the section on "Iterators" in Chapter 23. *Tree traversal*

Test Program

As with the other examples in this chapter, the class is tested using a simple interactive program that allows data records to be added to, searched for, and removed from the tree.

The test program starts with the usual `#includes`, then defines a trivial `DataItem` class. The tree is going to have to hold some kind of data, class `DataItem` exists just so as to have a simple example. A `DataItem` owns an integer key and a short name string.

```
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include "biny.h"

class DataItem {
public:
    DataItem(long k, char txt[]);
    void PrintOn(ostream& out) const;
    long K() const;
private:
    long    fK;
    char    fName[20];
};

DataItem::DataItem(long k, char txt[] )
{
    fK = k;
    int len = strlen(txt);
    len = (len > 19) ? 19 : len;
    strncpy(fName, txt, len);
    fName[len] = '\0';
}

void DataItem::PrintOn(ostream& out) const
{
    out << fName << " : " << fK << ";" << endl;
}

inline long DataItem::K() const { return fK; }
```

The actual `BinaryTree` object has been defined as a global. Variable `gTree` is initialized by an implicit call to the constructor, `BinaryTree::BinaryTree()` that gets executed in "preamble" code before entry to `main()`.

```
BinaryTree gTree;
```

The test program uses auxiliary functions to organize insertion and deletion of records, and searches for records with given keys. Insertion involves creation of a new `DataItem`, initializing it with data values entered by the user. This item is then added to the tree. (Note, the code has a memory leak. Find it and plug it!)

```
void DoInsert()
{
    char buff[100];
    int k;
    cout << "Enter key and name" << endl;
    cin >> k >> buff;
    DataItem *d = new DataItem(k, buff);
    if(gTree.Add(d, k)) cout << "OK" << endl;
    else cout << "Problems" << endl;
}
```

The search function gets a key from the user, and asks the tree to find the associated `DataItem`. If found, a record is asked to print itself:

```
void DoSearch()
{
    cout << "Enter search key : ";
    int k;
    cin >> k;
    DataItem *item = (DataItem*) gTree.Find(k);
    if(item == NULL) cout << "Not found " << endl;
    else {
        cout << "Found record : ";
        item->PrintOn(cout);
    }
}
```

Similarly, the delete function prompts for a key, and asks the tree for the record. If a `DataItem` is returned, it is deleted so recovering the space that it occupied in the heap.

```
void DoDelete()
{
    cout << "Enter key : ";
    int k;
    cin >> k;
    DataItem *item = (DataItem*) gTree.Remove(k);
    if(item == NULL) cout << "Wasn't there" << endl;
}
```

```

        else {
            cout << "Removed item: " ;
            item->PrintOn(cout);
            cout << "Destroyed" << endl;
            delete item;
        }
    }
}

```

The `main()` function has the usual interactive loop typical of these small test programs:

```

int main()
{
    for(int done = 0; ! done; ) {
        cout << ">";
        char ch;
        cin >> ch;
        switch(ch) {
case 'q':  done = 1; break;
case 's':
            DoSearch();
            break;
case 'i':
            DoInsert();
            break;
case 'd':
            DoDelete();
            break;
case 'p':
            gTree.PrintOn(cout);
            break;
case '?':
            cout << "q to quit, s to Search, i to "
                  "Insert,d to Delete, p Print" << endl;
            break;
default:
            ;
        }
    }
    return EXIT_SUCCESS;
}

```

21.6 COLLECTION CLASS LIBRARIES

You should not need to implement the code for any of the standard collection classes. Your IDE will come with a class library containing most of the more useful collection classes. Actually, it is likely that your IDE comes with several class libraries, more than one of which contains implementations of the frequently used collection classes.

"Standard Template Library"

Your IDE may have these classes coded as "template classes". Templates are introduced in Chapter 25. Templates offer a way of achieving "generic" code that is more sophisticated than the `void*` pointer to data item used in the examples in this chapter. In the long run, templates have advantages; they provide greater flexibility and better type security than the simpler approaches used in this chapter. They have some minor disadvantages; their syntax is more complex and they can complicate the processes of compilation and linking. There is a set of template classes, the "Standard Template Library", that may become part of "standard C++"; these classes would then be available on all implementations. Some of the classes in the Standard Template Library are collection classes.

Your IDE will also provide a "framework class library". A framework class library contains classes that provide prebuilt parts of a "standard Macintosh" program (as in Symantec's TCL2 library) or "standard Windows" program (as with Borland's OWL or Microsoft's MFC). Framework libraries are discussed more in Chapter 31. Your IDE's framework class library contains additional collection classes. These classes will have some restrictions. Normally they can only be used if you are building your entire program on top of the framework.

The IDE may have additional simpler examples of collection classes. They will be similar to, or maybe a little more sophisticated than the examples in this chapter and in Chapter 24. Many other libraries of useful components and collection classes are in the public domain and are available over the Internet.

Use and re-use!

You should not write yet another slightly specialized version of "doubly linked list", or "queue based on list". Instead, when analysing problems you should try to spot opportunities to use existing components.

Read the problem specification. It is going to say either "get and put a file record" (no need for collection classes), or "do something with this block of data" (arrays rather than collection classes), or "get some items from the user and do something with them" (probably a collection class needed).

Once you have established that the problem involves a variable number of data items, then you know that you probably need to use a collection class. Read further. Find how items are added to the collection, and removed. Find whether searches are done on the collections or whether collections of different items are combined in some way.

Once you have identified how the collection is used, you more or less know which collection class you want. If items are frequently added to and removed from the collection and the collection is often searched to find an item, you may be dealing with a case that needs a binary tree. If you are building collections of different items, and then later combining these, then you are probably going to need a list. If much of the processing consists of gathering items then sorting them, you might be able to use a priority queue.

For example, a problem may require you to represent polynomials like $7x^3 + 6x^2 - 4x + 5$, and do symbolic arithmetic like multiplying that first polynomial by $4x^2 - 3x + 7$. These polynomials are collections (the items in the collection represent the coefficients

for different powers of x); they vary in size (depending on the person who types in the examples). The collections are combined. This is going to be a case where you need some form of list. You could implement your own; but it is much better to reuse a prebuilt, debugged, possibly optimized version.

Initially, stick to the simple implementations of collection classes like those illustrated in this chapter or the versions in most public domain class libraries. As you gain confidence in programming, start to experiment with the more advanced templates provided by your IDE.

The good programmer is not the one who can rapidly hack out yet another version of a standard bit of code. The good programmer is the one who can build a reliable program by combining small amounts of new application specific code with a variety of prebuilt components.

EXERCISES

1. Implement class Stack.

A stack owns a collection of data items (void* pointers in this implementation). It supports the following operations:

Add (preferred name "Push")

adds another element to the front of the collection;

First (preferred name "Pop")

removes most recently added element from the collection;

Length (preferred name "Size")

reports number of elements in the collection;

Full

returns true if collection is full and error would result from further Push operation;

Empty

return true if collection is empty and error would result from further Pop operation..

Your stack can use a fixed size array for storing the void* pointers, or a dynamic array or a list. (Function Full should always return false if a list or dynamic array is used.)

2. Complete the implementation of class List by writing the Nth() and Remove(int pos) member functions.
3. The "awful warning" exercise.

Modify the implementation of List::Remove() by changing the last part of the code from:

```
fNum--;  
void *dataptr = tmp->fData;  
delete tmp;
```

```
        return dataptr;

to

        fNum--;
        void *dataptr = tmp->fData;
        delete prev;

        return dataptr;
```

Note that this introduces two errors. First, there is a memory leak. The listcell pointed to by `tmp` is being discarded, but as it is not being deleted it remains occupying space in the heap.

There is a much more serious error. The listcell pointed to by `prev` is still in use, it is still part of the list, but this code "deletes it".

Compile and run the modified program. Test it. Are any of the outputs wrong? Does your system "crash" with an address error?

Unless your IDE has an unusually diligent memory manager, the chances are that you will find nothing wrong in your tests of your program. Everything will appear to work.

Things work because the listcells that have been deleted exist as "ghosts" in the heap. Until their space gets reallocated, they still appear to be there. Their space is unlikely to be reallocated during the course of a short test with a few dozen addition, search, and removal operations.

Although you know the code is wrong, you will see it as working.

This should be a warning to all of us who happily say "Ah it's working!" after running a few tests on a piece of code.

22 A World of Interacting Objects

The two examples in this chapter illustrate the "world of interacting objects" that is typical of a program built using classes. They give you a practical model for an alternative to the "top-down" design approach used extensively in Part III. The examples also illustrate slightly simplified, informal versions of some of the schemes that are commonly used to document more elaborate object based programs.

The first example, "RefCards", is a little bit like the example in Section 17.3. In that example, a program manipulated "customer records" that contained data such as customer name, and amount ordered. Actually, that program manipulated a single structure in memory; the rest of the records were in a file. When a record was needed, it got loaded into memory. Now we can use things like an insane of a standard "list" class to hold a collection of records in memory, transferring these records to and from disk only when the program finishes and is restarted. The data records this time are "reference cards" – the sort of thing used to keep references to papers when you are doing scientific research. These records contain things like "authors' names", "paper title", "journal", and "page numbers". Although the RefCards program has some similarities to the earlier example, the use of classes in its design results in an implementation that is beginning to show a quite distinct structure.

Refcards example

The second example is an object-based reworking and elaboration of the "information retrieval" example from Section 18.3. That version of the program allowed the user to build up files containing newspaper articles, with an index based on a predefined set of keywords defined by an initialized data array. It used two programs. One added data to the file; the other performed single searches. Now we need something more general.

Infostore example

The program is to allow the user to define the "vocabulary" of keywords (as in the earlier example, it is actually a vocabulary of concepts as several different words can map onto a single concept used in the index). This vocabulary is to be extendible. If the user thinks of a new concept, or an additional word that maps onto an existing concept, then the system must allow this word to be added. This makes the system

more flexible. It is not limited like the original to scientific articles. If you want a collection of travel articles, you simply make up a suitable vocabulary and start saving data.

The programs are also to be integrated. Vocabulary changes, article addition, and searches are all to be handled by the one program. The previous separate programs performed single functions – article addition, or search. The new program does not have a single function that can serve as a starting point for design. Instead, it gets built from objects – a "user interaction" object, a "vocabulary" object, and an "info store" object.

22.1 REFCARDS

Scholars doing research used to keep their records on "reference cards". Although computer data bases and file systems are now common, some people still use the old cards and many of the simpler computer systems use "cards" as a metaphor in their design. Reference cards would contain a number of data fields including: authors, title of paper, journal name, issue number, page number, year of publication, keywords, and possibly an abstract. Computer based versions have the advantage that you can easily search a collection of such "cards" checking each to identify those that contain a particular keyword, or a specified author.

Specification

The RefCard program will work with collections of "reference card" records. There is no specified maximum for the number of records in a collection, but the implementation can assume that the largest collections will have at most a few hundred records. A "reference card" record is to have text data fields for the author, title, journal, keyword, and abstract; it is also to have integer fields for issue number, year, first page, and last page. The text data fields should each hold 250 characters.

The "RefCard" program is to:

- allow a user to create a collection of "reference card" records.
- save a collection to a disk file and then reload a collection in subsequent runs of the program.
- let the user add a record to a collection, view an existing record, change a field in an existing record, or remove a record from the collection.
- let the user search for the first record with a particular word (name) in the "author", "title", "keyword", or "abstract" field; after displaying the first matched record the program is to allow searches for subsequent matching records. In addition to

searches on specified individual fields, the search system should also allow for a search for a word in any of these fields.

- let the user display the contents of the "title", "keyword", "author", or "abstract" fields from all cards in the collection.

The program should be interactive, prompting the user for a command and for other data as needed. The command entry system should allow the user to view the range of commands appropriate in a given context.

22.1.1 Design

Preliminary design: Objects and their classes:

So, where do you start?

You start by saying "What objects are present in the running program?"

Some objects are obvious. The program will have "RefCard" objects. These will own text strings (character arrays) in which data like author names, keywords, title and so forth are stored. They will also probably own some integer data like "year of publication". What do they do? They display their data, they can store their data to disk files, they can read data from disk files. They've got to get filled with data somehow; so it would seem reasonable if a "RefCard" could interact with a user allowing data to be entered or changed.

RefCard objects

The program is to work with at most a few hundred of these "RefCard" objects. They aren't that large (a little less than 1300 bytes), and since there will be only a few hundred at most, it is reasonable to keep them all in memory while the program is running (more than 750 records fit into a megabyte and that isn't much for most current PCs). Keeping all records in memory will make searches much faster. So, there will have to be something that holds the collection in memory. There is no "unique search key" that could be used for something like a binary tree. Instead we will need something like a "list" or "dynamic array". A dynamic array seems more appropriate. Removal operations are going to be rare; lists are better than dynamic arrays only in cases where removals are frequent. The dynamic array can be an instance of an "off the shelf" class from a class library.

A list or dynamic array

Although a dynamic array can be used to hold the actual collection, there had better be something a little more elaborate. We need operations that involve the collection as a whole – like the searches (as well as operations like saving the collection to a disk file). The search system needs information like the string that is sought, and the current position in the collection if we are doing operations like "find" and "find again". We will also need some idea like "current position" if we are to handle requests like "show card ...", and "remove card ...". A "CardCollection" object could own the actual dynamic array with the cards and, in addition, other information like search string, position, and name of the file. A CardCollection object could organize searches and

CardCollection object

similar operations like viewing a particular field from all records. What exactly does a `CardCollection` own and do? Don't know. Its role will become clearer as we iterate through the design process.

***UserInteraction
object***

Although the `CardCollection` object could handle some of the interactions with the user (like getting information needed for a search), it will be useful to have more general interactions handled by a "UserInteraction" object. Actually, this is probably not strictly necessary for this program. However, as we move to more elaborate programs (particularly those built on top of framework class libraries as discussed in Part V) we will see "standard" arrangements of classes. It will be "standard" to have some "UserInteraction" object that controls the overall program flow. This `UserInteraction` object will create the `CardCollection` object, get the top level commands from the user ("add a card", "show a card", "change a card", "do a search", ...), sort out files and do similar tasks. In this program, the `UserInteraction` object will only have one `CardCollection`; but if a `UserInteraction` object could use more than one window for input and output it could have a separate `CardCollection` associated with each separate window (that is how word processors and other programs typically work).

main()?

If the `CardCollection` object is organizing everything, what is there left for the "main program"? Not much. A main program for one of these object based systems is usually simple: create the principal object, do any other initializations, tell principal object to "run", when "run" finishes do any tidying up that may be needed. In fact, we can write the main program already:

```
int main()
{
    UserInteraction    UI;
    UI.Initialize();
    UI.Run();
    UI.Terminate();
    return 0;
}
```

Classes and objects

We aren't yet into "class hierarchies" (see next chapter). When we get class hierarchies we will have to do a bit more work characterizing classes and finding relationships between classes. For now, things are simple. The various types of object identified so far correspond directly to the classes that we will need. They are summarized in Figure 22.1. Most are shown as "fuzzy blobs". The fuzzy blobs represent "analysis" level classes. We have a rough idea as to what they own and what they do, but we can't yet define any hard boundaries.

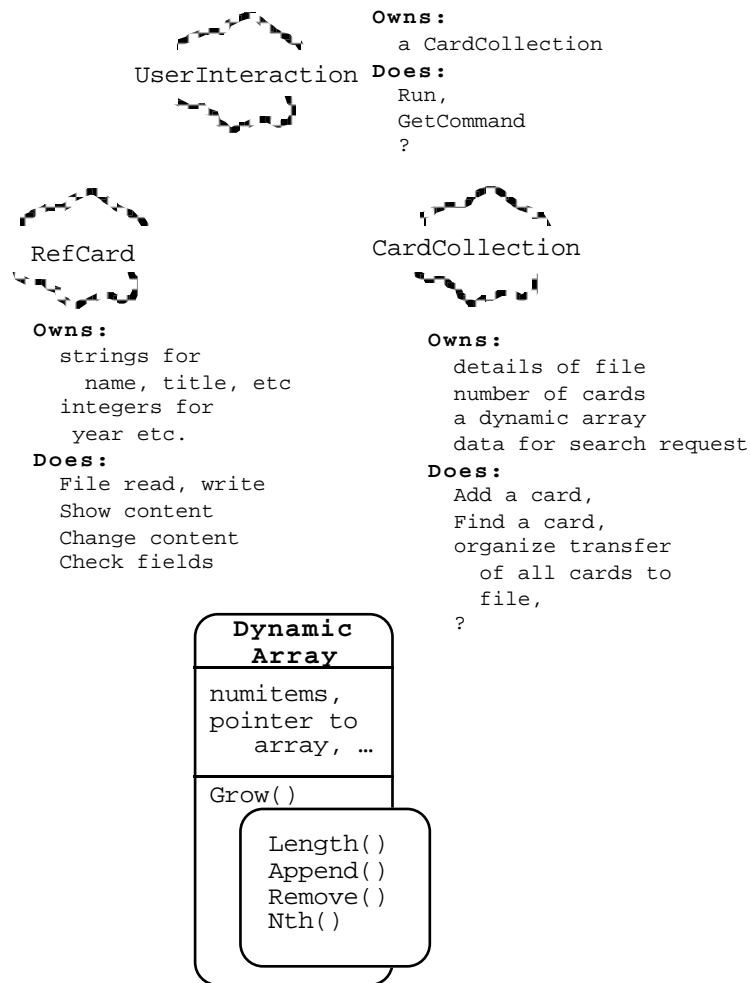


Figure 22.1 First idea for classes for RefCards example.

Class `DynamicArray` is an exception. It is shown as a design level class diagram with a firm boundary, exact specification of its interface and private data. After all, its from a class library and has already been designed and implemented. Our next task will be to firm up those fuzzy blobs so that they too can be defined with firm boundaries.

Design 2: Characterize interactions among objects

The next stage of the design process is typically iterative. Our aim in this stage is to clarify what instances of these classes own and exactly what they do. We aren't yet

interested in exactly how they do their tasks, just trying to identify how the overall work of the program gets split up into tasks that each different object must perform.

Scenarios

The best approach to assigning responsibilities is to try out "scenarios" that illustrate how some of the program's major tasks can be accomplished. A simple scenario for this program would be "the things that happen when its time to terminate"; it will be simple because the only important thing that will happen is that the `CardCollection` must save all the cards back to a disk file. A more complex scenario would be "the things that happen when we search for cards with the word 'Stroustrup' in their 'author' fields". This scenario would be more complex because it probably involves more types of object and more elaborate processing tasks.

While working through these scenarios you guess. You guess things like "object-1 will ask object -2 to perform task A". Then, you examine the implications. If object-2 is to perform task A then it had better own (or at least have access to) all the data needed to carry out this task. If this scenario has object-2 owning some data, then those data had better not appear as belonging to a different object in some other scenario. If object-2 doesn't own the data but has "access to them", then you had better sort out how object-2 got that access. Presumably some other object gave object-2 a pointer to the data, but which other object and when? You guess again noting down an extra responsibility for another object (and, hence, a function for its class).

Naturally, some of your guesses are wrong. You run through a few scenarios. Discover that in different scenarios you've allocated ownership of some specific data to different objects. You have to decide which class of object really should own the data and go back and change the scenario that is incorrect.

Of course, this is still a fairly simple program so the scenario analysis and other mechanisms for "fleshing out" the roles of the classes will be completed rather easily.

Example scenario: initialization

Assumptions: 1) the main program has already created a `UserInteraction` object, 2) the constructor for the `UserInteraction` object created a `CardCollection` object, 3) this `CardCollection` object is "empty", it will have a dynamic array with some default number of slots, and it will have initialized things like its count of cards to zero.

Initialization task: get the user to enter a filename, if the file exists read data on existing cards, otherwise create an empty file where a new collection of cards can be saved.

Possible interactions:

1. `UserInteraction` object gets user to enter a filename. It could open the file, but handling the file might be better left to the `CardCollection` object. So ...
2. `UserInteraction` object asks `CardCollection` object to open the file, passing the name of the file as an argument in the request.

3. The `CardCollection` object should first try to find an existing file (an "open" operation with "no-create" specified).

If this works then existing data should be loaded (see next step).

If that operation failed, the `CardCollection` object should try to create a new file.

If it can open a new file, the `CardCollection` object should report success (it should also make certain that all its data fields are initialized properly, though possibly this should already have been done in its constructor).

4. If the `CardCollection` object was able to open an old file, it has to load the existing cards into memory.

It is going to have to have a loop in which it creates `RefCard` objects, gets them filled with data from the file, and then adds them to its `DynamicArray` (lets call that `fStore`).

It will be easiest if the first data item in the file is an integer specifying how many cards there are. The `CardCollection` object can read this integer into its card count (`fNumCards`) and then have a loop like the following:

```
for(int i = 1; i<= fNumCards; i++) {  
    RefCard *r= new RefCard;  
    r -> ReadFrom(input file);  
    fStore.Append(r);  
}
```

The `CardCollection` object doesn't know what data is in a `RefCard` so it can't read the data. A `RefCard` does know what data it wants. So, as shown, the `CardCollection` object asks each newly created `RefCard` to read itself.

When this process finishes, we will have a set of `RefCards` that have each been created in the heap. Their addresses will be held in the `DynamicArray` `fStore`. Because it owns `fStore`, the `CardCollection` object can get at the individual `RefCards` whenever it needs them.

There should be some checks for successful file transfers. If everything ran OK, then the `Load` function should return a success indicator to the `Open` function which can then report success to the `UserInteraction` object.

5. The `UserInteraction` object should check the success/failure indicator returned by the `CardCollection` object; if the file couldn't be opened the `UserInteraction` object should either terminate the program or loop to allow the user to guess another file name.

6. Finally, the `UserInteraction` object should tell the `CardCollection` object to print a status report so that the user knows how many cards are in the collection.

**Diagramming the
object interactions of
a scenario**

It is usually helpful to represent such interactions through a diagram, like that shown in Figure 22.2. The diagram shows things that happen at different times; the time increases as you go down the diagram. The entries shown across the diagram illustrate what different objects are doing.

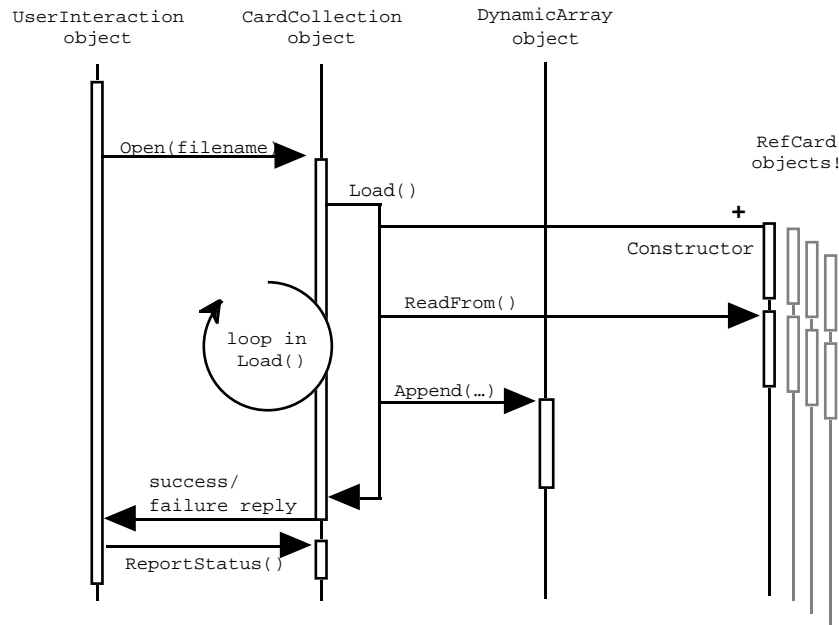


Figure 22.2 Object interactions when loading from file (activities resulting from execution of `UserInteraction::Initialize()` for an UI object).

Back in Chapter 15 where design techniques for top down functional decomposition were reviewed, diagrams were rated as less useful than text descriptions and pseudo code. It was suggested that diagrams like those charting possible function calls weren't that informative. These class interaction diagrams do provide a much clearer indication of the dynamics of a program. They can capture what is going on in each significant subtask that must be performed.

**Meanings of symbols
in diagram**

A diagram like that shown in 22.2 doesn't try to represent everything; for example, there is nothing about the file not opening and having to be created. The focus is on the more important interactions between objects. The classes of the objects involved are indicated by the class labels across the top. Single vertical lines show where an object is in existence. The `UserInteraction` (UI), `CardCollection`, and `DynamicArray` object all exist before the start of the diagrammed scenario and continue to exist after its

finish. The `RefCard` objects only get created part way through the scenario, so their vertical lines start half way down.

The outline rectangles indicate where an object is active, i.e. executing a function or its own function has invoked a global function or action by some other object. In the example, all activity is part of `UserInteraction::Initialize()` (the long rectangle for the `UserInteraction` object). Arrows indicate function calls (and sometimes are used to provide information on results returned). The line with the '+' tag indicates a place where a new object is to be created.

Thus, the diagram shows the process of a call from the `Initialize()` function of a UI object to the `Open()` function of a `CardCollection` object. This `Open()` function calls `Load()` (executed by the same `CardCollection` object). Function `CardCollection::Load()` has a loop. In this loop, a `RefCard` object gets created (the '+' line), then gets asked to execute its `ReadFrom()` function. Then the `DynamicArray` object gets asked to do an `Append()` operation.

Equivalence between diagram and earlier text description

Results:

The process of analysing and diagramming this scenario has added new functions to the responsibilities proposed for class `CardCollection`. It is going to have to have functions like:

New responsibilities identified from scenario

<code>int CardCollection::Open(char*)</code>	<i>open file with given name</i>
<code>void CardCollection::Load()</code>	<i>create RefCards, get them filled in with data from file</i>
<code>void CardCollection::ReportStatus()</code>	<i>state what was read</i>

The `CardCollection` object should probably be responsible for recording the name of its file (this name will be needed in some of the output shown to the user, like that from `ReportStatus()`). It had also better have an `fstream` data member so that it can actually keep hold of the file from the time its told to `Open()` till the time its told to `Close()` and save the data. So, we have also got a couple of extra data members:

```
CardCollection {
public:
    ...
private:
    int      fNumCards;
    char     *fFileName;
    ifstream fFile;
    DynamicArray fStore;
    ...
};
```

Example scenario: termination

Assumptions: The CardCollection object has an open fstream to which it can write its cards.

Termination: The termination stage of the program requires that all RefCards get saved to file and then they should probably be deleted (not absolutely necessary here as the program is about to finish, but it would matter if the program was supposed to allow the user to continue by opening another card collection).

Possible Interactions:

Interactions on termination

Figure 22.3 diagrams an idea as to the interactions. The UserInteraction object will ask the CardCollection object to Close(). The CardCollection object would start by calling its own Save() function. This would start by setting the file so that any existing data gets overwritten, then it would write out the number of cards, after this there would be a loop in which each RefCard object in the DynamicArray gets told to save its own data. The code would be something like:

```
fFile.seekp(0);
fFile << fNumCards << endl;
for(int i=1; i <= fNumCards; i++) {
    RefCard *r = (RefCard*) fStore.Nth(i);
    r->WriteTo(fFile);
}
```

Once the cards had been saved, the file should be closed and then there could be a second loop in which they get removed from the DynamicArray and are then deleted. This would be done in the CardCollection::Close() function:

```
...
fFile.close();
for(int i = fNumCards; i > 0; i--) {
    RefCard *r = (RefCard*) fStore.Remove(i);
    delete r;
}
```

When the function CardCollection::Close() was completed, control would return to UserInteraction::Terminate(). There would then be an opportunity to delete the CardCollection object.

Results:

New responsibilities identified

This analysis identifies just one extra function. Class CardCollection will have to define a Save() routine; like Load() this will be a private member function.

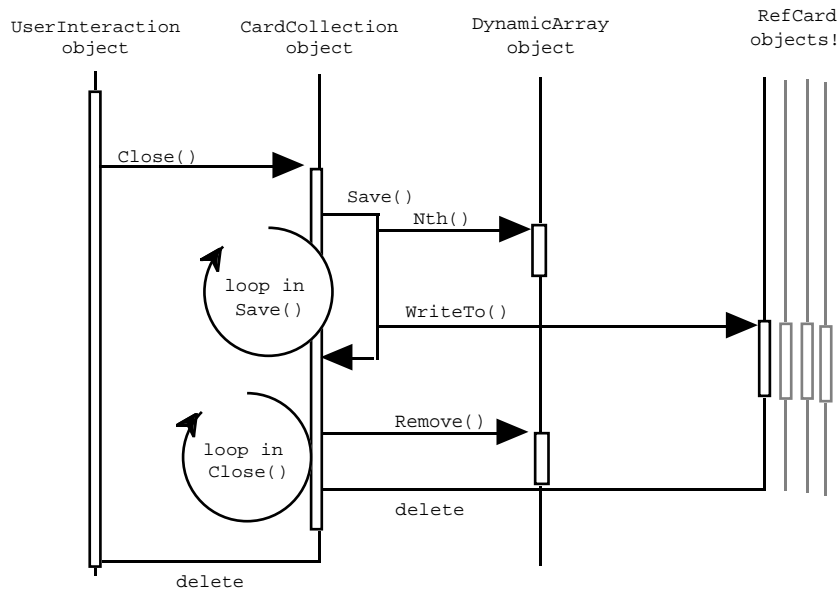


Figure 22.3 Object interactions resulting from `UserInteraction::Terminate()`.

Example: Interaction between the User and the UserInteraction object

The code for handling interactions between the human user and the UI object will itself be simple. This code will form the body of the function `UserInteraction::Run()`. Basically, all we need is a loop that gets a command from the user (a single letter will do), the command will (usually) be easy to convert into a request that the `CardCollection` perform some action like showing an existing card, or adding a new card.

A user command like "add a card" can be passed directly to the `CardCollection` object. A command like "change contents of a card" or "delete a card" will require additional input to identify the card. (Cards may as well be identified by their sequence number in the collection; these sequence numbers can appear in listing so that the user knows which card is which.) We will need a few simple auxiliary routines to get extra input data.

`UserInteraction::Run()` will be something along the following lines:

```

void UserInteraction::Run()
{
    int    done = 0;
    cout << "Enter commands, ? for help" << endl;
    for(;; !done; ) {

```

*Simple command
loop*

```

        char command = GetCommand();
        switch(command) {
        case 'q' :      done = 1; break;
        case '?' :      Help(); break;
        case 'a' :      fCollection->AddCard(); break;
        case 'c' :      DoChange(); break;
        case 'd' :      DoDelete(); break;
        ...
        case 'v' :      fCollection->DoView(); break;
        default :
            cout << "Command " << command <<
                " not recognized" << endl;
        }
    }
}

```

***Auxiliary private
member functions***

It implies the existence of a largish number of very simple auxiliary routines. These will all become additional private member functions of class `UserInteraction`. A function like `GetCommand()` will just read a character, convert it to lower case, and return it.

Functions like `DoChange()` and `DoDelete()` both need the user to input a card number. Obviously the number entered has to be validated; it will have to be in the range 1 to *n* where *n* is the number of cards owned by the collection. (Note, the `UserInteraction` object has to be able to ask the `CardCollection` object how many cards it has.) Since this number input routine is needed by several routines, it might as well become another private member function. Once these "DoX()" functions have got any necessary additional input, they will call matching functions of the `CardCollection` object.

Elaboration of these simple member functions can be handled by using much the same sort of top down functional decomposition techniques as presented in Part III. Here the "top" is a single (moderately complex) member function of a specific class.

Results:

***New responsibilities
identified***

Class `CardCollection` must report the number of cards it has, so we need:

```
int CardCollection::NumCards()
```

as an extra public member function. Class `CardCollection` will also need `AddCard()`, `DoView()`, `ShowCard(int cardnum)` and similar functions. Some additional scenarios will be needed to clarify the prototypes (argument lists) for these functions.

A largish number of simple private member functions have been identified for class `UserInteraction`. We are going to need:

```
char    UserInteraction::GetCommand();    get character
```

int	UserInteraction::PickACard();	<i>get valid card number</i>
void	UserInteraction::Help();	<i>list valid commands</i>
void	UserInteraction::DoDelete();	<i>organize delete</i>
void	UserInteraction::DoChange();	<i>organize change</i>
void	UserInteraction::DoShow();	<i>organize show</i>

Example scenario: add a card

Adding a card: The collection will have to create a new RefCard, get the RefCard to interact with the user to obtain the information that it needs for its data members, and then it will have to add the card to the dynamic array. An interaction diagram is shown in Figure 22.4.

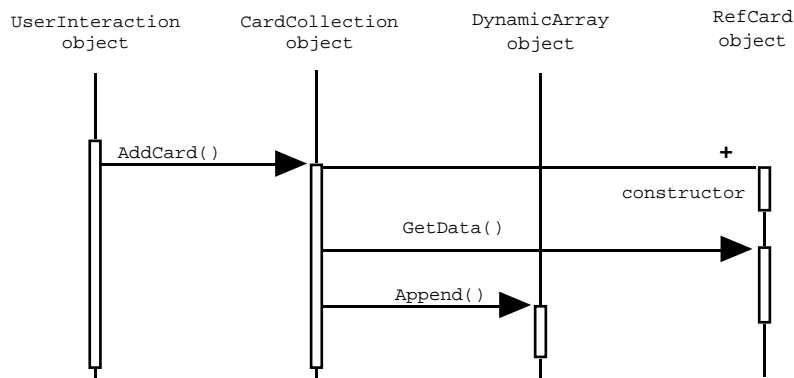


Figure 22.4 Object interactions resulting from a User's "add card" command.

Most of the work can be left to the newly created RefCard object. This can prompt the user to enter data for each of the data members (title, authors, year of publication etc.).

Results:

Need functions:

```
void CardCollection::AddCard();
```

and

```
void RefCard::GetData();
```

Example scenario: getting a card shown or changing an existing card

These two operations are going to involve very similar interactions among the participating objects. The pattern is illustrated in Figure 22.5.

The `UserInteraction` object will execute a routine (`PickACard()`) that prompts the user for the card number. This will involve an interaction with the `CardCollection` object to make sure that the number entered is in range.

If a valid card number is entered, the `UserInteraction` object will ask the `CardCollection` object to `ChangeCard()` or `ShowCard()`.

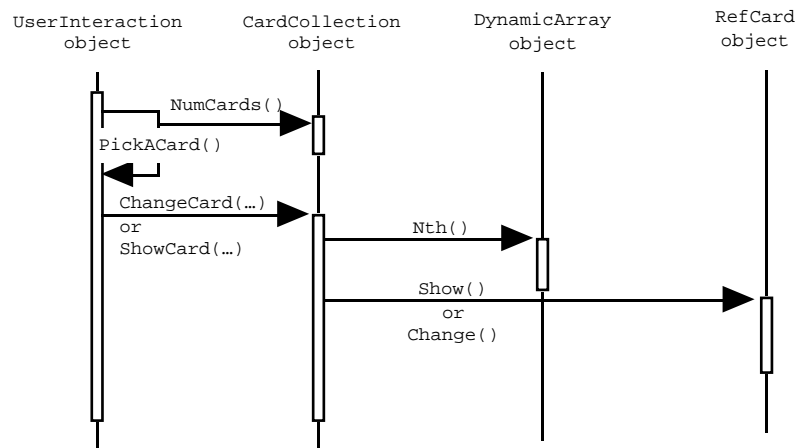


Figure 22.5 Object interactions resulting from a User's "show card" or "change card" commands.

The `CardCollection` object will use the `Nth()` member function of its `DynamicArray` to get a pointer to the chosen `RefCard`. The "show" or "change" command will then be forwarded to the `RefCard` object. A `RefCard` will handle "show" by displaying the contents of all data members. A "change" command will involve prompting the user to identify the data member to be changed (single character input), display of current contents of that data member, and acceptance of new input value.

Example scenario: deleting a card

The interactions for a delete command are summarized in Figure 22.6

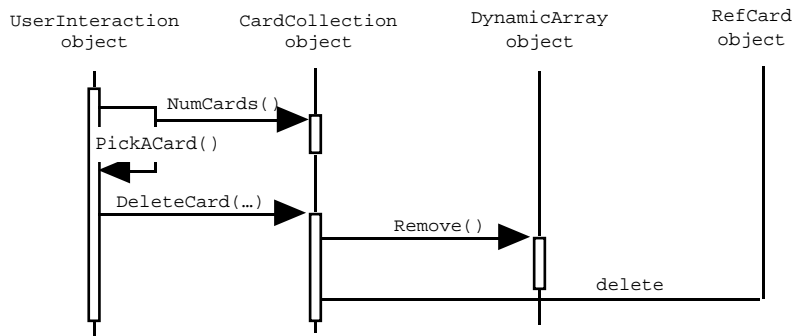


Figure 22.6 Object interactions resulting from a User's "delete card" command.

Example scenario: viewing a field from every card

View: The user is prompted to identify which field is to be viewed (choice restricted to 'author', 'title', 'keywords', or 'abstract'). The contents of the chosen field should then be displayed (along with a card identifier number) for each card in the collection.

Possible Interactions:

Figure 22.7 diagrams an idea as to the interactions. The `UserInteraction` object can simply pass a "view" request to the `CardCollection` object.

The `CardCollection` object will first have to get the user to identify the field that is to be displayed. This will involve the use of another auxiliary private member function, `GetField()`. This function will prompt the user to choose among the allowed fields (again, a single letter code should suffice for input). The function should return an integer identifier.

These "field identifiers" are shared with the `RefCard` objects. It would probably be best if they were defined in the `RefCard.h` header file.

After the code to get a field identifier, the `DoView()` function will need a loop in which it accesses each `RefCard` from its collection and tells it to print the contents of the chosen field. The code will be along the following lines:

Loop processing each card in the collection

```

for(int i = 1; i <= fNumCards; i++) {
    cout << i << "\t";
    RefCard *r = (RefCard*) fStore.Nth(i);
    r->PrintField(field);
    cout << endl;
}
  
```

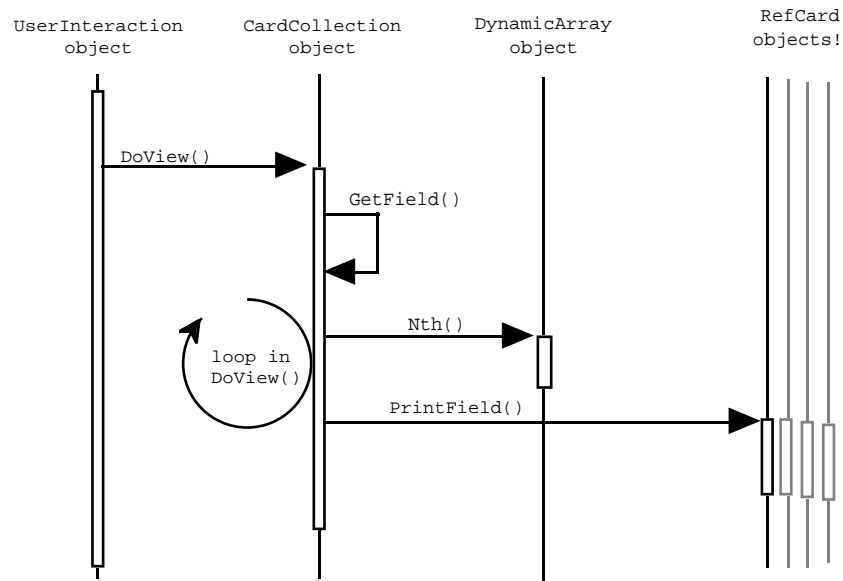


Figure 22.7 Object interactions resulting from a User's "view" command.

Results from scenario:

New functions: `CardCollection::DoView()`, `CardCollection::GetField()`, and `RefCard::PrintField()`.

Example: Searching for cards with a given word

The actual interactions among objects when doing a simple search would be somewhat similar to those shown for "view".

Simple form of search

Once again, the `CardCollection` object would need to get the user to identify the field (data member) of interest; though there is a change here in that system is supposed to allow a search on "any" of the four fields in addition to searches on individual fields. (Probably, function `GetField()` should be extended so that it has a parameter that indicates whether "any" is an acceptable input. This parameter would be "false" if `GetField()` were being called to find a field for "view" but "true" in the case of a search.)

After identifying the field of interest, the `CardCollection` object would have to get the user to enter the string (the name or word that is to be found in the search). It could then have a loop asking each `RefCard` in turn to "check a field" (rather than print a field as in "view").

Coding the search for a word in a character array like a title is not a problem. There is a function in the string library that does exactly this (function `strstr()`).

Checking a field for a string

The main problem is that a simple search would list all the cards that matched. The specification required something like the "Find..." and "Find Again..." commands that you get with most word processors.

More complex search required by specification

We will have to have two functions:

```
CardCollection::DoFind()
CardCollection::DoFindAgain()
```

The `DoFind()` function will do the more elaborate work. It will prompt the user to identify the field and the string, and organize a search for the first card that matches. It will have to arrange for the `CardCollection` object to store state information defining the field, string, and position in the collection where the first matching card was found.

DoFind()

The `DoFindAgain()` function should check that there is a search in progress (string defined, position reached last time set etc). If that is OK, it should have a loop that works through successive cards until another match is reached. It should then update the state data so that another later call will continue the search.

DoFindAgain()

This implies that we need some additional data members in class `CardCollection`:

```
char      *fFindString;
int       fFindPos;
int       fSearchField;
```

These are going to have to be set appropriately in the constructor.

Finalising the design

The design process using scenarios (and supplementary top-down functional decomposition when you get a complex member function) have to continue until you fully can characterize your classes. You need to get to the point where you can write down a complete class declaration and provide short (one sentence) descriptions of each of the member functions.

For this example, we eventually get to the following:

```
class RefCard {
public:
    RefCard();
    /*
    Disk i/o
    */
    void    ReadFrom(fstream& s);
    void    WriteTo(fstream& s) const;
    /*
    Communication with user
```

Class RefCard declaration

```

        */
        void    Show() const;
        void    GetData();
        void    Change();
        int     CheckField(int fieldnum, char *content);
        void    PrintField(int fieldnum);
    private:
        char    fAuthors[kNAMEFIELDSIZE];
        char    fTitle[kNAMEFIELDSIZE];
        char    fJournal[kNAMEFIELDSIZE];
        char    fKeywords[kNAMEFIELDSIZE];
        char    fAbstract[kNAMEFIELDSIZE];
        short   fFirstPage;
        short   fLastPage;
        short   fIssue;
        short   fYear;
};

```

As required by the specification, the character arrays used to store titles are fixed sized.

*Class RefCard,
function
specifications*

Constructor

Initialize all character arrays to blank string, all integers to zero.

ReadFrom, WriteTo

Read data from (write data to) a text file.

Show

Print identifying field labels and contents of all data members.

GetData

Prompt user and then read values for each data member in turn.

Change

Get user to identify data member to be changed (enter a letter), output details of current contents of that data member, read replacement data.

CheckField

Integer argument identifies data member to be checked, string argument is content to be searched for using `strstr()` function from string library.

PrintField

Integer argument identifies data member that is to be output.

```

class CardCollection {
public:
    CardCollection();
    /*
    Attaching to file
    */
    int    Open(const char filename[]);
    void    Close();
    /*
    Main commands
    */
    int    NumCards() const;
    void    ReportStatus() const;
    void    AddCard();
    void    DeleteCard(int cardnum);
    void    ShowCard(int cardnum) const;
    void    ChangeCard(int cardnum);
    void    DoFind();
    void    DoFindAgain();
    void    DoView();
private:
    int    GetField(int anyallowed);
    void    Load();
    void    Save();
    int    fNumCards;
    char    *fFileName;
    fstream    fFile;
    DynamicArray fStore;
    char    *fFindString;
    int    fFindPos;
    int    fSearchField;
};

```

***Class CardCollection
declaration***

Constructor

Initialize data fields, fNumCards is zero, fFindString is NULL etc.

***Class CardCollection
function
specifications***

Open

Either open existing file and call Load (), or create a new file. If can't do either report error.

Close

Call Save () to get cards to file, then clean up deleting cards.

NumCards

Report number of cards in current collection.

ReportStatus

Print out name of file, and details of number of cards.

AddCard

Create a card, get it to obtain its data from the user, add to collection, update count of cards owned..

DeleteCard

Remove identified card from collection then delete it, update count of cards owned.

ShowCard

Get pointer to chosen card from dynamic array, tell card to show itself.

ChangeCard

Get pointer to chosen card from dynamic array, tell card to interact with user to get changes.

DoFind

Use `GetField()` (specifying any as OK) to allow user to pick field, then prompt for string, then loop through cards asking them to check the specified field-string combination. Stop as soon as get a match, asking the card to show itself and recording, in state data, the point where match was found. If no matches, warn user and discard the search information

DoFindAgain

Check that there is a search in progress. If not, warn user. If there is a search, continue from current point in collection checking cards until either find a match or there are no more cards.

GetField

Use simple "menu" selection system to let user pick a field.

Load

Read number of cards in file, then loop creating cards, letting them read their data from file, and adding them to collection.

Save

Write number of cards to file, then let each card write itself to file.

***Class UserInteraction
declaration***

```
class UserInteraction {
public:
    UserInteraction();
```

```

    void    Initialize();
    void    Run();
    void    Terminate();

private:
    char    GetCommand();
    int     PickACard();
    void    Help();
    void    DoDelete();
    void    DoChange();
    void    DoShow();

    CardCollection *fCollection;
};

```

Constructor

Create a CardCollection object

Initialize

Ask the user to enter a filename, tell the CardCollection object to try to open that file.

Run

Loop getting and processing user commands until a "quit" command is entered.

Terminate

Tell the CardCollection object to close up, then get rid of it.

GetCommand

Get single character command from user.

PickACard

Prompt user for a card number, make certain that it is in range (ask CardCollection object how many cards there are to chose from).

Help

Print explanation of available commands.

DoDelete, DoShow, DoChange

Use PickACard() to get the card number then call the corresponding member function of the CardCollection object.

***Class UserInteraction
function
specifications***

As well as developing the class declarations and function summaries, you might want to produce "design diagrams" for the classes like that shown in Figure 19.1.

File (module) structure of program

Another design choice you now have to make is how these components will be organized in files. Here it would be appropriate to have the files:

CC.h, CC.cp	CardCollection class
D.h, D.cp	the dynamic array
main.cp	the little main() driver routine
RefCard.h, RefCard.cp	RefCard class
UI.h, UI.cp	UserInteraction class

"Header dependencies"

Because the objects of these classes interact so much, there are lots of inter-dependencies in the code. For example, when compiling the code in `UserInteraction.cp`, the compiler has to be able to check that all those requests to the `CardCollection` object are valid. This means that it will have to have read the `CardCollection.h` file before it compiles `UserInteraction.cp`.

Direct (uses) dependencies

It is often useful to draw up a diagram showing the interrelationships between files so that you remember to `#include` the necessary headers. Figure 22.8 illustrates some of the relations for this program.

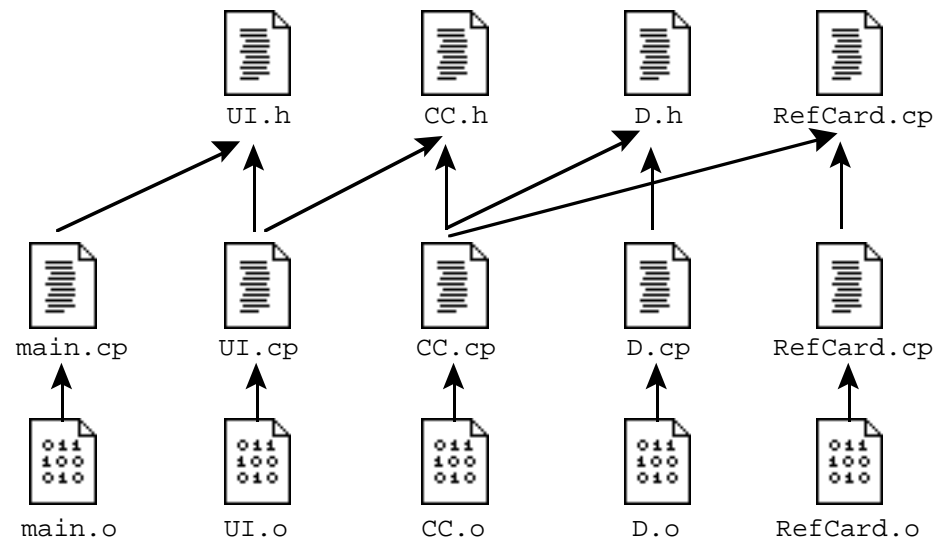


Figure 22.8 Illustration of direct file dependencies in the RefCard program.

The code in `CardCollection.cp` uses `RefCard` functions and `DynamicArray` functions so it must `#include` both these header files. Class `UserInteraction` needs the declaration of class `CardCollection` so it must include `CC.h`; similarly `main` needs `#include UI.h`.

There are further dependencies. Although `UserInteraction` doesn't make any direct use of a `DynamicArray` it does need to know about this class. Similarly, the `main` program needs to know about the existence of class `CardCollection`. These dependencies result from the presence of data members in classes.

Additional (indirect) dependencies

Class `CardCollection` has a `DynamicArray` as a data member. When the compiler is trying to process the code in `UI.cp` it will need to work out the size of a `CardCollection` object and so will need to have already read the file `D.h`.

When processing `main.cp`, the compiler would need to be reassured that the `CardCollection*` pointer data member in class `UserInteraction` was referring to a defined type of structure. It wouldn't need to read the complete declaration of class but would need to have the class declared (i.e. the `UI.h` file would need to include a declaration like `"class CardCollection;"` as well as the full declaration of class `UserInteraction`).

You have to sort out all the additional indirect dependencies so that you can `#include` all required files. If you forget some, you will get numerous compiler error messages. The actual messages may be obscure – complaints like "Illegal cast from int to void*", or "Undefined function" for a function that you know is defined. Compilers can get quite confused when they encounter references to classes that they haven't seen declared. If you get such odd compiler errors, start by checking that you included the right header files and in the correct order (as `CardCollection` uses `DynamicArray`, the `#include "D.h"` should come before the `#include "CC.h"` to make certain that class `DynamicArray` has been declared before an instance of the class gets used).

Compiler error messages relating to missing headers are often obscure

22.1.2 Implementation

As always, if the design is complete then the implementation is trivial.. The `main` program has already been given.

The header file `UI.h` would contain the class `UserInteraction` declaration shown earlier. The presence of the `CardCollection*` data member means that the header file would have to contain a declaration the existence of class `CardCollection`. The implementation file, `UI.cp`, would need to `#include` `stdlib`, `ctype`, `UI`, and `CC`.

UserInteraction

The constructor simply creates the `CardCollection`. The `Initialize()` member function will get a filename from the user and ask the `CardCollection` to open the file; if this fails the program terminates (this call to `exit()` lead to the need to `#include` `stdlib.h`). Function `Terminate()` gets the `CardCollection` to close up before it is deleted.

```
UserInteraction::UserInteraction()
```

constructor

*Initialization, getting
filename*

```

    {
        fCollection = new CardCollection;
    }

void UserInteraction::Initialize()
{
    char    buff[100];
    cout << "Enter name of file with cards : ";
    cin >> buff;
    int status = fCollection->Open(buff);
    if(status < 0) {
        cout << "Sorry, can't open (or create) file."
              << "Giving up" << endl;
        exit(1);
    }
    fCollection->ReportStatus();
}

void UserInteraction::Terminate()
{
    fCollection->Close();
    delete fCollection;
}

```

The highlighted statements are typical of the "hey object, do action" calls that pervade the code.

The complete version of the Run() member function is:

*Interaction loop for
user commands*

```

void UserInteraction::Run()
{
    int    done = 0;
    cout << "Enter commands, ? for help" << endl;
    for(; !done; ) {
        char command = GetCommand();
        switch(command) {
        case 'q' :    done = 1; break;
        case '?' :    Help(); break;
        case 'a' :    fCollection->AddCard(); break;
        case 'c' :    DoChange(); break;
        case 'd' :    DoDelete(); break;
        case 's' :    DoShow(); break;
        case 'f' :    fCollection->DoFind(); break;
        case 'g' :    fCollection->DoFindAgain(); break;
        case 'v' :    fCollection->DoView(); break;
        default :
            cout << "Command " << command <<
                  " not recognized" << endl;
        }
    }
}

```

Menu based programs like this should try to include a help function that explains their options:

```
void UserInteraction::Help()
{
    cout << "Commands are : " << endl;
    cout << "\ta  Add a new card." << endl;
    ...
    ...
    cout << "\tv  View one field from all cards" << endl;
    cout << "\tq  Quit" << endl;
}
```

Built in help

Functions DoDelete(), DoChange(), and DoShow() are very similar; DoDelete() can represent them all:

```
void UserInteraction::DoDelete()
{
    int which = PickACard();
    if(which < 1)
        return;
    fCollection->DeleteCard(which);
}
```

DoDelete() and similar functions

They all use the auxiliary function PickACard() to get a valid card number:

```
int UserInteraction::PickACard()
{
    if(fCollection->NumCards() < 1) {
        cout << "There aren't any cards so you can't do"
              "that now." << endl;
        return 0;
    }
    cout << "Which card? (Enter number in range 1 to " <<
          fCollection->NumCards() << ") : " << endl;
    int aNum;
    cin >> aNum;
    if(!cin.good()) {
        cout << "??";
        cin.clear();
        cin.ignore(100, '\n');
        return 0;
    }

    if((aNum < 1) || (aNum > fCollection->NumCards())) {
        cout << "Invalid input, ignored." << endl;
        return 0;
    }
    return aNum;
}
```

Auxiliary input functions

The other auxiliary input function, `GetCommand()`, is used by `Run()` to get a single input character. The call to `ignore()` removes ("flushes") any other input remaining in the stream (this avoids problems when a user does something like type a command name, e.g. 'view,' instead of just a command letter 'v').

```
char UserInteraction::GetCommand()
{
    char ch;
    cin >> ch;
    ch = tolower(ch);
    cin.ignore(100, '\n');
    return ch;
}
```

class *CardCollection* As usual, constructor for class `CardCollection` should initialize its data members. It is not absolutely necessary to initialize all members. For example, we know that there is no possibility that the `fFileName` field would be used before being set, so it is acceptable to leave this uninitialized. On the whole, you should be cautious and initialize everything!

```
constructor CardCollection::CardCollection()
{
    fFindString = NULL;
    fFindPos = -1;
    fNumCards = 0;
}
```

Opening and closing the associated file Function `Open()` really consists of two parts. The first deals with the case of an existing file; this uses the auxiliary `Load()` function to get the data. The second part deals with the case where it is necessary to create a new file. Function `Close()` needs a call to `Save()`, the actual file closing action, and some tidying up operations. (The tidying up isn't comprehensive; we don't get rid of the array of pointers owned by the `DynamicArray`. The next chapter covers "destructor" – special automatically invoked tidy up routines. Class `DynamicArray` really needs a "destructor" function to tidy away its pointer array.)

The initialization step of class `UserInteraction` also involves a call to a `CardCollection::ReportStatus()` function. This function is not shown. It would simply print out the name of the file associated with the `CardCollection` and the number of cards that it contained.

```
int CardCollection::Open(const char filename[])
{
    /* Keep copy of file name */
    fFileName = new char[strlen(filename)+1];
    strcpy(fFileName, filename);
}
```

```

    fFile.open(fFileName, ios::in | ios::out | ios::nocreate);
    if(fFile.good()) {
        Load();
        return 0;
    }

    fFile.open(fFileName, ios::in | ios::out);
    if(!fFile.good())
        return -1;
    return 0;
}

void CardCollection::Close()
{
    Save();
    fFile.close();
    // Should get rid of data structures like the cards
    for(int i = fNumCards; i > 0; i--) {
        RefCard *r = (RefCard*) fStore.Remove(i);
        delete r;
    }
    if(fFindString != NULL) delete [] fFindString;
}

```

The Load() function reads the number of cards then loops creating cards and getting them to read their data. Each card gets "appended" to the dynamic array. You would need some error checking on input operations even if, as here, it is limited to stopping the program if something seems to have gone wrong. *Reading the cards*

The Save() function is not shown. It just writes details of the size of the collection, then gets each member card to write itself to the file.

```

void CardCollection::Load()
{
    fFile.seekg(0);
    fFile >> fNumCards;
    fFile.ignore(100, '\n');
    for(int i=0; i < fNumCards; i++) {
        RefCard *r = new RefCard;
        r->ReadFrom(fFile);
        if(!fFile.good()) {
            cout << "Sorry, file must be corrupt, "
                 "giving up." << endl;
            exit(1);
        }
        fStore.Append(r);
    }
}

```

AddCard()

The function `AddCard()` is a simple implementation of the ideas shown in the interaction diagram shown in Figure 22.4:

```
void CardCollection::AddCard()
{
    RefCard *r = new RefCard();
    r->GetData();
    fStore.Append(r);
    fNumCards++;
}
```

DeleteCard()

The delete operation involves removing a chosen card from the `DynamicArray` `fStore`, deletion of the object, and updating of the member count. (Strictly, the member count is redundant as we could always ask the `DynamicArray` for the number of items that it holds).

```
void CardCollection::DeleteCard(int cardnum)
{
    RefCard *r = (RefCard*) fStore.Remove(cardnum);
    delete r;
    fNumCards--;
}
```

**ShowCard(),
ChangeCard()**

The `ShowCard()` and `ChangeCard()` functions are similar. A pointer to the chosen card is obtained from the `DynamicArray`. Then the card is told to perform an action. Function `ShowCard()` illustrates both:

```
void CardCollection::ShowCard(int cardnum) const
{
    RefCard *r = (RefCard*) fStore.Nth(cardnum);
    r->Show();
}
```

**Identifying a field for
search or display**

Function `GetField()` has to prompt the user for an indication of the search field (taking into account whether "any" is an allowed response). It can return a -1 value for an illegal input of a positive integer constant identifying a valid field. The constants like `kAUTHORFIELD` will have to be defined in `RefCard.h`.

```
int CardCollection::GetField(int anyallowed)
{
    cout << "Which data field?" << endl;
    cout << "a Authors, t Title, c Content (abstract), "
           "k Keywords" << endl;
    if(anyallowed) cout << "x for any of these" << endl;
    char ch;
    int result = -1;
    cin >> ch;
    switch(ch) {
```

```

    case 'x': if(anyallowed) result = kANYFIELD; break;
    case 'a': result = kAUTHORFIELD; break;
    case 't': result = kTITLEFIELD; break;
    case 'c': result = kABSTRACTFIELD; break;
    case 'k': result = kKEYWORDFIELD; break;
    }
    return result;
}

```

Function `DoView()` had better check that there are some cards to view. If there are, it needs to use `GetField()` to let the user pick a field to be displayed ("any" is not allowed here). If the user enters a valid field selection, then each card in the collection gets told to print the contents of that field. (Loops like the `for(;;)` loop here run from 1 to N because the `DynamicArray` uses 1 for the first element, a departure from the normal C convention of zero-based arrays).

Viewing a collection

```

void CardCollection::DoView()
{
    if(fNumCards < 1) {
        cout << "No cards to view!" << endl;
        return;
    }

    int field = GetField(0);
    if(field < 0) {
        cout << "Invalid field choice, ignored." << endl;
        return;
    }
    for(int i = 1; i <= fNumCards; i++) {
        cout << i << "\t";
        RefCard *r = (RefCard*) fStore.Nth(i);
        r->PrintField(field);
        cout << endl;
    }
}

```

Function `DoFind()` has some similarities to `DoView()`. It starts by getting the field selection ("any" is allowed); then prompts for and reads the search string. It makes a copy of this string, the copy is saved in the `CardCollection`'s state data. This makes it possible to resume the search if requested.

Finding a particular card

Once all the search data are entered, the function loops getting and checking successive cards from the collection. If a match is found, the card is displayed, and the function returns. The data member `fFindPos` is used to control the loop and it retains the position where a match is found.

If there is no match in the entire collection, a warning is displayed and the search data are tidied away.

```

void CardCollection::DoFind()

```

```

Identifying the search field      {
                                     fSearchField = GetField(1);
                                     if(fSearchField < 0) {
                                         cout << "Invalid field choice, ignored." << endl;
                                         return;
                                     }
Getting the search string        char buff[50];
                                     cin.ignore(100, '\n');
                                     cout << "Enter search string : ";
                                     cin.getline(buff, 49, '\n');

Saving a copy of the search string if(fFindString != NULL) delete [] fFindString;
                                     fFindString = new char[strlen(buff) + 1];
                                     strcpy(fFindString, buff);

Search loop                      for(fFindPos = 1; fFindPos <= fNumCards; fFindPos++) {
                                     RefCard *r = (RefCard*) fStore.Nth(fFindPos);
                                     int match = r->CheckField(fSearchField, fFindString);
                                     if(match) {
Process a successful match         r->Show();
                                     return;
                                     }
                                     }

Report failure and tidy up        cout << "No Matches" << endl;
                                     delete [] fFindString;
                                     fFindString = NULL;
                                     fFindPos = -1;
                                     }

```

Find again

The function `DoFindAgain()` must check that there is a search in progress and that we haven't already reached the end of the collection. If further search is meaningful, the function loops looking for the next matching card. As in `DoFind()`, a match results in display of a card and return from the routine while failure to get a match results in a warning.

The code should work even if the user does things like delete cards between successive find again operations.

```

void CardCollection::DoFindAgain()
{
    if(fFindPos < 0) {
        cout << "You've got to do a 'Find' before "
              "'Find Again'" << endl;
        return;
    }
    if(fFindPos >= fNumCards) {
        cout << "No more matches" << endl;
        delete [] fFindString;
        fFindString = NULL;
        fFindPos = -1;
        return;
    }
}

```



```

    }

    for(fFindPos++; fFindPos<= fNumCards; fFindPos++) {
        RefCard *r = (RefCard*) fStore.Nth(fFindPos);
        int match = r->CheckField(fSearchField, fFindString);
        if(match) {
            r->Show();
            return;
        }
    }
    cout << "No more matches" << endl;
    delete [] fFindString;
    fFindString = NULL;
    fFindPos = -1;
}

```

The constructor should initialize the strings to null and the numeric fields to zero:

class RefCard

```

RefCard::RefCard()
{
    fAuthors[0] = '\0';
    fTitle[0] = '\0';
    fJournal[0] = '\0';
    fKeywords[0] = '\0';
    fAbstract[0] = '\0';
    fFirstPage = fLastPage = fIssue = fYear = 0;
}

```

The functions for transfer to and from file are simple. The file used here is a text file rather than a binary file. It should be possible to read and edit such a file using a word processor. (You may find that you have to use one of the utility programs on your system to change the "file type" before you can edit these files. In some IDEs, data files written by programs are created with non-standard types.)

File transfer

The final call to `ignore()` in `ReadFrom` is there to consume the newline after the last of the numbers. If you don't consume this newline, the next `ReadFrom()` operation will fail as it will encounter the newline character when trying to read the "authors" string and so get out of phase.

```

void RefCard::WriteTo(fstream& s) const
{
    s << fAuthors << endl;
    s << fTitle << endl;
    s << fJournal << endl;
    s << fKeywords << endl;
    s << fAbstract << endl;
    s << fFirstPage << " " << fLastPage << endl;
    s << fIssue << " " << fYear << endl;
}

```

```

void RefCard::ReadFrom(fstream& s)
{
    s.getline(fAuthors, kNAMEFIELDSIZE-1, '\n');
    s.getline(fTitle, kNAMEFIELDSIZE-1, '\n');
    ...
    s >> fFirstPage >> fLastPage >> fIssue >> fYear;
    s.ignore(100, '\n');
}

```

GetData() Function `GetData()` is similar to `ReadFrom()` except that it prompts for each data item before reading:

```

void RefCard::GetData()
{
    cout << "Enter data for paper:" << endl;

    cout << "Author(s) : ";
    cin.getline(fAuthors, kNAMEFIELDSIZE-1, '\n');
    ...
    ...
    cout << "Enter page range,\n";
    fFirstPage = GetNumber("\tfirst page: ");
    fLastPage = GetNumber("\tlast page: ");
    fIssue = GetNumber("Issue # : ");
    fYear = GetNumber("Year : ");
}

```

Numeric values are required for the page numbers, year etc. We need code that prompts for a number, and then checks that it gets a (positive non zero) number. If the user enters something that is not a number, we have to clear the error condition, remove all characters from the input buffer and prompt again. Obviously, this code should be as a subroutine, we don't want the code duplicated for each numeric field. Hence, we have a `GetNumber()` routine.

This could be made a member function of `RefCard` but it doesn't really seem to belong. Instead it can be a filescope function defined in the `RefCard.cp` file:

<p><i>Auxiliary, non-member function</i> GetNumber</p> <p><i>Output prompt and read value</i></p> <p><i>If bad input, clear flag and buffer</i></p>	<pre> static int GetNumber(char *prompt) { int val = 0; int ok = 0; while(!ok) { cout << prompt; cin >> val; if(cin.good()) ok = 1; else { cout << "???" << endl; cin.clear(); cin.ignore(100, '\n'); } } } </pre>
--	--

```

    }
    return val;
}

```

Function `Show()` just outputs field labels and values:

Show()

```

void RefCard::Show() const
{
    cout << "Author(s)\t: " << fAuthors << endl;
    cout << "Title\t\t: " << fTitle << endl;
    ...
    cout << "Abstract\t: " << fAbstract << endl;
}

```

Function `Change()` has to prompt for a field identifier, then it should display the contents of the field before reading a new value:

Change()

```

void RefCard::Change()
{
    cout << "Changing card:" << endl;
    cout << "Select a (Authors), t (Title), j (Journal)" << endl;
    cout << "\tk (Keywords), c (Content, abstract)" << endl;
    cout << "\ty (Year), v (Volume), p (Page range)" << endl;
    char command;
    cin >> command;
    command = tolower(command);
    cin.ignore(100, '\n');
    switch(command) {
case 'a':
        cout << "Authors currently " << fAuthors << endl;
        cout << "Enter correction : ";
        cin.getline(fAuthors, kNAMEFIELDSIZE-1, '\n');
        break;
        ...
        ...
case 'v':
        fIssue = GetNumber("Volume");
default:
        cout << "???" << endl;
    }
}

```

Function `PrintField()` simply outputs the contents of a chosen field:

PrintField()

```

void RefCard::PrintField(int fieldnum)
{
    switch(fieldnum) {
case kAUTHORFIELD: cout << fAuthors; break;
case kTITLEFIELD: cout << fTitle; break;

```

```

        case kKEYWORDFIELD: cout << fKeywords; break;
        case kABSTRACTFIELD: cout << fAbstract; break;
    }
}

```

While function `CheckField()` uses `strstr()` to check whether a given string is contained in any of the string data fields:

```

CheckField()    int RefCard::CheckField(int fieldnum, char *content)
                  {
                    if((fieldnum == kANYFIELD) || (fieldnum == kAUTHORFIELD))
                      return (NULL != strstr(fAuthors, content));
                    if((fieldnum == kANYFIELD) || (fieldnum == kTITLEFIELD))
                      return (NULL != strstr(fTitle, content));
                    ...
                    ...
                    return 0;
                  }

```

Function `strstr(const char *s1, const char *s2)` "looks for a substring within a string" returning a `char*` pointer to the first place where substring `s2` can be found in `s1` (or `NULL` if it doesn't occur). It is one of the standard functions in the string library. You can find more details using your IDE's help system (or the separate "ThinkReference" program for the Symantec IDE).

22.2 INFOSTORE

Specification

The InfoStore program is to allow a user to maintain collections of news articles. These articles are to be indexed according to the "concepts" that they contain. Each concept can be represented by an arbitrary number of keywords. The vocabulary of concepts and keywords is to be user definable; there can be a fixed maximum on the number of concepts allowed (at least a few hundred). (The concept-keyword scheme is the same as in the example in Section 18.3. For example you might have the concept "ape" matched by any of a set of keywords that includes "ape", "apes", "chimps", "chimpanzee", "bobo", "gorilla",)

The "InfoStore" program is to allow a user to:

- create an initial vocabulary of keywords and concepts;
- add keywords and concepts to an existing vocabulary;
- add the text of news article to the system;

- perform a search for articles with search requirements specified by entry of required and prohibited keywords.

22.2.1 Initial design outline for InfoStore

Preliminaries

For a program like this, there are a few design decisions that come before the stage where we start thinking about the objects that might be present.

We have to decide how to store the permanent data in disk files. We now have three different kinds of data.

Files for permanent data

There are the actual news articles. These are just blocks of text and, as in the program in Section 18.3, we can store them all in a single file provided we can separate them (use null, '\0', characters) and we know where each begins.

Data files

Next, there are going to be index entries, one for each article in the main articles file. As in the example in Section 18.3, these are going to consist of a set of bits (bit value 1 implies presence of a particular concept in an article, value 0 implies absence), and a "file address" (record of where an article starts in the main file). We will need a second file to hold these index entries.

Index files

Finally, we need to have some form of "vocabulary file". The earlier program used a fixed "compiled-in" vocabulary; but that is too restrictive. We now need a file that contains keywords and numbers. The number associated with a keyword will identify the concept to which it belongs. So, if for example, concept 25 is the system's representation of "ape", the vocabulary file should contain entries including "ape 25", "chimpanzee 25", "gorilla 25". (It is assumed that keywords cannot be associated with multiple concepts; so you can't have concept 25 "ape" and concept 95 "gangster" with gorilla appearing as both "gorilla 25" and "gorilla 95".) The vocabulary file can be a simple text file with lines that contain keyword concept number pairs. The program can build any more elaborate vocabulary structures from this input.

Vocabulary files

Each "information store" thus needs three files. We can keep them together by allowing the user to define a "basename" for the files, e.g. "travel" or "science", and create the files with distinguishing suffixes (e.g. "travel.dat", "travel.ndx", and "travel.vcb").

Common "base name" for files

If the user extends the vocabulary then any articles already stored will have to be re-indexed. The program should deal with this automatically.

Affect of changing the vocabulary

The program will load all the vocabulary data and build a look-up structure that can be used to check whether a word corresponds to one of the concepts. The articles and index entries will be left in the files. When a search is being performed, each index entry in turn will get loaded into memory; the user's query can be represented by a Bitmap structure similar to that used an index entry. Articles that match a search request will just be copied character by character from the data file to the output; they don't have to be loaded completely into memory. When an article is being added, it gets

Memory resident data and disk-based data

copied character by character from input file to the data file. Again, the complete article never has to be in memory. While adding an article, the program will build up a new index entry and store, temporarily, the current word from the article as was done in the program in Section 18.3

Thus, the only large memory resident structures will be those used for the vocabulary.

Program operation

Normally, the user would be expected to start by building up a vocabulary. This would probably be done in several separate runs of the program with each run adding a few more keywords. Once a basic vocabulary had been established, articles might start to be entered. The user would want to identify a text file with an article. The system should copy the contents of this file to its data file, at the same time building up an index record that would then get written to the index file. In any particular run of the program, the user might add a few articles, and make one or two searches.

Finding objects

Once you have resolved preliminaries like how the files might be organized and how the program would be used, you can start postulating possible objects.

"Obvious objects"

There are a few "obvious" objects. As in the last example, the system will probably use a "UserInteraction" object that organizes most of the processing. Once again, the main program will do little more than create this `UserInteraction` object and tell it to "run".

UserInteraction object

The `UserInteraction` object will accept commands like "expand vocabulary", "add an article", "do a search". The `UserInteraction` object might need to get some additional information but would deal with most requests by passing them on to other objects in the system.

Vocabulary object

Another plausible candidate is a "Vocabulary" object. Something has to own the keywords and concept numbers. This something has to have organized some fast lookup mechanism (hash table or tree). This something has to get the words in from the vocabulary file, and back out to the file whenever extra words have been added during a run of the program. Words have got to be looked up and concept numbers returned. There certainly seems to be a group of related data, and larger number of operations on these data, that could be packaged up in an object that is an instance of some "Vocabulary" class.

InfoStore object

Another possible candidate is an "Infostore" object. Once again, "something" has to own the three files, and the string that represents the basename of the files. This "something" can organize opening of the files, keeping track of the length of the data files and the number of entries in the index file. It can forward user requests for changes to the vocabulary on to the vocabulary object and perform other organization roles.

Words?

There are several other possible objects. Maybe "Words" should be objects. A Word object could own a string and a concept number. But there don't seem to be many

tasks for a Word to perform. The Vocabulary object might employ Words, but the rest of the program would almost certainly just be working with character strings. Words can be left for now. They might reappear later on but they don't seem to have sufficient of a role to justify consideration during preliminary design.

News articles? No. They certainly don't do anything (other things scramble through the text of a news article). Although articles may exist as an "objects" in the data file, they never really exist in memory. Most processing involving articles works on them one character at a time.

News Articles?

How about an "ArticleFinder" object? You could argue that this "owns data" e.g. it will own the user's query (this will take the form of a set of required and set of prohibited concept numbers, both represented as bit maps). It will check this query against each entry in the index file and print those articles that match.

ArticleFinder?

However, an ArticleFinder object doesn't really have the right feel. Objects are primarily things that are created dynamically and remain around for a reasonable length of time (like the RefCards in the last example). If you think how the program would work, the InfoStore object would create an ArticleFinder each time the user made a search request. This ArticleFinder would do its stuff. Then it would be destroyed. It doesn't have the right kind of lifetime. It really is "just a function" and the data that it "owns" are really just automatic variables that it uses. So discount ArticleFinder – not an object. The InfoStore can take on responsibility for finding matching articles; it will just use a group of member functions to do this.

Don't go confusing a function (verb) with an object (noun)!

Lists? Use as needed. The program will hold in memory just single index records, single queries. Most of the data are left in the files. The only large collection will be the words of the vocabulary. We can add a list (or dynamic array) if needed.

Lists and collection classes

Bitmap? Yes. We can reuse class Bitmap from Chapter 19. An index entry will contain a bitmap.

Bitmap

IndexEntry? Plausible. If we define an IndexEntry class we have a place to put related behaviours like checking for matches with other Bitmaps that represent queries. An IndexEntry will own a Bitmap and a long integer to record the location of an article. It can read itself and write itself to file. It can be built up by telling it to set concept bits. Class IndexEntry seems to earn its way.

IndexEntry

The "Vocabulary" object will create some hash table data structure. But this probably would not exist as an independently defined class. If you had a class library with a "reusable HashTable" you might proceed differently. However, reusable HashTables are not common. HashTable structures tend to be purpose built for specific applications with minor variations to adapt to special needs.

HashTable?

The most plausible classes are illustrated in Figure 22.9. As in the RefCards example, most are "fuzzy blob" classes because we haven't really defined what any do or own. This time, class Bitmap is the exception. Once again, because it is a known reusable class it can be shown with firmly defined boundaries.

In this example, the preliminary classes shown in Figure 22.9 did become the final classes used in the implementation. However, it would not be unusual for changes to be made during the later more detailed design steps.

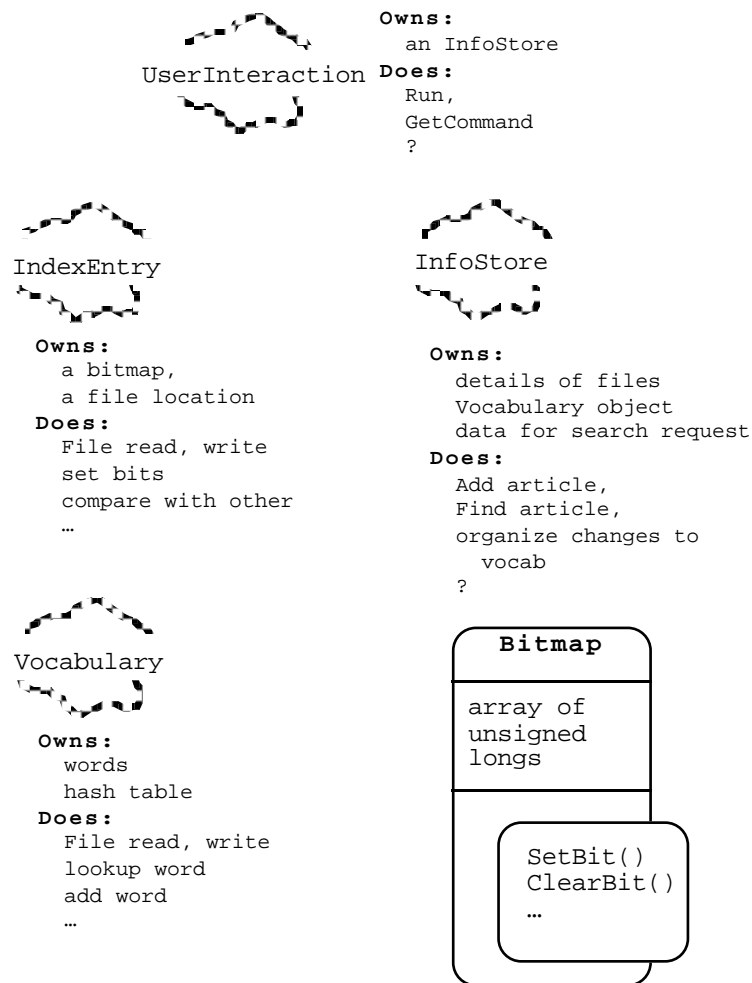


Figure 22.9 First idea for classes for InfoStore example.

22.2.2 Design and Implementation of the Vocabulary class

As in the previous example, our next task is to elaborate these initial ideas of classes. Once again, this will involve using scenarios to examine possible interactions among objects. Although the processing steps involved may be more elaborate, the actual patterns of interaction are more limited in this example.

This example does illustrate another aspect of the use of objects. An object-based approach often makes it possible to design and implement parts of a program in total

isolation. Once the parts have been made to work, you fit them together to make a whole. It is just the same process as we have been doing with "reusable classes" like `DynamicArray`, except that the classes developed will only be "used" in a test context and then "reused" in the final program product.

Separate development of parts is of great practical importance. Most programs are built by teams. It is obviously more practical for individual team members to work on clearly separate parts. Separate development helps even if a single programmer is developing the system. Separate development means that the programmer is writing, and testing, two or more simple programs rather than one larger more complex program.

The "Vocabulary" object represents a fairly substantial part of the program. It owns quite a lot of data in varied forms. It is certainly going to own the actual vocabulary entry items (concept number and keyword string) and a hashtable structure allowing fast lookup. It may own other data. It is going to have to do things like add words to its collection and lookup words to see if they are already in the collection. However, it doesn't need services of other objects and probably it is only the `InfoStore` object that ever requests actions by the `Vocabulary` object. Thus, it is a good candidate for separate development.

*Focus on isolable
Vocabulary object*

We have to start by examining scenarios that focus on use of the `Vocabulary` object. Together these will define the "public interface" for a `Vocabulary` class. Once this has been defined, separate development is possible. The design and implementation of the `Vocabulary` class can be completed and verified using a little test program that exploits the same public interface.

So, what does a `Vocabulary` object (or, more briefly, a `Vocab` object) get asked to do? Firstly, there will be file input and output. Figures 22.10 and 22.11 illustrate plausible scenarios. Activity will start with the user telling the `UserInteraction` object to "open" an `InfoStore`. This will result in an "open store" request being passed to the `InfoStore` object. We can ignore most of the activity of `InfoStore::OpenStore()` for now; it will involve getting a "base name" from the user and then opening of all three files. (Inconsistencies such as only one or two files existing will terminate the program.) The scenario in Figure 22.10 picks up at the point where the files have all been opened successfully. The `InfoStore` object will ask the `Vocabulary` object to load its data from the already opened vocab-file.

*Scenarios for file
input and output*

The first item in the file might as well be an integer giving the number of words in the vocabulary. There could be a few thousand words. We want to allow for a few hundred concepts and each concept might be represented by several different keywords in the news articles. So we can expect hundreds, possibly thousands, of keyword/concept number pairs.

We will need something to store these data. As noted earlier, we might use instances of some class `Word`. But at least for the present it appears that we could make do with a simple struct like the `VocabItem` used in the earlier simpler version of the program.

VocabItem struct

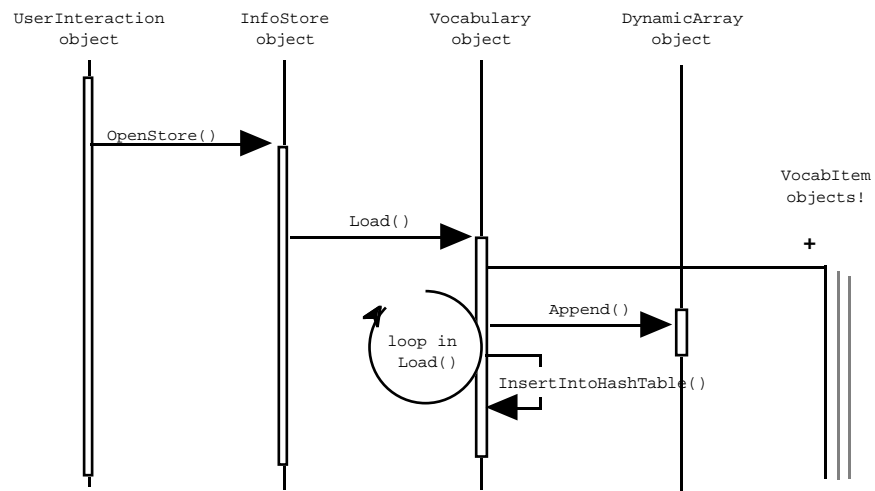


Figure 22.10 Object interactions while loading a vocabulary file.

VocabItems A `VocabItem` struct will have a `char*` pointer and an integer. Function `Vocabulary::Load()` can have a loop in which it creates `VocabItems`. If these are just simple structures, then the `Vocab` object better do the work of reading their data. It reads a word into a temporary buffer. A new character array can be created (this operation is not shown in Figure 22.10) and the word gets copied into the new array. The character array's address can be stored in the new `VocabItem` along with the integer concept number also read from file.

A dynamic array to store the VocabItems

We have to store the complete collection of words. We will have to provide a service like "list all the words with their concept numbers", and "list all words associated with concept number ...". So, we will be working sequentially through the collection.

For this collection we can obviously use an instance of class `DynamicArray`; it can be a data member in the `Vocab` object. Once a `VocabItem` has been created and its data fields filled in, it can be added to the dynamic array. This is shown in Figure 22.10.

Separate hash array

We also need the fast lookup version. This will be a hash table of pointers to the same `VocabItems`. The array used for hash address will have to be larger than the maximum number of keywords we expect. We had better arrange to have it created in the constructor for class `Vocab`. Once we have read a `VocabItem`, we have to add it to the hash table in addition to the main dynamic array. In Figure 22.10, this is illustrated as the call to `InsertIntoHashTable()`. Obviously, this is a non-trivial process. Later it will get broken down using a "top down functional decomposition" approach. This more detailed design step will add some other private member functions to class `Vocab`.

Figure 22.11 illustrates those parts of a `Close()` operation that involve the `Vocab` object. It will loop, getting the `VocabItems` from its array and writing their contents to file. It would be worthwhile checking whether the vocabulary has been changed; there

is no need to spend time rewriting the file if the existing file is still valid. Class `Vocab` should have some boolean or integer indicator, `fChanged`, that gets set when words are added. If this is not set, the write to file step can be omitted. It is possible that another lot of data might get loaded into the same `Vocab` object, so once the current data are finished with the arrays should be cleared and the existing `VocabItems` should be deleted.

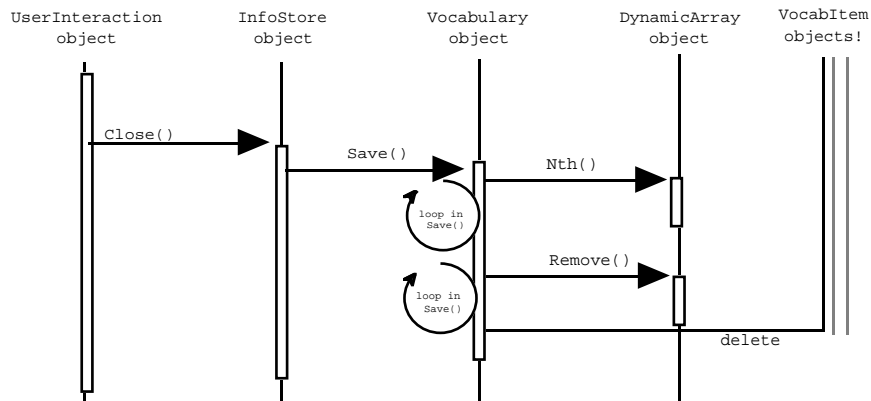


Figure 22.11 Object interactions while saving to a vocabulary file.

So far, we seem to have:

```

class Vocab {
public:
    Vocab(?);
    /*
    File support
    */
    void Load(fstream& in);
    void Save(fstream& out);
    /*
    Status
    */
    void ReportStatus() const;

    ...
private:
    struct VocabItem {
        short fCNum;
        char* fWord;
        ... // Maybe other data
    };
    void InsertIntoHashTable(VocabItem*);

```

```

        int      fNumWords;
        int      fNumConcepts;
        int      fChanged;

        VocabItem **fHashTable;

        DynamicArray fTbl;
        ...
    };

```

Here it has been assumed that `VocabItem` is essentially a private struct used only by class `Vocab`. It would be useful for the `Vocab` object to maintain counts of the number of keywords and concepts that it had defined; hence data members like `fNumWords`. A `ReportStatus()` member function might be useful. It could print out details of the number of keywords and the number of concepts.

The hash table is an array of pointers to `VocabItems`. Since it is a dynamically allocated structure located somewhere in the heap, the type of `fHashTable` is pointer to (an array of) pointer(s) to `VocabItems`, or `VocabItem**`. (This is as discussed previously in Chapters 20 and 21.)

***What else might a
Vocab object do?***

Now that we can load a `Vocab` object from a disk file, what should we do with it?

The `UserInteraction` object will offer a basic menu of commands like "do search", "do addition of article", and "do vocabulary operations". These will result in requests to the `InfoStore` object to do the search, or addition, or organize vocabulary options. In the case of vocabulary modifications, the `InfoStore` object will probably present the user with a kind of submenu. The commands will be things like:

***InfoStore command
options involving a
Vocab object***

- **Concept**
Add new concept by giving the first keyword. (The `Vocab` object should allocate a new concept number).
- **Word**
Add another word for an existing concept. The user would then have to enter the concept number and the new keyword (system should check the number is in range).
- **List**
List all concepts or words. There would have to be another prompt to find whether the user want a printout showing all words, or just those associated with a particular concept number (or, maybe, a list arranged by concept number).
- **Test**
Test whether word is associated with a concept. This is really just a "lookup" operation on a word provided by the user.

The keyword/concept-number idea is not very well defined. Really, the only difference between adding a concept and adding an alternative keyword is that for a "concept" the

Vocab object selects the next possible concept number whereas for a keyword the user has to specify an existing concept number.

The Vocab object could check that the user doesn't request a new concept or keyword and then enter a word that already exists. However, the user can do this check anyway by using the Test option to check a word before trying to enter it.

These different tasks involve three basic patterns of interactions among the objects. None seem sufficiently elaborate to merit an object interaction diagram.

The first pattern is a "creational pattern" which will get used for the Concept and Word commands. The InfoStore object gets the necessary data from the user (a text string, and in the case of the Word command the input will also contain a concept number). The InfoStore invokes member functions `Vocab::AddNewConcept()` or `Vocab::AddExtraWord()`. These functions build a new `VocabItem`, and then add it to both the hash table and the dynamic array. The functions had better return integer error codes. There will be a limit on the number of concepts; there may be other constraints that could cause these operations to fail.

*Creational patterns
for VocabItems*

The InfoStore object had better check the validity of any concept number entered with an extra keyword. It can pass the number entered by the user to the Vocab object to check; this could be done before any call to `AddExtraWord()`. So class Vocab had better provide a `CheckConceptNum()` member function.

The additional parts of class Vocab's public interface identified by considering these interactions are:

```
int      Vocab::CheckConceptNum(int conceptnum) const;
int      Vocab::AddExtraWord(const char aWord[], int conceptnum);
int      Vocab::AddNewConcept(const char firstWord[]);
```

The various suboptions under List would all be handled by the InfoStore object asking the Vocab object to perform a specialized listing operation. The Vocab object would work using a loop that looks at successive `VocabItems` from its dynamic array and prints the appropriate ones. Listing all words is easy; the loop in a `ListWords()` function simply prints every `VocabItem`. A list of all `VocabItems` associated with a given concept number requires only an extra test and the same basic loop structure; this can be handled using a `Vocab::ListConcept(int conceptnum)` member function. Listing the keywords for each concept in turn could be handled by a function, `ListAllConcepts()`, that has a loop working through all concept numbers calling the `ListConceptNum()` function for successive numbers. Of course this means running through the array many times. This may be a bit costly, but there is no need to look for more efficient schemes, like sorting by concept number, because the execute-time is going to be determined almost entirely by the printing processes. More elaborate schemes would just add code but not produce any noticeable change in performance.

*Iterating through the
dynamic array*

The "listing" options require the following additional functions:

```
void      Vocab::ListConcept(int conceptnum) const;
void      Vocab::ListAllConcepts() const;
```

```
void          Vocab::ListWords() const;
```

***Looking up a word in
the hashtable***

The Test command will require that the Vocab object identify the concept number associated with a given keyword. It will need a function like:

```
void          Vocab::IdentifyConcept(const char aWord[]) const;
```

***Other interactions
involving Vocab
object***

This will print details of the concept number, or report that the keyword is not known.

The Vocab object will also be used when generating index records for new articles and creating queries. Probably, both these requests will come from the InfoStore object. They require the same function, it will be given the word to look up, and will return an integer concept number (a code like -1 could be used to indicate that the word is not defined).

```
int           Vocab::Lookup(const char aWord[]) const;
```

Of course, at least one Vocab object gets created so a constructor had better be defined. It would probably be useful if the program could specify a default size for the vocabulary. Other arguments for the constructor might be identified later.

***Completing the
public interface***

If all the interactions involving Vocab objects have been identified, then we have completely characterised the public interface for the class.

```
class Vocab {
public:
    Vocab(int VocabSize);
    /*
    File support
    */
    void    Load(fstream& in);
    void    Save(fstream& out);
    /*
    Status
    */
    void    ReportStatus() const;
    int     CheckConceptNum(int conceptnum) const;

    /*
    Checking and adding words
    */
    int     Lookup(const char aWord[]) const;
    int     AddExtraWord(const char aWord[], int conceptnum);
    int     AddNewConcept(const char firstWord[]);
    /*
    Getting info on concepts
    */
    void    ListConcept(int conceptnum) const;
    void    ListAllConcepts() const;
    void    IdentifyConcept(const char aWord[]) const;
```

```

        void    ListWords() const;
private:
    ...
};

```

It is now possible to complete the design, implementation and testing of this class. There is no need to build an `InfoStore` class. A simple interactive test program can easily be written to exercise the various member routines.

Detailed design, implementation, and test of class Vocab

What remains?

The remaining design work will involve minor choices on detailed representation of the data and probably some further functional decomposition for the more elaborate member functions.

The index entries for the file are to use class `Bitmap`. This allows chosen bits to be tested. It stores 512 bits, numbered 0...511. The number of concepts should be limited to 512. Internally, concept numbers should be represented by integers in the range 0...511 but it would probably be best if the user saw these as 1...512 (this means that there will have to be conversions on input and output).

Concept numbers

The argument for class `Vocab`'s constructor can be used to define the initial size for the dynamic array (this will involve a minor C++ feature not previously illustrated). By default, the dynamic array only grows by 5 elements; that will be too small, a larger increment should be defined, maybe 25% of the initial size.

Array sizes

We don't want the hash table becoming full. It will probably be worthwhile defining a maximum size for the vocabulary (some multiple of the initial size) and refusing to add words once this size is reached. If the hash table is made slightly larger, we can guarantee that it never becomes full. The entries in the hashtable will either be `NULL` or pointers to `VocabItems` created in the heap and also referenced from the main dynamic array.

Most of the functions should be straightforward. The listing functions just involve loops accessing successive `VocabItems` in the dynamic array. The hashtable functions (`Test()`, `Lookup()`, `AddExtraWord()`, and `AddNewConcept()`) will use code similar to that illustrated earlier in Chapters 18 and 20.

Hash keys are going to have to be computed for the various strings. The code will be the same as that illustrated previously (Section 18.2.1) but it should now take the form of a private member function for class `Vocab`:

Another private member function

```

    unsigned long Vocab::HashString(const char str[]) const;

```

Although a good hashing function, it is relatively expensive because of its loop through all the characters in a string. It might be worth saving the hash keys in the `VocabItems`.

This would avoid the need to recompute the keys for all the words when the files are reloaded. Consequently, we might redefine `VocabItem` as follows:

```
struct Vocab::VocabItem {
    unsigned long fKey;
    short fCNum;
    char* fWord;
};
```

Implementation

Constructor The constructor has the following form

```
Vocab::Vocab(int vocabsiz) : fTbl(vocabsiz, vocabsiz/4)
{
    fNumWords = fNumConcepts = fChanged = 0;
    fTblSize = 5*vocabsiz;
    fMaxWords = 4*vocabsiz;
    fHashTable = new VocabItem* [fTblSize];
    for(int i=0; i < fTblSize; i++)
        fHashTable[i] = NULL;
}
```

The bit in bold illustrates the extra feature of C++ – initialization of data members that are instances of classes with their own constructors.

*Data members that
are instances of other
classes*

We have already had classes that had data members that were instances of other classes; after all, in the `RefCards` example, the `CardCollection` object had a `DynamicArray`. But in the previous examples we have been able to rely on default constructors. The default constructor for a `DynamicArray` gives it ten elements, so `CardCollections` start with an array of size ten. With `Vocab` objects, we want the `DynamicArray` to start at some programmer specified size, so we can't just leave it to the default constructor.

*Data members that
have their own
constructors*

C++ allows you to pass arguments to the constructors for any data members that require such initialization. These data member constructors get executed prior to the body of the class's own constructor. They have to be specified as shown above. They are separated from the argument list of the constructor by a colon (:), and are listed before the opening { bracket of the body.

In this example, the `fTbl` data member (the `DynamicArray`) is initialized using the `DynamicArray(size, increment)` constructor.

The hash table is made 25% larger than the maximum number of words. Thus it can never be more than 80% full and so the simple linear probing mechanism will work quite satisfactorily. The element of the hash table need to be initialized to `NUL`.

File I/O

The `Load()` and `Save()` functions are as follows:

```
void Vocab::Load(fstream& in)
```



```

{
    fChanged = fNumWords = fNumConcepts = 0;
    in >> fNumWords;
    if(in.eof()) { in.clear(); return; }
    if(fNumWords > fMaxWords) {
        cout << "Problems with file.  Seems to have too many"
              "words." << endl;
        exit(1);
    }
    in >> fNumConcepts;
    if(fNumConcepts >= kMAXCONCEPTS) {
        cout << "Bad data in file." << endl;
        exit(1);
    }

    for(int i=0; i < fNumWords; i++) {
        VocabItem *v = new VocabItem;
        if(!in.good())break;
        in >> v->fKey >> v->fCNum;
        char lword[100];
        in >> lword;
        v->fWord = new char[strlen(lword) + 1];
        strcpy(v->fWord, lword);
        fTbl.Append(v);
        InsertIntoHashTable(v);
    }
    if(!in.good()) {
        cout << "Sorry, problems reading vocab file. "
              "Giving up" << endl;
        exit(1);
    }
}

```

Checks on file contents

Loop creating VocabItems

Add VocabItems to both arrays

The file used for the vocabulary is really a simple text file. Consequently, a user may edit it with some standard editor or word processor. The `Load()` routine has to do some checking to validate the input. The main part of `Load()` is the loop where the new `VocabItem` structs are created, their data are read in, and they are then added to both the dynamic array and the hash table.

Function `Save()` is called when the program has finished using the current data in the `Vocab` object. If changed, the updated data should be saved to file. All existing data structures have to be cleaned out.

```

void Vocab::Save(fstream& out)
{
    if(fChanged != 0) {
        out << fNumWords << " " << fNumConcepts << endl;
        for(int i=1; i<= fNumWords; i++) {
            VocabItem *v = (VocabItem*) fTbl.Nth(i);
            out << v->fKey << " " << v->fCNum << " " <<
                v->fWord << endl;
        }
    }
}

```

```

    }
}
for(int j =0; j < fTblSize; j++) fHashTable[j] = NULL;
for(j = fNumWords; j> 0; j--) {
    VocabItem *v = (VocabItem*) fTbl.Remove(j);
    delete [] v->fWord; // Get rid of the string
    delete v;           // and the structure
}
fChanged = fNumWords = fNumConcepts = 0;
}

```

Tidying up is hard work!

**Simple access
functions**

Functions like `ReportStatus()` (print details of number of words and concepts), and `CheckConceptNum(int num)` (check value against `fNumConcepts`) are all trivial so their code is not shown.

Listing functions

The listing functions are generally similar, `ListConcept()` is shown here as a representative. After checking its argument, it has a loop that works through successive elements of the `DynamicArray` (request `fTbl.Nth(i)` returns the *i*-th element). The `VocabItem` accessed via the array is checked, and if it is associated with the required concept its string is printed. (The code assumes that the `conceptnum` argument is defined in the internal `0..N-1` form rather than the `1...N` form used in communications with the user.)

```

void Vocab::ListConcept(int conceptnum) const
{
    if((conceptnum < 0) || (conceptnum >= fNumConcepts)) {
        cout << "No such concept number." << endl;
        return;
    }

    /*
    As output for user, change to user-numbering of concepts
    (1...N) rather than internal 0...N-1.
    */
    cout << "Words mapped onto concept: #" << (conceptnum+1)
         << endl;
    for(int i = 1; i <= fNumWords; i++) {
        VocabItem *v = (VocabItem*) fTbl.Nth(i);
        if(v->fCNum == conceptnum) cout << v->fWord << endl;
    }
}

```

Vocabulary extension

The functions that extend the vocabulary are:

```

int Vocab::AddExtraWord(const char aWord[], int conceptnum)
{
    if(fNumWords == fMaxWords)
        return 0;
    fNumWords++;
    fChanged = 1;
}

```

```

VocabItem *v = new VocabItem;
v->fCNum = conceptnum;
v->fKey = HashString(aWord);
v->fWord = new char[strlen(aWord) + 1];
strcpy(v->fWord, aWord);

fTbl.Append(v);
InsertIntoHashTable(v);
return 1;
}

int Vocab::AddNewConcept(const char firstWord[])
{
    if((fNumConcepts == kMAXCONCEPTS) ||
        (fNumWords == fMaxWords)) return 0;
    AddExtraWord(firstWord, fNumConcepts);
    fNumConcepts++;
    return 1;
}

```

*Calculate and save
hash key*

If the vocabulary is not already full, `AddExtraWord()` creates a new `VocabItem`, fills it with the given data and adds it to the hash table and the dynamic array. Function `AddNewConcept()` uses `AddExtraWord()` while providing the concept number; the count of concepts is then updated. These functions mark the vocabulary as changed so that a later call to `Save()` will result in transfer to disk.

The code for the hash functions can be based on that in earlier examples. A *Hash functions* representative function from this group is `InsertIntoHashTable()`. This simply reworks earlier hash table insertion code.

```

void Vocab::InsertIntoHashTable(VocabItem* v)
{
    unsigned long k = v->fKey;
    k = k % fTblSize;
    int pos = k;
    int startpos = pos;

    for(;;) {
        if(fHashTable[pos] == NULL) {
            fHashTable[pos] = v;
            return;
        }
        /*
        Shouldn't get duplicates.
        Maybe should report this as an error.
        */
        if(0 == strcmp(v->fWord, fHashTable[pos]->fWord))
            return;

        pos++;
        if(pos >= fTblSize)

```

```

        pos -= fTblSize;
    if(pos == startpos) {
        /*
        OOPS! This should never happen.
        The hash table should never become full; entry
        of words is supposed to be restricted so max
        80% full.
        */
        cout << "Error in hashing functions of Vocab."
              << endl;
        exit(1);
    }
}
}

```

Test

With class `Vocab` defined, we need a test program. This will simply be another of those small interactive program where the user is given a menu of commands like "add a word", "list concept" and so forth. Small auxiliary routines will prompt the user for any necessary data and invoke the appropriate operations on an instance of class `Vocab`.

This "scaffolding" code belongs with the `Vocab` class and should be considered as part of the class's documentation.

Part of the `main()` function of this test program is:

```

...
#include "Vocab.h"

Vocab gv(1000);

...

int main()
{
    fstream testfile("testfile", ios::in | ios::out);
    gv.Load(testfile);
    gv.ReportStatus();
    int done = 0;
    for(; ! done ; ) {
        cout << "Enter command : ";
        char ch;
        cin >> ch;
        ch = tolower(ch);
        switch(ch) {
case 'q':  done = 1; break;
case 'c':  AddConcept(); break;
case 'l':  List(); break;
case 'w':  AddWord(); break;

```

```

case 't': TestWord(); break;
case '?': cout << "Commands are" << endl;
          cout << "\tl List all concepts or words." << endl;
          ...
          cout << "\tq Quit" << endl;
          break;
default:
          cout << "? Unrecognized command " << ch << endl;
          break;
          }
    }
    gv.Save(testfile);
    testfile.close();
    return 0;
}

```

An example of the auxiliary functions needed is:

```

void AddWord()
{
    cout << "Enter concept number for which you want an"
          "additional keyword : ";

    int n;
    cin >> n;
    /*
    convert from user 1..N representation to 0..N-1
    */
    if(!gv.CheckConceptNum(n-1)) {
        cout << "That number doesn't correspond to a "
              "defined concept." << endl;
        return;
    }

    cout << "Enter extra keyword : ";
    char aWord[40];
    cin >> aWord;
    if(gv.AddExtraWord(aWord,n-1))
        cout << "OK, added." << endl;
    else cout << "Sorry, vocab. full, can't add." << endl;
}

```

22.2.3 Other classes in the InfoStore program

With class `Vocab` completed and tested, development of the rest of the system could resume. As illustrated in Figure 22.12, the situation has changed. Now class `Vocab` can be treated as a predefined "reusable" component.

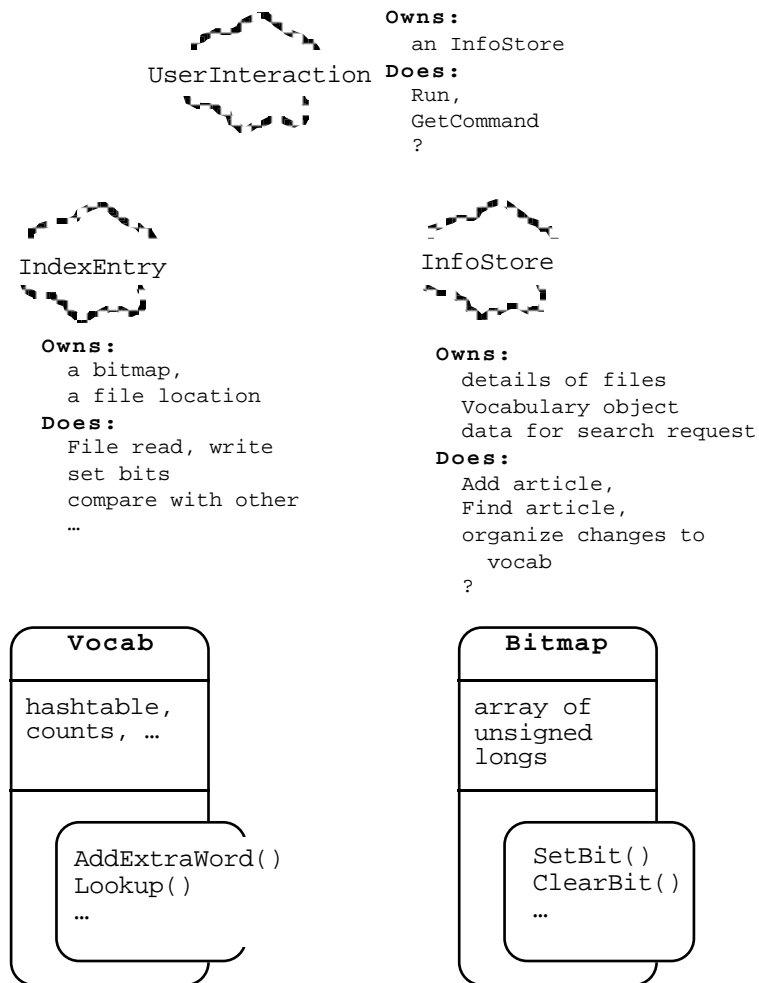


Figure 22.12 Revised model for classes.

Class UserInteraction

Class `UserInteraction` won't present many problems. This class, and the `main()` function, will be similar to the corresponding parts of the previous example. We might as well use the same main program. So we can expect class `UserInteraction` to have the same public interface and parts of the same implementation structure as last time:

```
class UserInteraction {
public:
    UserInteraction();
```

```

        void    Initialize();
        void    Run();
        void    Terminate();
    private:
        void    Help();
        char    GetCommand();
        // and maybe some changed stuff!
        ...
};

```

Function `Run()` will present the user with a menu of options and function `Help()` will provide some explanation of the options. There will have to be some additional private auxiliary member functions that deal with the top level commands (like "Add article") by doing some validity checks, or getting some data, and then calling an appropriate member function of the `InfoStore` object.

This time, class `UserInteraction` is going to have to own an `InfoStore` object rather than a `CardCollection` object. It doesn't much matter whether it has an `InfoStore` data member or an `InfoStore*` data member. If a pointer data member is used, as in the `RefCards` example, the data object can be created in the `UserInteraction` constructor. However, this time we will make it an actual `InfoStore` object.

***InfoStore data
member***

In the `RefCards` example, function `UserInteraction::Initialize()` opened the data file, while `Terminate()` closed the file. You could only move from one `RefCard` collection to another by exiting and restarting the program. It might be worth making operations a little more general.. We could have "Open" and "Close" commands in the menu offered in `Run()`. If no set of files is open, the only commands available would be "Open" and "Quit". If a set of files is open, the commands would be "Quit", "Add article", "Search", "Change Vocabulary" and "Close". (The prompt for a command should indicate the system's state; the `UserInteraction` object should probably have a data member, `fState`, to indicate its open/closed state.)

The code for `UserInteraction::Run()` will have to be along the following lines:

```

void UserInteraction::Run()
{
    int    done = 0;
    cout << "Enter commands, ? for help" << endl;
    for(; !done; ) {
        if(fState == 0) cout << "(closed) > ";
        else cout << "(open)    > ";
        char command = GetCommand();
        switch(command) {
        case 'q' :    done = 1; break;
        case '?' :    Help(); break;
        case 'a' :    DoAdd(); break;
        case 'c' :    DoClose(); break;
        case 'o' :    DoOpen(); break;
        case 's' :    DoSearch(); break;

```

```

        case 'v' :      DoVocab(); break;
        default :
            cout << "Command " << command << " not"
                  "recognized" << endl;
            }
    }
}

```

The auxiliary functions like `DoSearch()` will defer most work to the `InfoStore` object:

```

void UserInteraction::DoSearch()
{
    if(fState == 0) {
        cout << "You have to have an Information Store open"
              << endl;
        cout << "if you want to search!" << endl;
        return;
    }
    fStore.DoSearch();
}

```

The remaining functions should all be easy to code. There may not be anything to do in `Terminate()` and `Initialize()`. They got included as a move towards a standard `UserInteraction` class. These two examples in this chapter with their similar structure and `UserInteraction` classes provide, in a very limited way, a model for the more elaborate programs that can be built with the framework class libraries introduced in Part V. In those libraries, you will find standardized classes that accept commands from a user and route these to appropriate data objects.

Classes `InfoStore` and `IndexEntry`

`InfoStore`

The real work in this program is done by `InfoStore` along with its helpers `Vocab` and `IndexEntry`. Class `InfoStore` has to deal with a variety of requests from the `UserInteraction` object. We know that the `Vocab` object doesn't need to make requests to the `InfoStore` object, and it is pretty unlikely that the `IndexEntry` objects will need to ask anything of the `InfoStore`. Consequently, the class's public interface will be determined entirely by the needs of the `UserInteraction` object. We can therefore sketch it in now:

```

class InfoStore {
public:
    InfoStore(int VocabSize = 1000);
    int      OpenStore();
}

```



```

        void    Close();

        void    ChangeVocab();
        void    AddArticle();
        void    DoSearch();
    private:
        ...
        fstream    fIndexFile;
        fstream    fVocabFile;
        fstream    fDataFile;
        Vocab      fVocab;
        long       fNumArticles;
        ...
};

```

The public functions are just those called from member functions of `UserInteraction`. The private data members shown have already been identified. The `InfoStore` object is supposed to own the files, since they are used for both input and output they will be `fstream` objects. The `InfoStore` need a `Vocab` object; it seems likely that it should have a count of the number of articles in the files.

There will be many additional private auxiliary member functions. The extra member functions will get identified as the known functions, like `AddArticle()`, are developed using "top-down functional decomposition".

Function `OpenStore()` has to either successfully open a set of three existing files, or if none exist it should create a new set of three files. It should terminate the program if it cannot get a complete set of files. If it is able to open existing files, then this function should have a call asking the `Vocab` object to load the `VocabItems` as previously discussed.

OpenStore()

Function `ChangeVocab()` will end up very much like the little test program written to check class `Vocab`! It will have a similar prompting function that gets user commands ("add word", "list concept", etc), and similar auxiliary functions to organize things like the listing of concepts.

ChangeVocab() – already been written (more or less)!

The `InfoStore` object will have to have its own flag data member to indicate whether the vocabulary has been changed recently. The `Vocab` object already keeps track of whether it has been changed at all since its data were loaded, and uses this to determine whether to save its data when it gets closed. The `InfoStore` object has rather different concerns. It must prevent searches of files, or closing of files, if the index entries haven't been updated to match any changes in the vocabulary.

Another VocabChanged flag?

An `InfoStore` object should set its flag data member when it asks its `Vocab` object to add a word. It should check this flag when asked to search or close files. If its flag is set, it should first go through all the index records and articles in its files. It has to repeat the indexing process by reading each article from the `InfoStore`'s main data file, updating the index entry and rewriting the updated index entry to the index file. The process is similar to that involved in the addition of a new article as described in more

Responsibility for updating index entries if vocabulary changes

detail below. Once the articles have been reindexed, the `InfoStore` object can clear its version of the "vocabulary changed" flag.

AddArticle Function `AddArticle()` will start by prompting the user for the name of a file. It must then copy the content of the article to the end the data file, at the same time building up an index entry. Each word read during this copying process must be checked; any that are keywords should cause the new index entry to be updated. The interactions involved in this process are outlined below.

DoSearch Function `DoSearch()` had better start by checking whether the file contains any articles. If the data files do contains some articles, this member function has to get the query from the user. Queries consist of three parts. First there is the set of required concepts. A loop will be used to get the user to enter required keywords (the function should warn about any words entered that aren't keywords); these will be used to build up a bitmap of required concepts. The second data item is the minimum number of concepts that must match. The third item would be another set of keywords (really concept numbers) that should not be present. Once the query has been assembled, it must be checked against `IndexEntry`s read from the index file. Matches result in display of articles. Again, the interactions are outlined in more detail below.

IndexEntry

The previous slightly simpler version of this program, in Section 18.3, used some ad hoc structures to represent index entries. Although an index entry is really a composite involving a bit map and a file location, the previous representation had an array of unsigned longs and a separate long integer data element. There was no packaging of the operations on these data, the code was scattered through the other functions. With classes, we can do better.

We can now have class `IndexEntry`. This will package the data and related functions. We know some of the things an `IndexEntry` must do. An `IndexEntry` is going to have to transfer itself to/from file; this is going to be a binary transfer as they are supposed to be represented as fixed size blocks of bits in the file. An `IndexEntry` gets built up – literally bit-by-bit. When an article is processed, the concepts it contains are identified and the `IndexEntry` is told to set the corresponding bit in its bitmap. It also has to be told to note the location of an article.

These known behaviours provide a first outline for class `IndexEntry`:

```
class IndexEntry {
public:
    IndexEntry();
    void    Load(fstream& in);
    void    Store(fstream& out);
    void    SetBit(int bitnum);
    void    SetLocation(unsigned long where);
    ...
private:
```

```

        Bitmap      fBits;
        unsigned long fLocation;
};

```

The `Bitmap` data member will be an instance of the class developed in Chapter 19; those `Bitmap` objects deal with things like clearing all bits, setting individual bits, and writing bit data to the file. So a lot of an `IndexEntry`'s work can be delegated to the `Bitmap` object that it owns.

Class `IndexEntry` will have some additional responsibilities related to checking matches with search queries. These will be added later when they have been more clearly identified.

Adding or re-indexing articles

Figure 22.13 illustrates the interactions among objects when an article is added or its index entry updated to reflect changes in the vocabulary.

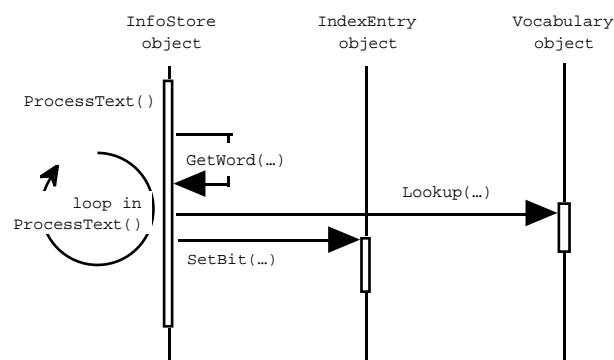


Figure 22.13 Object interactions while adding articles to information store.

Class `InfoStore` will have a `ProcessText()` private member function that deals with the detail of the indexing operations. It will have to take as arguments the input file (could be a text file with a single article or the existing data file), an `IndexEntry` to update, and a flag to indicate whether it is copying the data from the file or simply updating the `IndexEntry`. The function that calls `ProcessText()` had better set the input file so that it is at the correct position for reading (this would be the start of a new text file, but at the location of an existing article the index entries are being updated).

If it is a new article that is being added to the collection, the `IndexEntry` should already have been initialized with all its bits zero, and its `fLocation` field set to contain the current end point of the data file (the place where the copy of the new article will start). Since the vocabulary always expands, the only changes will be new words and

concepts. Consequently, there is no need to reinitialize the `IndexEntry` when an article that is being re-indexed; the existing bits in the index bit map won't change, maybe a few more bits will get set.

Function `ProcessText()` will contain a loop that gets words from the file; class `InfoStore` had better have another private member function `GetWord()`.

```
while(GetWord(...) {
    int concept = fVocab.Lookup(aWord);
    if(concept>=0)
        article_ndx.SetBit(concept);
}
```

The `GetWord()` function will be similar to the function in Section 18.3; it will need an extra flag argument to indicate whether the characters are to be copied as well as built up into words. The "words" get filled into a character array that would be a local variable of `ProcessText()`.

Each "word" would have to be checked. Hence the call to the `Lookup()` function of the `Vocab` object. If `Lookup()` returns a valid concept number, the `IndexEntry` will have to be told to set this corresponding bit (the call to `SetBit()`).

The `ProcessText()` function gets executed in two circumstances. First the `InfoStore` object may have been told to add an article. The operations needed in this situation would be:

<i>Coding a new article</i>	<i>Initialize a new IndexEntry, zeroing out its bit map</i> <i>Make certain data file is positioned so that writes append data</i> <i>after all existing data</i> <i>Note current end position in IndexEntry</i> <i>Call ProcessText (specify data to be copied, input from new</i> <i>text file)</i> <i>Write a null character to the data file to mark the end of the</i> <i>article.</i> <i>Write the new index entry at the end of the index file.</i> <i>Update record of number of articles.</i>
------------------------------------	---

Alternatively, the `InfoStore` object may be fixing up all its index records before doing a search or closing the files. The operations need in this context would be:

<i>Fixing up existing index entries after a vocabulary change</i>	<i>for each article in collection</i> <i>Load existing index entry</i> <i>Find where related article starts</i> <i>Position data file so character read operations begin at</i> <i>articles</i> <i>Call ProcessText(specifying no copying, input from data</i> <i>file)</i> <i>rewrite updated index entry in its original position in</i> <i>the index file</i>
--	---

Class `InfoStore` will need additional private member functions that organize these operations.

```
void      InfoStore::CodeArticle(fstream& infile);
void      InfoStore::FixupRecords();
```

Function `CodeArticle` would be called from the main `AddArticle()` function that opens a user-specified file with the additional news article. Function `FixupRecords()` would be called the `Close()` function (to make certain that set of index, vocabulary, and article files are consistent), and before searches. Obviously, it would start by testing the `InfoStore::fVocabChanged` flag variable to determine whether there was any need to update the files (and it would clear this flag variable once the files had been remade).

Searches

As noted earlier, the `DoSearch()` function would have to start by making certain that a search operation was valid (file contains articles, all files consistent). Then it would build up the query structure. Finally, it would have a loop that involved checking each `IndexEntry` from the index file against the query; articles corresponding to matching queries would be printed. The code for `DoSearch()` would be along the following lines:

```
if(fNumArticles == 0)                                     InfoStore::
    report search not worthwhile, and return                DoSearch()

call FixupRecords to make certain files are consistent

Build a Bitmap that represents the set of concepts required

Find minimum number of matches required

Build a second Bitmap representing unwanted concepts

Position index file at start

for each entry in file do
    load index entry

    test loaded entry against Bitmap representing excluded
        concepts, if any present then don't further check

    count number of matching concepts in index entry and
        Bitmap representing the required concepts

    if match at least the required number
        print article starting at location in index entry
```

As usual, this outline implicitly identifies a number of auxiliary functions; these will all become private member functions of class `InfoStore`.

There would have to be two auxiliary functions that build `Bitmap` objects. A `Bitmap` for a query must have at least one bit set; so a `GetQuery()` function would need to have a loop that kept prompting the user to enter "search terms"; something along the following lines:

Building a bit map that represents a query	<pre> Bitmap e; prompt for search terms do { Input a word key = fVocab.Lookup(word); if(key < 0) cout << "(not used)" << endl; else { cout << "(Concept #" << key+1 << ")" << endl; e.SetBit(key); } } while (count of concepts in query < 1 or user specifies another keyword); </pre>
---	--

As shown, the function should identify the concept numbers associated with the words entered so that the user will know whether a proposed query involves multiple concepts or whether all the keywords entered happen to map onto a single concept number. Two more simple private member functions would appear useful – something to input a word, and something to get a "yes/no" response to a prompt like "Do you want to enter another word?". When a `Bitmap` representing a query has been built, another auxiliary function can prompt the user for the minimum number of matches (it should check the query `Bitmap`, if there is only one bit set then there is no need to ask the user).

The function to get excluded words would be generally similar. The loop structure would be changed slightly because an empty `Bitmap` is valid in this context.

The extra functions need for class `InfoStore` could be:

```

Bitmap      InfoStore::GetQuery();
Bitmap      InfoStore::GetExclude();
int          InfoStore::GetRequiredNum(const Bitmap&);
void         InfoStore::InputWord(char aWord[]);
int          InfoStore::YesNo();

```

**Extensions to class
*IndexEntry***

Searches necessitate some extra member functions in class `IndexEntry`:

```

int          IndexEntry::CheckNoCommonElements(const Bitmap& bad);
int          IndexEntry::CountCommonElements(const Bitmap& good);
unsigned long IndexEntry::Location() const;

```

The first function verifies that there are no bits in common between the `IndexEntry`'s own `Bitmap` and that given as an argument; this is used to filter out articles with

excluded keywords. The second checks the number of bits that are in common. Finally, class `IndexEntry` must provide read access to the details of the location its news article. These functions are trivial to implement as all the necessary bit manipulations are provided by class `Bitmap`.

22.2.4 Final class design for the InfoStore program

The final designs for classes `InfoStore` and `IndexEntry` are shown in Figures 22.14 and 22.15.

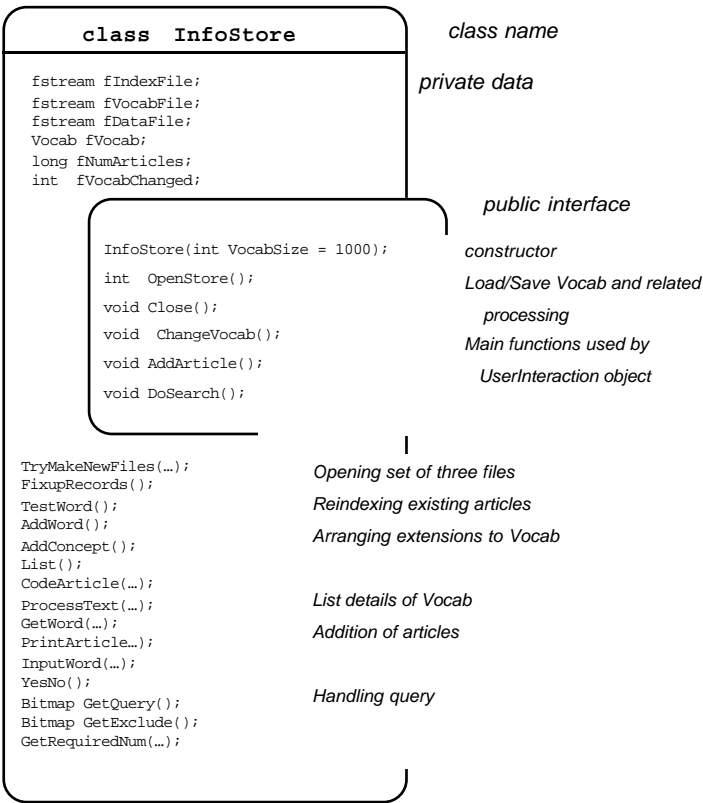
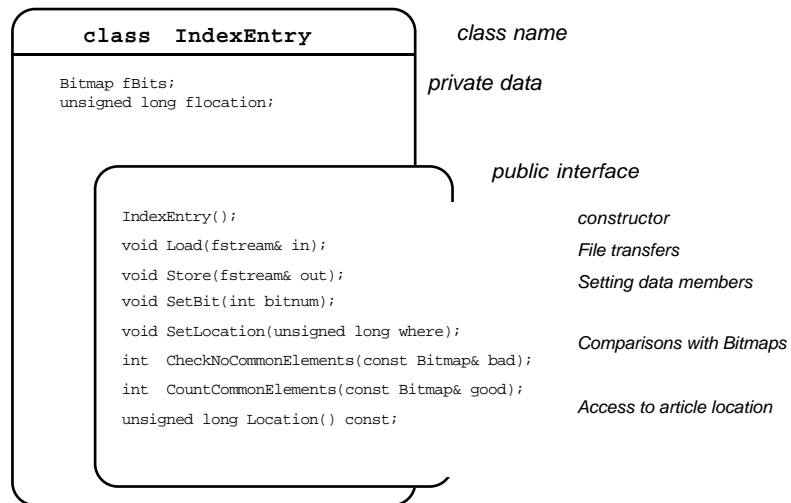


Figure 22.14 Final design for class `InfoStore`.

Figure 22.15 Final design for class `IndexEntry`.

EXERCISES

1. Complete the implementation of the `InfoStore` program.

23 Intermediate class

The class construct has many ramifications and extensions, a few of which are introduced in this chapter.

Section 23.1 looks at the problem of data that need to be shared by all instances of a class. Shared data are quite common. For example, the air traffic control program in Chapter 20 had a minimum height for the aircraft defined by a constant; but it might be reasonable to have the minimum height defined by a variable (at certain times of the day, planes might be required to make their approaches to the auto lander somewhat higher say 1000 feet instead of 600 feet). The minimum height would then have to be a variable. Obviously, all the aircraft are subject to the same height restriction and so need to have access to the same variable. The minimum height variable could be made a global; but that doesn't reflect its use. If really is something that belongs to the aircraft and so should somehow belong to class `Aircraft`. C++ classes have "static" members; these let programmers define such shared data.

*"static" class
members for shared
data*

Section 23.2 introduces "friends". One of the motivations for classes was the need to build privacy walls around data and specialist housekeeping functions. Such walls prevent misuse of data such as can occur with simple structs that are universally accessible. Private data and functions can only be used within the member functions of the class. But sometimes you want to slightly relax the protection. You want private data and functions to be used within member functions, and in addition in a few other functions that are explicitly named. These additional functions may be global functions, or they may be the member functions of some second class. Such functions are nominated as "friends" in a class declaration. (The author of a class nominates the friends if any. You can't come along later and try to make some new function a "friend" of an existing class because, obviously, this would totally defeat the security mechanisms.) There aren't many places where you need friend functions. They sometimes appear when you have a cluster of separate classes whose instances need to work together closely. Then you may get situations where there a class has some data members or functions that you would like to make accessible to instances of other members of the class cluster without also making them accessible to general clients.

*Friends – sneaking
through the walls of
privacy*

-
- Iterators** Section 23.3 introduces iterators. Iterator classes are associated with collection classes like those presented in Chapter 21. An Iterator is very likely to be a "friend" of the collection class with which it is associated. Iterators help you organize code where you want to go through a collection looking at each stored item in turn.
- Operator functions** My own view is that for the most part "operator functions", the topic of Section 23.4, are an overrated cosmetic change to the ordinary function call syntax. Remember how class `Number` in Chapter 19 had functions like `Multiply()` (so the code had things like `a.Multiply(b)` with `a` and `b` instances of class `Number`)? With operator functions, you can make that `a * b`. Redefining the meaning of operator `*` allows you to pretty up such code.
- Such cosmetic uses aren't that important. But there are a few cases where it is useful to redefine operators. For instance, you often want to extend the interface to the `iostream` library so that you can write code like `Number x; ... cout << "x = " << x << endl`. This can be done by defining a new global function involving the `<<` operator. Another special case is the assignment operator, operator `=`; redefinition of operator `=` is explained in the next section on resource manager classes. Other operators that you may need to change are the pointer dereference operator, `->` and the `new` operator. However, the need to redefine the meanings of these operators only occurs in more advanced work, so you won't see examples in this text.
- Resource manager classes** Instances of simple classes, like class `Number`, class `Queue`, class `Aircraft` are all represented by a single block of data. But there are classes where the instances own other data structures (or, more generally, other resources such as open files, network connections and so forth). Class `DynamicArray` is an example; it owns that separately allocated array of `void*` pointers. Classes `List` and `BinaryTree` also own resources; after all, they really should be responsible for those listcells and treenodes that they create in the heap.
- Destructor functions for resource manager classes** Resource managers have special responsibilities. They should make certain that any resources that they claim get released when no longer required. This requirement necessitates a new kind of function – a "destructor". A destructor is a kind of counterpart for the constructor. A constructor function initializes an object (possibly claiming some resources, though usually additional resources are claimed later in the object's life). A destructor allows an object to tidy up and get rid of resources before it is itself discarded. The C++ compiler arranges for calls to be made to the appropriate destructor function whenever an object gets destroyed. (Dynamic objects are destroyed when you apply operator `delete`; automatic objects are destroyed on exit from function; and static objects are destroyed during "at_exit" processing that takes place after return from `main()`.)
- Operator = and resource manager classes** There is another problem with resource manager classes – assignment. The normal meaning of assignment for a struct or class instance is "copy the bytes". Now the bytes in a resource manager will include pointers to managed data structures. If you just copy the bytes, you will get two instances of the resource manager class that both have pointers to the same managed data structure. Assignment causes sharing. This is very rarely what you would want.

If assignment is meaningful for a resource manager, its interpretation is usually "give me a copy just like this existing X"; and that means making copies of any managed resources. The C++ compiler can not identify the managed resources. So if you want assignment to involve copying resources, you have to write a function does this. This becomes the "assignment function" or "operator=()" function. You also have to write a special "copy constructor".

Actually, you usually want to say "instances of this resource manager class cannot be assigned". Despite examples in text books, there are very few situations in real programs where you want to say something like "give me a binary tree like this existing binary tree". There are mechanisms that allow you to impose constraints that prohibit assignment.

*Preventing
assignment*

The final section of this chapter introduces the idea of inheritance. Basically, inheritance allows you to define a new class that in some way extends an existing defined class. There are several different uses for inheritance and the implications of inheritance are the main topic of Part V of this text.

Inheritance

Although your program may involve many different kinds of object, there are often similarities among classes. Sometimes, it is possible to exploit such similarities to simplify the overall design of a program. An example like this is used to motivate the use of class hierarchies where specialized classes inherit behaviours from more general abstract classes.

The next subsection shows how class hierarchies can be defined in C++ and explains the meanings of terms like "virtual function". Other subsections provide a brief guide to how programs using class hierarchies actually work and cover some uses of multiple inheritance.

23.1 SHARED CLASS PROPERTIES

A class declaration describes the form of objects of that class, specifying the various data members that are present in each object. Every instance of the class is separate, every instance holds its own unique data.

Sometimes, there are data that you want to have shared by all instance of the class. The introduction section of this chapter gave the example of the aircraft that needed to "share" a minimum height variable. For second example, consider the situation of writing a C++ program that used Unix's Xlib library to display windows on an Xterminal. You would probably implement a class Window. A Window would have data members for records that describe the font to be used for displaying text, an integer number that identifies the "window" actually manipulated by the interpretive code in the Xterminal itself, and other data like background colour and foreground colour. Every Window object would have its own unique data in its data members. But all the windows will be displayed on the same screen of the same display. In Xlib the screen and the display are described by data structures; many of the basic graphics calls require these data structures to be included among the arguments.

You could make the "Display" and the "Screen" global data structures. Then all the Window objects could use these shared globals.

But the "Display" and the "Screen" should only be used by Windows. If you make them globals, they can be seen from and maybe get misused in other parts of the program.

The C++ solution is to specify that such quasi globals be changed to "class members" subject to the normal security mechanisms provided by C++ classes. If the variable that represents the minimum height for aircraft, or those that represent the Display and Screen used by Windows, are made private to the appropriate classes, then they can only be accessed from the member functions of those classes.

Of course, you must distinguish these shared variables from those where each class instance has its own copy. This is done using the keyword `static`. (This is an unfortunate choice of name because it is a quite different meaning from previous uses of the keyword `static`.) The class declarations defining these shared variables would be something like the following:

*Class declarations
with static data
members*

```
class Aircraft {
public:
    Aircraft();
    ...
private:
    static int      sMinHeight;
    int            fTime;
    PlaneData      fData;
};

class Window {
public:
    ...
private:
    static Screen    sScreen;
    static Display   sDisplay;
    GC              fGC;
    XRectangle       fRect;
    ...
};
```

(As usual, it is helpful to have some naming convention. Here, static data members of classes will be given names starting with 's'.)

*Defining the static
variables*

The class declarations specify that these variables will exist somewhere, but they don't define the variables. The definitions have to appear elsewhere. So, in the case of class `Aircraft`, the header file would contain the class declaration specifying the existence of the class data member `sMinHeight`, the definition would appear in the `Aircraft.cp` implementation file:

```
#include "Aircraft.h"
```

```

int Aircraft::sMinHeight = 1000; // initialize to safe 1000' value

...
int Aircraft::TooLow()
{
    return (fData.z < sMinHeight);
}

```

The definition must use the full name of the variable; this is the member name qualified by the class name, so `sMinHeight` has to be defined as `Aircraft::sMinHeight`. The `static` qualifier should not be repeated in the definition. The definition can include an initial value for the variable.

The example `TooLow()` function illustrates use of the `static` data member from inside a member function.

Quite often, such `static` variables need to be set or read by code that is not part of any of the member functions of the class. For example, the code of the `AirController` class would need to change the minimum safe height. Since the variable `sMinHeight` is private, a public access function must be provided:

```

void Aircraft::SetMinHeight(int newmin) { sMinHeight = newmin;
}

```

Most of the time the `AirController` worked with individual aircraft asking them to perform operations like print their details: `fAircraft[i]->PrintOn(cout)`. But when the `AirController` has to change the minimum height setting, it isn't working with a specific `Aircraft`. It is working with the `Aircraft` class as a whole. Although it is legal to have a statement like `fAircraft[i]->SetMinHeight(600)`, this isn't appropriate because the action really doesn't involve `fAircraft[i]` at all.

Static member functions

A member function like `SetMinHeight()` that only operates on `static` (class) data members should be declared as a `static` function:

```

class Aircraft {
public:
    Aircraft();
    ...
    static void    SetMinHeight(int newmin);
private:
    static int     sMinHeight;
    int            fTime;
    PlaneData      fData;
};

```

Class declarations with static data and function members

This allows the function to be invoked by external code without involving a specific instance of class `Aircraft`, instead the call makes clear that it is "asking the class as a whole" to do something.

Calling a static member function

```

void AirController::ChangeHeight()
{
    int h;
    cout << "What is the new minimum? ";
    cin >> h;
    if((h < 300) || (h > 1500)) {
        cout << "Don't be silly" << endl; return;
    }
    Aircraft::SetMinHeight(h);
}

```

Use of statics

You will find that most of the variables that you might initially think of as being "globals" will be better defined as `static` members of one or other of the classes in your program.

One fairly common use is getting a unique identifier for each instance of a class:

```

class Thing {
public:
    Thing();
    ...
private:
    static int    sIdCounter;
    int    fId;
    ...
};

int Thing::sIdCounter = 0;

Thing::Thing() { fId = ++sIdCounter; ... }

```

Each instance of class `Thing` has its own identifier, `fId`. The `static` (class) variable `sIdCounter` gets incremented every time a new `Thing` is created and so its value can serve as the latest `Thing`'s unique identifier.

23.2 FRIENDS

As noted in the introduction to this chapter, the main use of "friend" functions will be to help build groups (clusters) of classes that need to work closely together.

In Chapter 21, we had class `BinaryTree` that used a helper class, `TreeNode`. `BinaryTree` created `TreeNodes` and got them to do things like replace their keys. Other parts of the program weren't supposed to use `TreeNodes`. The example in Chapter 21 hid the `TreeNode` class inside the implementation file of `BinaryTree`. The header file defining class `BinaryTree` merely had the declaration `class TreeNode;` which simply allowed it to refer to `TreeNode*` pointers. This arrangement prevents other parts of a program from using `TreeNodes`. However, there are times when you can't arrange the implementation like that; code for the main class (equivalent to

BinaryTree) might have to be spread over more than one file. Then, you have to properly declare the auxiliary class (equivalent of `TreeNode`) in the header file. Such a declaration exposes the auxiliary class, opening up the chance that instances of the auxiliary class will get used inappropriately by other parts of the program.

This problem can be resolved using a friend relation as follows:

```
class Auxiliary {
    friend class MainClass;
private:
    Auxiliary();
    int    SetProp1(int newval);
    void   PrintOn(ostream&) const;
    ...
    int    fProp1;
    ...
};

class MainClass {
public:
    ...
};
```

A very private class

that has a friend

All the member functions and data members of class `Auxiliary` are declared private, even the constructor. The C++ compiler will systematically enforce the private restriction. If it finds a variable declaration anywhere in the main code, e.g. `Auxiliary a1;`, it will note that this involves an implicit call to the constructor `Auxiliary::Auxiliary()` and, since the constructor is private, the compiler will report an access error. Which means that you can't have any instances of class `Auxiliary`!

However, the `friend` clause in the class declaration partially removes the privacy wall. Since class `MainClass` is specified to be a friend of `Auxiliary`, member functions of `MainClass` can invoke any member functions (or data members) of an `Auxiliary` object. Member functions of class `MainClass` can create and use instances of class `Auxiliary`.

There are other uses of friend relations but things like this example are the main ones. The friend relation is being used to selectively "export" functionality of a class to chosen recipients.

23.3 ITERATORS

With collection classes, like those illustrated in Chapter 21, it is often useful to be able to step through the collection processing each data member in turn. The member functions for `List` and `DynamicArray` did allow for such iterative access, but only in a relatively clumsy way:

```

DynamicArray    dl;
...
...
for(int i = 1; i < dl.Length(); i++) {
    Thing* t = (Thing*) dl.Nth(i);
    t->DoSomething();
    ...
}

```

That code works OK for `DynamicArray` where `Nth()` is basically an array indexing operation, but it is inefficient for `List` where the `Nth()` operation involves starting at the beginning and counting along the links until the desired element is found.

The `PrintOn()` function for `BinaryTree` involved a "traversal" that in effect iterated through each data item stored in the tree (starting with the highest key and working steadily to the item with the lowest key). However the `BinaryTree` class didn't provide any general mechanism for accessing the stored elements in sequence.

Mechanisms for visiting each data element in turn could have been incorporated in the classes. The omission was deliberate.

Increasingly, program designers are trying to generalize, they are trying to find mechanisms that apply to many different problems. General approaches have been proposed for working through collections.

The basic idea is to have an "Iterator" associated with the collection (each collection has a specialized form of Iterator as illustrated below). An Iterator is in itself a simple class. Its public interface would be something like the following (function names may differ and there may be slight variations in functionality):

```

class Iterator {
public:
    Iterator(...);
    void    First(void);
    void    Next(void);
    int     IsDone(void);
    void    *CurrentItem(void);
private:
    ...
};

```

The idea is that you can create an iterator object associated with a list or tree collection. Later you can tell that iterator object to arrange to be looking at the "first" element in the collection, then you can loop examining the items in the collection, using `Next()` to move on to the next item, and using the `IsDone()` function to check for completion:

```

Collection c1;
...
Iterator    il(c1);
il.Start();
while(!il.IsDone()) {

```



```

Thing* t = (Thing*) il.CurrentItem();
t->DoSomething();
...;
il.Next();
}

```

This same code would work whether the collection were a `DynamicArray`, a `List`, or a `BinaryTree`.

As explained in the final section of this chapter, it is possible to start by giving an abstract definition of an iterator as a "pure abstract class", and then define derived subclasses that represent specialized iterators for different types of collection. Here, we won't bother to define the general abstraction, and will just define and use examples of specialized classes for the different collections.

An "abstract base class" for Iterators?

The iterators illustrated here are "insecure". If a collection gets changed while an iterator is working, things can go wrong. (There is an analogy between an iterator walking along a list and a person using stepping stones to cross a river. The iterator moves from listcell to listcell in response to `Next()` requests; it is like a person stepping onto the next stone and stopping after each step. Removal of the listcell where the iterator is standing has an effect similar to magically removing a stepping stone from under the feet of the river crosser.) There are ways of making iterators secure, but they are too complex for this introductory treatment.

Insecure iterators

23.3.1 ListIterator

An iterator for class `List` is quite simple to implement. After all, it only requires a pointer to a listcell. This pointer starts pointing to the first listcell, and in response to "Next" commands should move from listcell to listcell. The code implementing the functions for `ListIterator` is so simple that all its member functions can be defined "inline".

Consequently, adding an iterator for class `List` requires only modification of the header file:

```

#ifndef __MYLIST__
#define __MYLIST__

class ListIterator;

class List {
public:
    List();

    int          Length(void) const;
    ...
    friend class ListIterator;
private:

```

Nominate friends

```

    struct ListCell { void *fData; ListCell *fNext; };
    ListCell *Head(void) const;

    int          fNum;
    ListCell     *fHead;
    ListCell     *fTail;
};

    class ListIterator {
    public:
        ListIterator(List *l);
        void First(void);
        void Next(void);
        int  IsDone(void);
        void *CurrentItem(void);
    private:
        List::ListCell *fPos;
        List            *fList;
    };

    inline int List::Length(void) const { return fNum; }
    inline List::ListCell *List::Head() const { return fHead; }

    inline ListIterator::ListIterator(List *l)
    { fList = l; fPos = fList->Head(); }
    inline void ListIterator::First(void) { fPos = fList->Head(); }
    inline void ListIterator::Next(void)
    { if(fPos != NULL) fPos = fPos->fNext; }
    inline int ListIterator::IsDone(void) { return (fPos == NULL); }
    inline void *ListIterator::CurrentItem(void)
    { if(fPos == NULL) return NULL; else return fPos->fData; }
    #endif

```

Friend nomination

There are several points to note in this header file. Class `List` nominates class `ListIterator` as a friend; this means that in the code of class `ListIterator`, there can be statements involving access to private data and functions of class `List`.

**Access function
`List::Head()`**

Here, an extra function is defined – `List::Head()`. This function is private and therefore only useable in class `List` and its friends (this prevents clients from getting at the head pointer to the chain of listcells). Although, as a friend, a `ListIterator` can directly access the `fHead` data member, it is still preferable that it use a function style interface. You don't really want friends becoming too intimate for that makes it difficult to locate problems if something goes wrong.

**Declaration of
`ListIterator` class**

The class declaration for `ListIterator` is straightforward except for the type of its `fPos` pointer. This is a pointer to a `ListCell`. But the struct `ListCell` is defined within class `List`. If, as here, you want to refer to this data type in code outside of that of class `List`, you must give its full type name. This is a `ListCell` as defined by class `List`. Hence, the correct type name is `List::ListCell`.

The member functions for class `ListIterator` are all simple. The constructor keeps a pointer to the `List` that it is to work with, and initializes the `fPos` pointer to the first listcell in the list. Member function `First()` resets the pointer (useful if you want the iterator to run through the list more than once); `Next()` advances the pointer; `CurrentItem()` returns the data pointer from the current listcell; and `IsDone()` checks whether the `fPos` pointer has advanced off the end of the list and become `NULL`. (The code for `Next()` checks to avoid falling over at the end of a list by being told to take the "next" of a `NULL` pointer. This could only occur if the client program was in error. You might choose to "throw an exception", see Chapter 26, rather than make it a "soft error".)

The test program used to exercise class `List` and class `DynamicArray` can be extended to check the implementation of class `ListIterator`. It needs a new branch in its `switch()` statement, one that allows the tester to request that a `ListIterator` "walk" along the `List`:

```
case 'w':
{
    ListIterator li(&c1);
    li.First();
    cout << "Current collection " << endl;
    while(!li.IsDone()) {
        Book p = (Book) li.CurrentItem();
        cout << p << endl;
        li.Next();
    }
}
break;
```

The statement:

```
ListIterator li(&c1);
```

creates a `ListIterator`, called `li`, giving it the address of the `List`, `c1`, that it is to work with (the `ListIterator` constructor specifies a pointer to `List`, hence the need for an `&` address of operator).

The statement, `li.First()`, is redundant because the constructor has already performed an equivalent initialization. It is there simply because that is the normal pattern for walking through a collection:

```
li.First();
while(!li.IsDone()) {
    ... li.CurrentItem();
    ...
    li.Next();
}
```

Note the need for the typecast:

```
Book p = (Book) li.CurrentItem();
```

In the example program, `Book` is a pointer type (actually just a `char*`). The `CurrentItem()` function returns a `void*`. The programmer knows that the only things that will be in the `cl` list are `Book` pointers; so the type cast is safe. It is also necessary because of course you can't really do anything with a `void*` and here the code needs to process the books in the collection.

Backwards and forwards iterators in two way lists

Class `List` is singly linked, it only has "next" pointers in its listcells. This means that it is only practical to "walk forwards" along the list from the head to the tail. If the list class uses listcells with both "next" and "previous" pointers, it is practical to walk the list in either direction. Iterators for doubly linked lists usually take an extra parameter in their constructor; this is a "flag" that indicates whether the iterator is a "forwards iterator" (start at the head and follow the next links) or a "backwards iterator" (start at the tail and follow the previous links).

23.3.2 Treeliterator

Like doubly linked lists that can have forwards or backwards iterators, binary trees can have different kinds of iterator. An "in order" iterator process the left subtree, handles the data at a `treenode`, then processes the right subtree; a "pre order" iterator processes the data at a tree node before examining the left and right subtrees. However, if the binary tree is a search tree, only "in order" traversal is useful. An in order style of traversal means that the iterator will return the stored items in increasing order by key.

An iterator that can "walk" a binary tree is a little more elaborate than that needed for a list. It is easy to descend the links from the root to the leaves of a tree, but there aren't any "back pointers" that you could use to find your way back from a leaf to the root. Consequently, a `TreeIterator` can't manage simply with a pointer to the current `TreeNode`, it must also maintain some record of information describing how it reached that `TreeNode`.

Stack of pointers maintain state of traversal

As illustrated in Figure 23.1, the iterator uses a kind of "stack" of pointers to `TreeNodes`. In response to a `First()` request, it chases down the left vine from the root to the left most leaf; so, in the example shown in Figure 23.1 it stacks up pointers to the `TreeNodes` associated with keys 19, 12, 6.

A `CurrentItem()` request should return the data item associated with the entry at the top of this stack.

A `Next()` request has to replace the topmost element by its successor (which might actually already be present in the stack). As illustrated in Figure 23.1, the `Next()` request applied when the iterator has entries for 19, 12, and 6, should remove the 6 and add entries for 9 and 7.

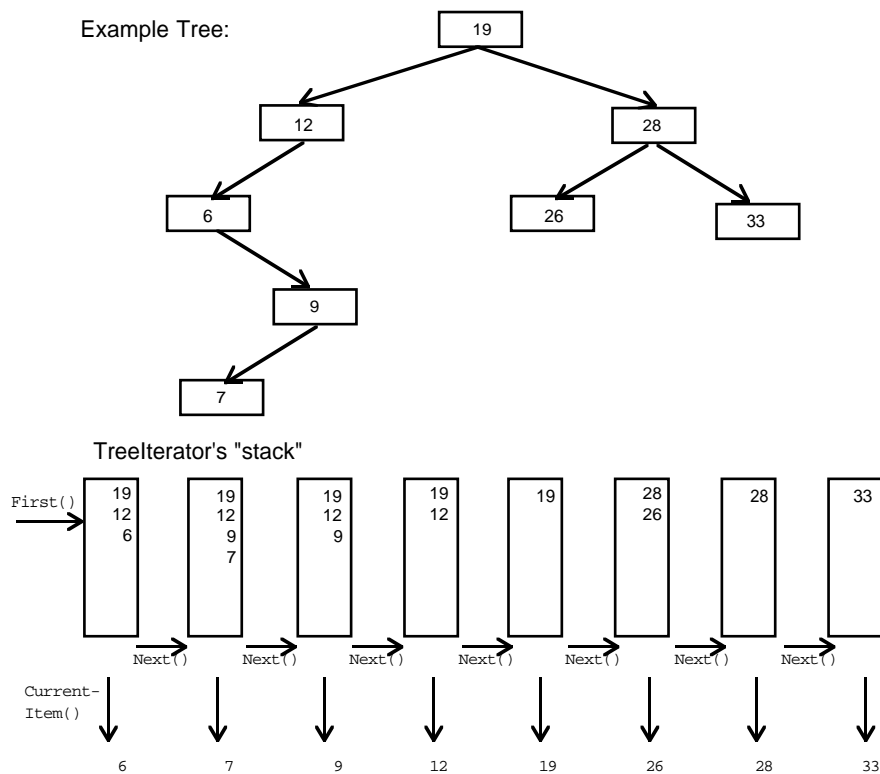


Figure 23.1 Tree and tree iterator.

A subsequent `Next()` request removes the 7, leaving 19, 12, and 9 on the stack. Further `Next()` requests remove entries until the 19 is removed, it has to be replaced with its successor so then the stack is filled up again with entries for 28 and 26.

The programmer implementing class `TreeIterator` has to choose how to represent this stack. If you wanted to be really robust, you would use a `DynamicArray` of `TreeNode` pointers, this could grow to whatever size was needed. For most practical purposes a fixed size array of pointers will suffice, for instance an array with one hundred elements. The size you need is determined by the maximum depth of the tree and thus depends indirectly on the number of elements stored in the tree. If the tree were balanced, a depth of one hundred would mean that the tree had quite a large number of nodes (something like 2^{99}). Most trees are poorly balanced. For example if you inserted 100 data items into a tree in decreasing order of their keys, the left branch would be one hundred deep. Although a fixed array will do, the code needs to check for the array becoming full.

Representing the stack

Class `BinaryTree` has to nominate class `TreeIterator` as a "friend", and again for style its best to provide a private access function rather than have this friend rummage around in the data:

```
class BinaryTree
{
public:
    BinaryTree();
    ...
    friend class TreeIterator;
private:
    TreeNode      *Root(void);
    ...
};

inline TreeNode *BinaryTree::Root(void) { return fRoot; }
```

Class `TreeIterator` has the standard public interface for an iterator; its private data consist of a pointer to the `BinaryTree` it works with, an integer defining the depth of the "stack", and the array of pointers:

```
class TreeIterator {
public:
    TreeIterator(BinaryTree *tree);
    void First(void);
    void Next(void);
    int IsDone(void);
    void *CurrentItem(void);
private:
    int fDepth;
    TreeNode *fStack[kITMAXDEPTH];
    BinaryTree *fTree;
};
```

The constructor simply initializes the pointer to the tree and the depth counter. This initial value corresponds to the terminated state, as tested by the `IsDone()` function. For this iterator, a call to `First()` must be made before use.

```
TreeIterator::TreeIterator(BinaryTree *tree)
{
    fTree = tree;
    fDepth = -1;
}

int TreeIterator::IsDone(void)
{
    return (fDepth < 0);
}
```

Function `First()` starts at the root and chases left links for as far as it is possible to go; each `TreeNode` visited during this process gets stacked up. This process gets things set up so that the data item with the smallest key will be the one that gets fetched first.

```
void TreeIterator::First(void)
{
    fDepth = -1;
    TreeNode *ptr = fTree->Root();
    while(ptr != NULL) {
        fDepth++;
        fStack[fDepth] = ptr;
        ptr = ptr->LeftLink();
    }
}
```

Data items are obtained from the iterator using `CurrentItem()`. This function just returns the data pointer from the `TreeNode` at the top of the stack:

```
void *TreeIterator::CurrentItem(void)
{
    if(fDepth < 0) return NULL;
    else
        return fStack[fDepth]->Data();
}
```

The `Next()` function has to "pop" the top element (i.e. remove it from the stack) and replace it by its successor. Finding the successor involves going down the right link, and then chasing left links as far as possible. Again, each `TreeNode` visited during this process gets "pushed" onto the stack. (If there is no right link, the effect of `Next()` is merely to pop an element from the stack.)

```
void TreeIterator::Next(void)
{
    if(fDepth < 0) return;

    TreeNode *ptr = fStack[fDepth];
    fDepth--;
    ptr = ptr->RightLink();
    while(ptr != NULL) {
        fDepth++;
        fStack[fDepth] = ptr;
        ptr = ptr->LeftLink();
    }
}
```

Use of the iterator should be tested. An additional command can be added to the test program shown previously:

```

case 'w':
{
    TreeIterator ti(&gTree);
    ti.First();
    cout << "Current tree " << endl;
    while(!ti.IsDone()) {
        DataItem *d = (DataItem*) ti.CurrentItem();
        d->PrintOn(cout);
        ti.Next();
    }
}
break;

```

23.4 OPERATOR FUNCTIONS

Those `Add()`, `Subtract()`, and `Multiply()` functions in class `Number` (Chapter 19) seem a little unaesthetic. It would be nicer if you could write code like the following:

```

Number a("97417627567654326573654365865234542363874266");
Number b("65765463658764538654137245665");
Number c;
c = a + b;

```

The operations `'+'`, `'-'`, `'/'` and `'*'` have their familiar meanings and `c = a + b` does read better than `c = a.Add(b)`. Of course, if you are going to define `'+'`, maybe you should define `++`, `+=`, `--`, `-=`, etc. If you do start defining operator functions you may have quite a lot of functions to write.

Operator functions are overrated. There aren't that many situations where the operators have intuitive meanings. For example you might have some "string" class that packages C-style character strings (arrays each with a `'\0'` terminating character as its last element) and provides operations like `Concatenate` (append):

```

String a("Hello");
String b(" World");

c = a.Concatenate(b);           // or maybe?  c = a + b;

```

You could define a `'+'` operator to work for your string class and have it do the concatenate operation. It might be obvious to you that `+` means "append strings", but other people won't necessarily think that way and they will find your `c = a + b` more difficult to understand than `c = a.Concatenate(b)`.

When you get to use the graphics classes defined in association with your IDE's framework class library, you will find that they often have some operator functions defined. Thus class `Point` may have an `operator+` function (this will do something

like vector addition). Or, you might have class `Rectangle` where there is an `"operator+(const Point&)"` function; this curious thing will do something like move the rectangle's topleft corner by the `x, y` amount specified by the `Point` argument (most people find it easier if the class has a `Rectangle::MoveTopLeftCorner()` member function).

Generally, you should not define operator functions for your classes. You can make exceptions for some. Class `Number` is an obvious candidate. You might be able to pretty up class `Bitmap` by giving it "And" and "Or" functions that are defined in terms of operators.

Apart from a few special classes where you may wish to define several operator functions, there are a couple of operators whose meanings you have to redefine in many classes.

23.4.1 Defining operator functions

As far as a compiler is concerned, the meaning of an operator like '+' is defined by information held in an internal table. This table will specify the code that has to be generated for that operator. The table will have entries like:

operator context		translation
long	+	long
		load integer register with <i>first data item</i> add <i>second data item</i> to contents of register
double	+	double
		load floating point register with <i>first data item</i> add <i>second data item</i> to contents of register

The translation may specify a sequence of instructions like those shown. But some machines don't have hardware for all arithmetic operations. There are for example RISC computers that don't have "floating point add" and "floating point multiply"; some don't even have "integer divide". The translations for these operators will specify the use of a function:

operator context		translation
long	/	long
		push dividend and divisor onto stack call ".div" function

In most languages, the compiler's translation tables are fixed. C++ allows you to add extra entries. So, if you have some "add" code for a class `Point` that you've defined and you want this called for `Point + Point`, you can specify this to the compiler. It takes details from your specification and appends these to its translation tables:

operator context		translation
point	+	point
		push the two points onto the stack call the function defined by the programmer

The specifications that must appear in your classes are somewhat unpronounceable. An addition operator would be defined as the function:

```
operator+( )
```

(say that as "operator plus function"). For example, you could have:

```
class Point {
public:
    Point();
    ...
    Point operator+(const Point& other) const;
    ...
private:
    int    fh, fv;
};
```

with the definition:

```
Point Point::operator+(const Point& other) const
{
    Point vecSum;
    vecSum.fh = this->fh + other.fh;
    vecSum.fv = this->fv + other.fv;
    return vecSum;
}
```

This example assumes that the + operation shouldn't change either of the `Point`s that it works on but should create a temporary `Point` result (in the return part of a function stackframe) that can be used in an assignment; this makes it like + for integers and doubles.

It is up to you to define the meaning of operator functions. Multiplying points by points isn't very meaningful, but multiplying points by integers is equivalent to scaling. So you *could* have the following where there is a multiply function that changes the `Point` object that executes it:

```
class Point {
public:
    Point();
    ...
    Point operator+(const Point& other) const;
    Point& operator*(int scalefactor);
    ...
private:
    int    fh, fv;
};
```

with a definition:

```
Point& Point::operator*(int scalefactor)
{
    // returning a reference allows expressions that have
    // scaling operations embedded inside them.
    fh *= scalefactor;
    fv *= scalefactor;
    return *this;
}
```

with these definitions you can puzzle anyone who has to read and maintain your code by having constructs like:

```
Point a(6,4);

...;
a*3;

Point b(7, 2);
Point c;
...
c = b + a*4;
```

Sensible maintenance programmers will eventually get round to changing your code to:

```
class Point {
public:
    Point();
    ...
    Point operator+(const Point& other) const;
    void ScaleBy(int scalefactor);
    ...
};

void Point::ScaleBy(int scalefactor)
{
    fh *= scalefactor;
    fv *= scalefactor;
}
```

resulting in more intelligible programs:

```
Point a(6,4);

...;
a.ScaleBy(3);
```

```

Point b(7, 2);
Point c;
...
a.ScaleBy(4);
c = b + a;

```

Avoid the use of operator functions except where their meanings are universally agreed. If their meanings are obvious, operator function can result in cosmetic improvements to the code; for example, you can pretty up class `Number` as follows:

```

class Number {
public:
    // Member functions declared as before
    ...
    Number operator+(const Number& other) const;
    ...
    Number operator/(const Number& other) const;
private:
    // as before
    ...
};

inline Number Number::operator+(const Number& other) const
{
    return this->Add(other);
}

```

Usually, the meanings of operator functions are not obvious

23.4.2 Operator functions and the iostream library

You will frequently want to extend the meanings of the `<<` and `>>` operators. A C++ compiler's built in definition for these operators is quite limited:

operator context	translation
long << long	load integer register with <i>first data item</i> shift left by the specified number of places
long >> long	load integer register with <i>first data item</i> shift right by the specified number of places

But if you `#include` the `iostream` header files, you add all the "takes from" and "gives to" operators:

operator context	translation
ostream << long	push the ostream id and the long onto the stack call the function "ostream::operator<<(long)"

```
istream >> long    push the istream id and the address of the long
                   onto the stack
                   call the function "istream::operator>>(long&)"
```

These entries are added to the table as the compiler reads the istream header file with its declarations like:

```
class ostream {
public:
    ...
    ostream& operator<<(long);
    ostream& operator<<(char*);
    ...
};
```

Such functions declared in the istream.h header file are member functions of class istream or class ostream. An ostream object "knows" how to print out a long integer, a character, a double, a character string and so forth.

How could you make an ostream object know how to print a Point or some other programmer defined class?

Typically, you will already have defined a PrintOn() member function in your Point class.

```
class Point {
public:
    ...
    void PrintOn(ostream& out);
private:
    int    fh, fv;
};

void Point::PrintOn(ostream& out)
{
    out << "(" << fh << ", " << fv << ") ";
}
```

and all you really want to do is make it possible to write something like:

```
Point p1, p2;
...
cout << "Start point " << p1 << ", end point " << p2 << endl;
```

rather than:

```
cout << "Start point ";
p1.PrintOn(cout);
cout << ", end point ";
```

```
p2.PrintOn(cout);
cout << endl;
```

You want some way of telling the compiler that if it sees the << operator involving an ostream and a Point then it is to use code similar to that of the Point::PrintOn() function (or maybe just use a call to the existing PrintOn() function).

You *could* change the classes defined in the ostream library. You could add extra member functions:

```
class ostream {
// everything as now plus
    ostream& operator<<(const Point& p);
    ...
};
```

and provide your definition of ostream& ostream::operator<<(const Point&).

It should be obvious that this is not desirable. The ostream library has been carefully developed and debugged. You wouldn't want hundreds of copies each with minor extensions hacked in.

Fortunately, such changes aren't necessary. There is another way of achieving the desired effect.

*A global
operator<<(ostream&
, Point&) function*

You can define global operator functions. These functions aren't members of classes. They are simply devices for telling the compiler how it is to translate cases where it finds an operator involving arguments of specified types.

In this case, you need to define a new meaning for the << operator when it must combine an ostream and a Point. So you define:

```
?? operator<<(ostream& os, const Point& p)
{
    p.PrintOn(os);
    return ??
}
```

(the appropriate return type will be explained shortly). The compiler invents a name for the function (it will be something complex like __leftshift_Tostreamref_cTPointref) and adds the new meaning for << to its table:

operator context	translation
ostream << point	push the ostream id and the point's address onto the stack
	call the function __leftshift_Tostreamref_cTPointref

This definition then allows constructs like: Point p; ...; cout << p;.

Of course, the ideal is for the stream output operations to be concatenated as in:

```
cout << "Start point " << p1 << ", end point " << p2 << endl;
```

This requirement defines the return type of the function. It must return a reference to the ostream:

```
ostream& operator<<(ostream& os, const Point& p)
{
    p.PrintOn(os);
    return os;
}
```

Having a reference to the stream returned as a result permits the concatenation. Figure 23.2 illustrates the way that the scheme works.

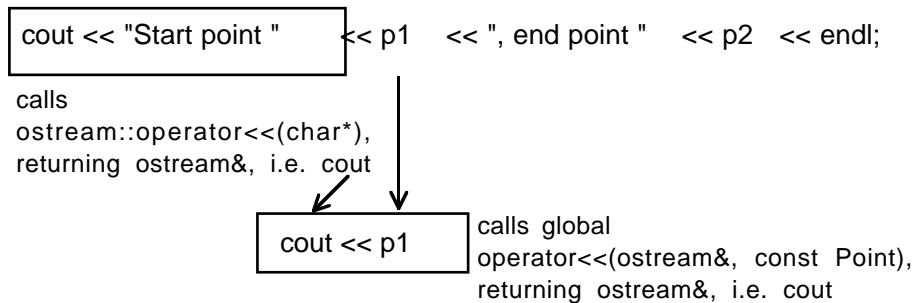


Figure 23.2 Illustration of groupings involved in concatenated use of ostream& operator<<() functions.

You might also want:

```
ostream& operator<<(ostream& os, Point *p_pt)
{
    p_pt->PrintOn(os);
    return os;
}
```

23.5 RESOURCE MANAGER CLASSES AND DESTRUCTORS

This section explains some of the problems associated with "resource manager" classes.

Resource manager classes are those whose instances own other data structures. Usually, these will be other data structures separately allocated in the heap. We've already seen examples like class `DynamicArray` whose instances each own a separately allocated array structure. However, sometimes the separately allocated data structures

may be in operating system's area; examples here are resources like open files, or "ports and sockets" as used for communications between programs running on different computers.

The problems for resource managers are:

- disposal of managed resources that are no longer required;
- unintended sharing of resources.

The first subsection, 23.5.1, provides some examples illustrating these problems. The following two sections present solutions.

23.5.1 Resource management

Instances of classes can acquire resources when they are created, or as a result of subsequent actions. For example, an object might require a variable length character string for a name:

```
class DataItem {
public:
    DataItem(const char* dname);
    ...
private:
    char    *fName;
    ...
};

DataItem::DataItem(const char* dname)
{
    fName = new char[strlen(dname) + 1];
    strcpy(fName, dname);
    ...
}
```

Another object might need to use a file:

```
class SessionLogger {
public:
    SessionLogger();
    ...
    int    OpenLogFile(const char* logname);
    ...
private:
    ...
    ofstream    fLfile;
    ...
};
```



```
int SessionLogger::OpenLogFile(const char* logname)
{
    fLfile.open(logname, ios::out);
    return fLfile.good();
}
```

Instances of the `DataItem` and `SessionLogger` classes will be created and destroyed in various ways:

```
void DemoFunction()
{
    while(AnotherSession()) {
        char    name[100];
        cout << "Session name: "; cin >> name;
        SessionLogger sl;
        if(0 == sl.OpenLogFile(name)) {
            cout << "Can't continue, no file.";
            break;
        }
        for(;;) {
            char    dbuff[100];
            ...
            DataItem    *dptr = new DataItem(dbuff);
            ...
            delete dptr;
        }
    }
}
```

In the example code, a `SessionLogger` object is, in effect, created in the stack and subsequently destroyed for each iteration of the `while` loop. In the enclosed `for` loop, `DataItem` objects are created in the heap, and later explicitly deleted.

Figure 23.3 illustrates the representation of a `DataItem` (and its associated name) in the heap, and the effect of the statement `delete dptr`. As shown, the space occupied by the primary `DataItem` structure itself is released; but the space occupied by its name string remains "in use". Class `DataItem` has a "memory leak".

Figure 23.4 illustrates another problem with class `DataItem`, this problem is sharing due to assignment. The problem would show up in code like the following (assume for this example that class `DataItem` has member functions that change the case of all letters in the associated name string):

```
DataItem d1("This One");
DataItem d2("another one");
...
d2 = d1;
```

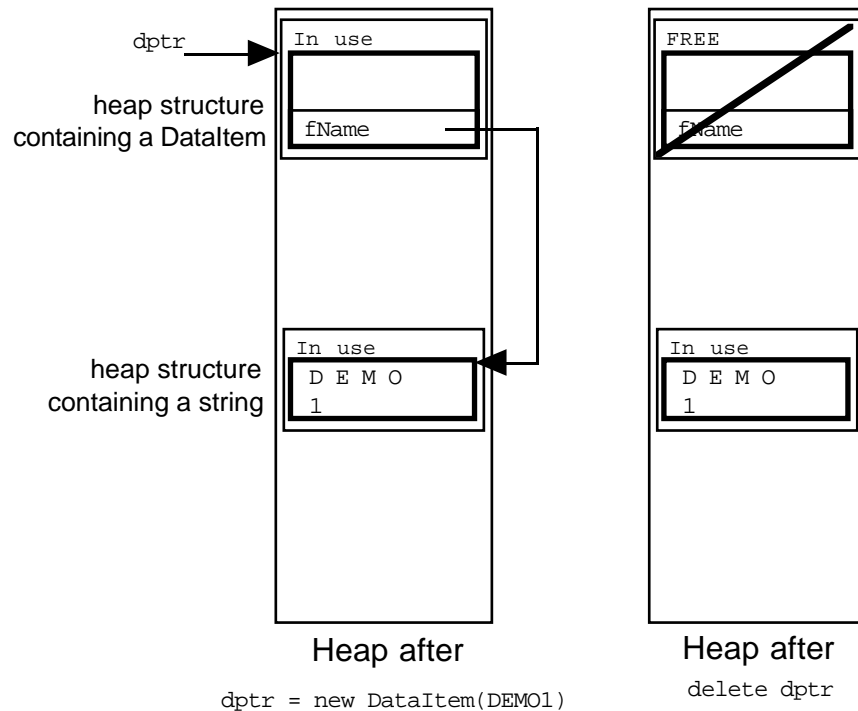


Figure 23.3 Resource manager class with memory leak.

```
...
d1.MakeLowerCase();
d2.MakeUpperCase();
d1.PrintOn(cout);
...
```

The assignment `d2 = d1` will work by default. The contents of record structure `d1` are copied field by field into `d2`, so naturally `d2`'s `fName` pointer is changed to point to the same string as that referenced by `d1.fName`. (There is also another memory leak; the string that used to be owned by `d2` has now been abandoned.)

Since `d2` and `d1` both share the same string, any operations that they perform on that string will interact. Although object `d1` has made its string lower case, `d2` changes it to upper case so that when printed by `d1` it appears as upper case.

Class `SessionLogger` has very similar problems. The resource that a `SessionLogger` object owns is some operating system structure describing a file. Such structures, let's just call them "file descriptors," get released when files are closed. If a `SessionLogger` object is destroyed before it closes its associated file, the file descriptor structures remain. When a program finishes, all files are closed and the associated file descriptor structures are released.

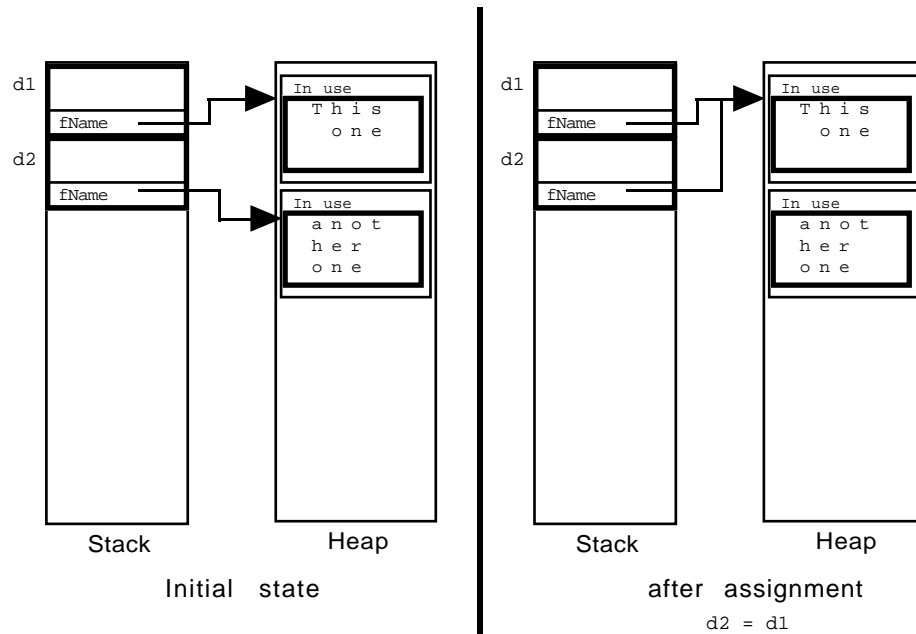


Figure 23.4 Assignment leading to sharing of resources.

However, an operating system normally limits the number of file descriptors that a program can own. If `SessionLogger` objects don't close their files, then eventually the program will run out of file descriptors (its a bit like running out of heap space, but you can make it happen a lot more easily).

Structure sharing will also occur if a program's code has assignment statements involving `SessionLoggers`:

```
SessionLogger s1, s2;
...
s1.OpenLogFile("testing");
...
s2 = s1;
```

Both `SessionLogger` objects use the same file. So if one does something like cause a seek operation (explicitly repositioning the point where the next write operation should occur), this will affect the other `SessionLogger`.

23.5.2 Destructor functions

Some of the problems just explained can be solved by arranging that objects get the chance to "tidy up" just before they themselves get destroyed. You *could* attempt to achieve this by hand coding. You would define a "TidyUp" function in each class:

```
void DataItem::TidyUp() { delete [] fName; }

void SessionLogger::TidyUp() { fLfile.close(); }
```

You would have to include explicit calls to these `TidyUp()` functions at all appropriate points in your code:

```
while(AnotherSession()) {
    ...
    SessionLogger s1;
    ...
    for(;;) {
        ...
        DataItem      *dptr = new DataItem(dbuff);
        ...
        dptr->TidyUp();
        delete dptr;
    }
    s1.TidyUp();
}
```

That is the problem with "hand coding". It is very easy to miss some point where an automatic goes out of scope and so forget to include a tidy up routine. Insertion of these calls is also tiresome, repetitious "mechanical" work.

Tiresome, repetitious "mechanical" work is best done by computer program. The compiler program can take on the job of putting in calls to "TidyUp" functions. Of course, if the compiler is to do the work, things like names of functions have to be standardized.

For each class you can define a "destructor" routine that does this kind of tidying up. In order to standardize for the compiler, the name of the destructor routine is based on the class name. For class `x`, you had constructor functions, e.g. `x()`, that create instances, and you can have a destructor function `~x()` that does a tidy up before an object is destroyed. (The character `~`, "tilde", is the symbol used for NOT operations on bit maps and so forth; a destructor is the NOT, or negation, of a constructor.)

Rather than those "TidyUp" functions, class `DataItem` and class `SessionLogger` would both define destructors:

```
class DataItem {
public:
    DataItem(const char *name);
```

```

    ~DataItem();
    ...
};

DataItem::~DataItem() { delete [] fName; }

class SessionLogger {
public:
    SessionLogger();
    ~SessionLogger() { this->fLFile.close(); }
    ...
};

```

Just as the compiler put in the implicit calls to constructor functions, so it puts in the calls to destructors.

You can have a class with several constructors because there may be different kinds of data that can be used to initialize a class. There can only be one destructor; it takes no arguments. Like constructors, a destructor has no return type.

Destructors can exacerbate problems related to structure sharing. As we now have a destructor for class `DataItem`, an individual `DataItem` object will dutifully delete its name when it gets destroyed. If assignment has led to structure sharing, there will be a second `DataItem` around whose name has suddenly ceased to exist.

You don't have to define destructors for all your classes. Destructors are needed for classes that are themselves resource managers, or classes that are used as "base classes" in some class hierarchy (see section 23.6).

Several of the collection classes in Chapter 21 were resource managers and they should have had destructors.

Class `DynamicArray` would be easy, it owns only a single separately allocated array, so all that its destructor need do is get rid of this:

```

class DynamicArray {
public:
    DynamicArray(int size = 10, int inc = 5);
    ~DynamicArray();

    ...

private:
    ...
    void    **fItems;
};

DynamicArray::~DynamicArray() { delete [] fItems; }

```

Note that the destructor does not delete the data items stored in the array. This is a design decision for all these collection classes. The collection does not own the stored items, it merely looks after them for a while. There could be other pointers to stored

items elsewhere in the program. You can have collection classes that do own the items that are stored or that make copies of the original data and store these copies. In such cases, the destructor for the collection class should run through the collection deleting each individual stored item.

Destructors for class `List` and class `BinaryTree` are a bit more complex because instances of these classes "own" many listcells and treenodes respectively. All these auxiliary structures have to be deleted (though, as already explained, the actual stored data items are not to be deleted). The destructor for these collection class will have to run through the entire linked network getting rid of the individual listcells or treenodes.

A destructor for class `List` is as follows:

```
List::~~List()
{
    ListCell *ptr;
    ListCell *temp;
    ptr = fHead;
    while(ptr != NULL) {
        temp = ptr;
        ptr = ptr->fNext;
        delete temp;
    }
}
```

The destructor for class `BinaryTree` is most easily implemented using a private auxiliary recursive function:

```
BinaryTree::~~BinaryTree()
{
    Destroy(fRoot);
}

void BinaryTree::Destroy(TreeNode* t)
{
    if(t == NULL)
        return;
    Destroy(t->LeftLink());
    Destroy(t->RightLink());
    delete t;
}
```

The recursive `Destroy()` function chases down branches of the tree structure. At each `TreeNode` reached, `Destroy()` arranges to get rid of all the `TreeNodes` in the left subtree, then all the `TreeNodes` in the right subtree, finally disposing of the current `TreeNode`. (This is an example of a "post order" traversal; it processes the current node of the tree after, "post", processing both subtrees.)

23.5.3 The assignment operator and copy constructors

There are two places where structures or class instances are, by default, copied using a byte by byte copy. These are assignments:

```

DataItem d1("x"), d2("y");
...
d2 = d1;
...

```

and in calls to functions where objects are passed by value:

```

void foo(DataItem dd) { ... ; ... ; ... }

void test()
{
    DataItem anItem("Hello world");
    ...
    foo(anItem);
    ...
}

```

This second case is an example of using a "copy constructor". Copy constructors are used to build a new class instance, just like an existing class instance. They do turn up in other places, but the most frequent place is in situations like the call to the function requiring a value argument.

As illustrated in section 23.5.1, the trouble with the default "copy the bytes" implementations for the assignment operator and for a copy constructor is that they usually lead to undesired structure sharing.

If you want to avoid structure sharing, you have to provide the compiler with specifications for alternative ways of handling assignment and copy construction. Thus, for `DataItem`, we would need a copy constructor that made a copy of the character string `fName`:

```

DataItem::DataItem(const DataItem& other)
{
    fName = new char[strlen(other.fName) + 1];
    strcpy(fName, other.fName);
    ...
}

```

*A copy constructor
that duplicates an
"owned resource"*

Though similar, assignments are a little more complex. The basic form of an *Assignment operator* function for the example class `DataItem` would be:

```

?? DataItem::operator=(const DataItem& other)
{
    ...
}

```

```

        delete [] fName;
        fName = new char[strlen(other.fName) + 1];
        strcpy(fName, other.fName);
        ...
    }

```

Plugging a memory leak

The statement:

```
delete [] fName;
```

gets rid of the existing character array owned by the `DataItem`; this plugs the memory leak that would otherwise occur. The next two statements duplicate the content of the other `DataItem`'s `fName` character array.

If you want to allow assignments at all, then for consistency with the rest of C++ you had better allow concatenated assignments:

```

DataItem d1("XXX");
DataItem d2("YYY");
DataItem d3("ZZZ");
...
d3 = d2 = d1;

```

To achieve this, you have to have the `DataItem::operator=()` function to return a reference to the `DataItem` itself:

```

DataItem& DataItem::operator=(const DataItem& other)
{
    ...
    delete [] fName;
    fName = new char[strlen(other.fName) + 1];
    strcpy(fName, other.fName);
    ...
    return *this;
}

```

There is a small problem. Essentially, the code says "get rid of the owned array, duplicate the other's owned array". Suppose somehow you tried to assign the value of a `DataItem` to itself; the array that has then to be duplicated is the one just deleted. Such code will usually work, but only because the deleted array remains as a "ghost" in the heap. Sooner or later the code would crash; the memory manager will have rearranged memory in some way in response to the delete operation.

You might guess that "self assignments" are rare. Certainly, those like:

```

DataItem d1("xyz");
...
d1 = d1;

```


are rare (and good compilers will eliminate statements like `d1 = d1`). However, self assignments do occur when you are working with data referenced by pointers. For example, you might have:

```
DataItem *d_ptr1;
DataItem *d_ptr2;
...
// Copy DataItem referenced by d_ptr1 into the DataItem
// referenced by pointer d_ptr2
*d_ptr2 = *d_ptr1;
```

It is of course possible that `d_ptr1` and `d_ptr2` are pointing to the same `DataItem`.

You have to take precautions to avoid problems with self assignments. The following arrangement (usually) works:

```
DataItem& DataItem::operator=(const DataItem& other)
{
    if(this != &other) {
        delete [] fName;
        fName = new char[strlen(other.fName) + 1];
        strcpy(fName, other.fName);
    }
    return *this;
}
```

It checks the addresses of the two `DataItems`. One address is held in the (implicit) pointer argument `this`, the second address is obtained by applying the `&` address of operator to `other`. If the addresses are equal it is a self assignment so don't do anything.

Of course, sometimes it is just meaningless to allow assignment and copy constructors. You really wouldn't want two `SessionLoggers` working with the same file (and they can't really have two files because their files have to have the same name). In situations like this, what you really want to do is to prevent assignments and other copying. You can achieve this by declaring a private copy constructor and a private `operator=` function;

Preventing copying

```
class SessionLogger {
public:
    SessionLogger();
    ~SessionLogger();
    ...
private:
    // No assignment, no copying!
    void operator=(const SessionLogger& other);
    SessionLogger(const SessionLogger& other);
    ...
};
```

You shouldn't provide an implementation for these functions. Declaring these functions as `private` means that such functions can't occur in client code. Code like `SessionLogger s1, s2; ...; s2 = s1;` will result in an error message like "Cannot access `SessionLogger::_assign()` here". Obviously, such operations won't occur in the member functions of the class itself because the author of the class knows that assignment and copying are illegal. The return type of the `operator=` function does not matter in this context, so it is simplest to declare it as `void`.

Assignment and copy construction should be disabled for collection classes like those from Chapter 24, e.g.:

```
class BinaryTree {
public:
    ...
private:
    void operator=(const BinaryTree& other);
    BinaryTree(const BinaryTree& other);
    ...
};
```

23.6 INHERITANCE

Most of the programs that you will write in future will be "object based". You will analyze a problem, identify "objects" that will be present at run-time in your program, and determine the "classes" to which these objects belong. Then you will design the various independent classes needed, implement them, and write a program that creates instances of these classes and allows them to interact.

Independent classes? That isn't always the case.

In some circumstances, in the analysis phase or in the early stages of the design phase you will identify similarities among the prototype classes that you have proposed for your program. Often, exploitation of such similarities leads to an improved design, and sometimes can lead to significant savings in implementation effort.

23.6.1 Discovering similarities among prototype classes

Example application Suppose that you and some colleagues had to write a "Macintosh/Windows" program for manipulating electrical circuits, the simple kinds of circuit that can be made with those "Physics is Fun" sets that ambitious parents buy to disappoint their kids at Xmas. Those kits have wires, switches, batteries, lamp-bulbs and resistors, and sometimes more. A program to simulate such circuits would need an editing component that allowed a circuit to be laid out graphically, and some other part that did all the "Ohm's Law" and "Kirchoff's Law" calculations to calculate currents and "light up" the simulated bulbs.

You have used "Draw" programs so you know the kind of interface that such a program would have. There would be a "palette of tools" that a user could use to add components. The components would include text (paragraphs describing the circuit), and circuit elements like the batteries and light bulbs. The editor part would allow the user to select a component, move it onto the main work area and then, by doubly clicking the mouse button, open a dialog window that would allow editing of text and setting parameters such as a resistance in ohms. Obviously, the program would have to let the user save a partially designed circuit to a file from where it could be restored later.

What objects might the program contain?

The objects are all pretty obvious (at least they are obvious once you've been playing this game long enough). The following are among the more important:

- A "document" object that would own all the data, keep track of the components added and organize transfers to and from disk. *Objects needed*
 - Various collections, either "lists" or "dynamic arrays" used to store items. Lets call them "lists" (although, for efficiency reasons, a real implementation would probably use dynamic arrays). These lists would be owned by the "document". There might be a list of "text paragraphs" (text describing the circuit), a "list of wires", a "list of resistors" and so forth.
 - A "palette object". This would respond to mouse-button clicks by giving the document another battery, wire, resistor or whatever to add to the appropriate list.
 - A "window" or "view" object used when displaying the circuit.
 - Some "dialog" objects used for input of parameters.
 - Lots of "wire" objects.
 - Several "resistor objects".
 - A few "switch" objects".
 - A few "lamp bulb" objects".
- and for a circuit that actually does something
- At least one battery object.

For each, you would need to characterize the class and work out a list of data owned and functions performed.

During a preliminary design process your group would be right to come up with classes Battery, Document, Palette, Resistor, Switch. Each group member could work on refining one or two classes leading to an initial set of descriptions like the following:

- class TextParagraph *Preliminary design ideas for classes*
 - Owns:
 - a block of text and a rectangle defining position in main view (window).
 - Does:
 - GetText() – uses a standard text editing dialog to get text changed;
 - FollowMouse() – responds to middle mouse button by following mouse to reposition within view;
 - DisplayText() - draws itself in view;

Rect() – returns bounding rectangle;

...

Save() and Restore() - transfers text and position details to/.from file.

- class Battery

Owns:

Position in view, resistance (internal resistance), electromotive force, possibly a text string for some label/name, unique identifier, identifiers of connecting wires...

Does:

GetVoltStuff() – uses a dialog to get voltage, internal resistance etc.

TrackMouse() – respond to middle mouse button by following mouse to reposition within view;

DrawBat() - draws itself in view;

AddWire() – add a connecting wire;

Area() – returns rectangle occupied by battery in display view;

...

Put() and Get() – transfers parameters to/from file.

- class Resistor

Owns:

Position in view, resistance, possibly a text string for some label/name, unique identifier, identifiers of connecting wires...

Does:

GetResistance() – uses a dialog to get resistance, label etc.

Move() – respond to middle mouse button by following mouse to reposition within view;

Display() - draws itself in view;

Place() – returns area when resistor gets drawn;

...

ReadFrom() and WriteTo() – transfers parameters to/from file.

You should be able to sketch out pseudo code for some of the main operations. For example, the document's function to save data to a file might be something like the following:

Prototype code using instances of classes

```
Document::DoSave
    write paragraphList.Length()
    iterator i1(paragraphList)
    for i1.First(), !i1.IsDone() do
        paragraph_ptr = i1.CurrentItem();
        paragraph_ptr->Save()
        i1.Next();

    write BatteriesList.Length()
    iterator i2(BatteriesList)
    for i2.First, !i2.IsDone() do
        battery_ptr = i2.CurrentItem()
```

```
battery_ptr->Put()
```

```
...
```

The function to display all the data of the document would be rather similar:

```
Document::Draw
    iterator i1(paragraphList)
    for i1.First(), !i1.IsDone() do
        paragraph_ptr = i1.CurrentItem();
        paragraph_ptr->DisplayText()
        i1.Next();

    iterator i2(BatteriesList)
    for i2.First(), !i2.IsDone() do
        battery_ptr = i2.CurrentItem()
        battery_ptr->DrawBat()
```

```
...
```

Another function of "Document" would sort out which data element was being picked when the user wanted to move something using the mouse pointer:

```
Document::LetUserMoveSomething(Point mousePoint)
    iterator i1(paragraphList)
    Paragraph *pp = NULL;
    for i1.First(), !i1.IsDone() do
        paragraph_ptr = i1.CurrentItem();
        Rectangle r = paragraph_ptr->Rect()
        if(r.Contains(mousePoint) pp = paragraph_ptr;
        i1.Next();
    if(pp != NULL)
        pp->FollowMouse()
        return

    iterator i2(BatteriesList)
    battery *pb
    for i2.First(), !i2.IsDone() do
        battery_ptr = i2.CurrentItem()
        Rectangle r = battery_ptr->Area()
        if(r.Contains(mousePoint) pb = battery_ptr ;
        i2.Next();

    if(pb != NULL)
        pb->TrackMouse()
        return
```

```
...
```

Design problems?

By now you should have the feeling that there is something amiss. The design with its "batteries", "wires", "text paragraphs" seems sensible. But the code is coming out curiously clumsy and unattractive in its inconsistencies.

Batteries, switches, wires, and text paragraphs may be wildly different kinds of things, but from the perspective of "document" they actually have some similarities. They are all "things" that perform similar tasks. A document can ask a "thing" to:

- Save yourself to disk;
- Display your editing dialog;
- Draw yourself;
- Track the mouse as it moves and reposition yourself;
- ...

Similarities among classes

Some "things" are more similar than others. Batteries, switches, and resistors will all have specific roles to play in the circuit simulation, and there will be many similarities in their roles. Wires are also considered in the circuit simulation, but their role is quite different, they just connect active components. Text paragraphs don't get involved in the circuit simulation part. So all of them are "storable, drawable, editable" things, some are "circuit things", and some are "circuit things that have resistances".

A class hierarchy

You can represent such relationships among classes graphically, as illustrated in Figure 23.5. As shown there, there is a kind of hierarchy.

An pure "abstract" class

Class Thing captures just the concept of some kind of data element that can draw itself, save itself to file and so forth. There are no data elements defined for Thing, it is purely conceptual, purely abstract.

Concrete class TextParagraph

A TextParagraph is a particular kind of Thing. A TextParagraph does own data, it owns its text, its position and so forth. You can also define actual code specifying exactly how a TextParagraph might carry out specific tasks like saving itself to file. Whereas class Thing is purely conceptual, a TextParagraph is something pretty real, pretty "concrete". You can "see" a TextParagraph as an actual data structure in a running program.

Partially abstract class CircuitThing

In contrast, a CircuitThing is somewhat abstract. You can define some properties of a CircuitThing. All circuit elements seem to need unique identifiers, they need coordinate data defining their position, and they need a character string for a name or a label. You can even define some of the code associated with CircuitThings – for instance, you could define functions that access coordinate data.

Concrete class Wire

Wires are special kinds of CircuitThings. It is easy to define them completely. They have a few more data fields (e.g. identifiers of the components that they join, or maybe coordinates for their endpoints). It is also easy to define completely how they perform all the functions like saving their data to file or drawing themselves.

Partially abstract class Component

Components are a different specialization of CircuitThing. Components are CircuitThings that will have to be analyzed by the circuit simulation component of the program. So they will have data attributes like "resistance", and they may have many additional forms of behaviour as required in the simulation.

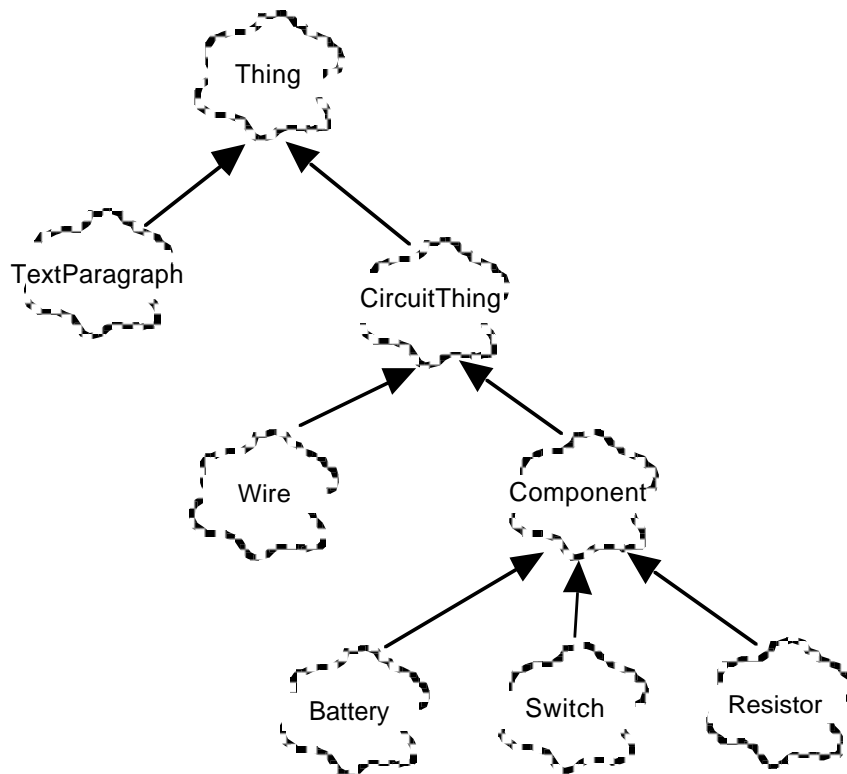


Figure 23.5 Similarities among classes.

Naturally, Battery, Switch, and Resistor define different specializations of this idea of Component. Each will have its unique additional data attributes. Each can define a real implementation for functions like Draw().

*Concrete classes
Battery, Switch, ...*

The benefits of a class hierarchy

OK, such a hierarchy provides a nice conceptual structure when talking about a program but how does it really help?

One thing that you immediately gain is consistency. In the original design sketch, text paragraphs, batteries and so forth all had some way of defining that these data elements could display themselves, save themselves to file and so forth. But each class was slightly different; thus we had `TextParagraph::Save()`, `Battery::Put()` and `Resistor::WriteTo()`. The hierarchy allows us to capture the concept of "storability" by specifying in class `Thing` the ability `WriteTo()`. While each

Consistency

specialization performs `WriteTo()` in a unique way, they can at least be consistent in their names for this common behaviour. But consistency of naming is just a beginning.

**Design
simplifications**

If you exploit such similarities, you can greatly simplify the design of the overall application as can be seen by re-examining some of the tasks that a `Document` must perform.

While you might want separate lists of the various specialized `Components` (as this might be necessary for the circuit simulation code), you could change `Document` so that it stores data using a single `thingList` instead of separate `paragraphList`, `BatteriesList` and so forth. This would allow simplification of functions like `DoSave()`:

**Functions exploiting
similarities**

```
Document::DoSave(...)
    write thingList.Length()
    iterator il(thingList)
    for il.First(), !il.IsDone() do
        thing_ptr = il.CurrentItem();
        thing_ptr->WriteTo()
        il.Next();

Document::Draw
    iterator il(thingList)
    for il.First(), !il.IsDone() do
        thing_ptr = il.CurrentItem();
        thing_ptr->Draw()
        il.Next();

Document::LetUserMoveSomething(Point mousePoint)
    iterator il(thingList)
    Thing *pt = NULL;
    for il.First(), !il.IsDone() do
        thing_ptr = il.CurrentItem();
        Rectangle r = thing_ptr ->Area()
        if(r.Contains(mousePoint) pt = thing_ptr ;
        il.Next();
    if(pt != NULL)
        pt->TrackMouse()
    return
```

The code is no longer obscured by all the different special cases. The revised code is shorter and much more intelligible.

Extendability

Note also how the revised `Document` no longer needs to know about the different kinds of circuit component. This would prove useful later if you decided to have another component (e.g. class `Voltmeter`); you wouldn't need to change the code of `Document` in order to accommodate this extension.

Code sharing

The most significant benefit is the resulting simplification of design, and simultaneous acquisition of extendability. But you may gain more. Sometimes, you can define the code for a particular behaviour at the level of a partially abstract class. Thus, you should be able to define the access function for getting a `CircuitThing`'s

identifier at the level of class `CircuitThing` while class `Component` can define the code for accessing a `Component`'s electrical resistance. Defining these functions at the level of the partially abstract classes saves you from writing very similar functions for each of the concrete classes like `Battery`, `Resistor`, etc.

23.6.2 DEFINING CLASS HIERARCHIES IN C++

C++ allows you to define such hierarchical relations amongst classes. So, there is a way of specifying "class `Thing` represents the abstract concept of a storable, drawable, moveable data element", "class `TextParagraph` is a kind of `Thing` that looks after text and ...".

You start by defining the "base class", in this case that is class `Thing` which is the *Base class* base class for the entire hierarchy:

```
class Thing {
public:
    virtual ~Thing() { }
    /* Disk I/O */
    virtual void ReadFrom(istream& i s) = 0;
    virtual void WriteTo(ostream& os) const = 0;
    /* Graphics */
    virtual void Draw() const = 0;
    /* mouse interactions */
    virtual void DoDialog() = 0;           // For double click
    virtual void TrackMouse() = 0;        // Mouse select and drag
    virtual Rect Area() const = 0;
    ...
};
```

Class `Thing` represents just an idea of a storable, drawable data element and so naturally it is simply a list of function names.

The situation is a little odd. We know that all `Things` can draw themselves, but we can't say how. The ability to draw is common, but the mechanism depends very much on the specialized nature of the `Thing` that is asked to draw itself. In class `Thing`, we have to be able to say "all `Things` respond to a `Draw()` request, specialized `Thing` subclasses define how they do this".

This is what the keyword `virtual` and the `odd = 0` notation are for.

Roughly, the keyword `virtual` identifies a function that a class wants to define in such a way that subclasses may later extend or otherwise modify the definition. The `=0` part means that we aren't prepared to offer even a default implementation. (Such undefined virtual functions are called "pure virtual functions".)

In the case of class `Thing`, we can't provide default definitions for any of the functions like `Draw()`, `WriteTo()` and so forth. The implementations of these functions vary too much between different subclasses. This represents an extreme case;

*virtual keyword and
=0 definition*

often you can provide a default implementation for a virtual function. This default definition describes what "usually" should be done. Subclasses that need to do something different can replace, or "override", the default definition.

virtual destructor

The destructor, `~Thing()`, does have a definition: `virtual ~Thing() { }`. The definition is an empty function; basically, it says that by default there is no tidying up to be done when a `Thing` is deleted. The destructor is virtual. Subclasses of class `Thing` may be resource managers (e.g. a subclass might allocate space for an object label as a separate character array in the heap). Such specialized `Things` will need destructors that do some cleaning up.

Thing* variables

A C++ compiler prevents you from having variables of type `Thing`:

```
Thing      aThing;           // illegal, Thing is an abstraction
```

This is of course appropriate. You can't have `Things`. You can only have instances of specialized subclasses. (This is standard whenever you have a classification hierarchy with abstract classes. After all, you never see "mammals" walking around, instead you encounter dogs, cats, humans, and horses – i.e. instances of specialized subclasses of class `mammal`). However, you can have variables that are `Thing*` pointers, and you can define functions that take `Thing&` reference arguments:

```
Thing *first_thing;
```

The pointer `first_thing` can hold the address of (i.e. point to) an instance of class `TextParagraph`, or it might point to a `Wire` object, or point to a `Battery` object.

Derived classes

Once you have declared class `Thing`, you can declare classes that are "based on" or "derived from" this class:

Public derivation tag

```
class TextParagraph : public Thing {
    TextParagraph(Point topleft);
    virtual ~TextParagraph();
    /* Disk I/O */
    virtual void ReadFrom(istream& is);
    virtual void WriteTo(ostream& os) const;
    /* Graphics */
    virtual void Draw() const;
    /* mouse interactions */
    virtual void DoDialog(); // For double click
    virtual void TrackMouse(); // Mouse select and drag
    virtual Rect Area() const;
    // Member functions that are unique to TextParagraphs
    void EditText();
    ...
private:
    // Data needed by a TextParagraph
    Point fTopLeft;
    char *fText;
    ...
}
```

```

};

class CircuitThing : public Thing {
    CircuitThing(int ident, Point where);
    virtual ~CircuitThing();
    ...
    /* Disk I/O */
    virtual void ReadFrom(istream& is);
    virtual void WriteTo(ostream& os) const;
    ...
    // Additional member functions that define behaviours
    // characteristic of all kinds of CircuitThing
    int GetId() const { return this->fId }
    virtual Rect Area() const {
        return Rect(
            this->flocation.x - 8, this->flocation.y - 8,
            this->flocation.x + 8, this->flocation.y + 8);
    }
    virtual double Current() const = 0;
    ...
protected:
    // Data needed by a CircuitThing
    int fId;
    Point flocation;
    char *fLabel;
    ...
};

```

Protected access specifier

In later studies you will learn that there are a variety of different ways that "derivation" can be used to build up class hierarchies. Initially, only one form is important. The important form is "public derivation". Both `TextParagraph` and `CircuitThing` are "*publicly derived*" from class `Thing`:

```

class TextParagraph : public Thing {
    ...
};

class CircuitThing : public Thing {
    ...
};

```

Public derivation acknowledges that both `TextParagraph` and `CircuitThing` are specialized kinds of `Things` and so code "using `Things`" will work with `TextParagraphs` or `CircuitThings`. This is exactly what we want for the example where the `Document` object has a list of "pointers to `Things`" and all its code is of the form `thing_ptr->DoSomething()`.

We need actual `TextParagraph` objects. This class has to be "concrete". The class declaration has to be complete, and all the member functions will have to be defined.

public derivation

TextParagraph, a concrete class

*CircuitThing, a
partially implemented
abstract class*

Naturally, the class declaration starts with the constructor(s) and destructor. Then it will have to repeat the declarations from class `Thing`; so we again get functions like `Draw()` being declared. This time they don't have those `= 0` definitions. There will have to be definitions provided for each of the functions. (It is not actually necessary to repeat the keyword `virtual`; this keyword need only appear in the class that introduces the member function. However, it is usually simplest just to "copy and paste" the block of function declarations and so have the keyword.) Class `TextParagraph` will introduce some additional member functions describing those behaviours that are unique to `TextParagraphs`. Some of these additional functions will be in the public interface; most would be private. Class `TextParagraph` would also declare all the private data members needed to record the data possessed by a `TextParagraph` object.

Class `CircuitThing` is an in between case. It is not a pure abstraction like `Thing`, nor yet is it a concrete class like `TextParagraph`. Its main role is to introduce those member functions needed to specify the behaviours of all different kinds of `CircuitThing` and to describe those data members that are possessed by all kinds of `CircuitThing`.

Class `CircuitThing` cannot provide definitions for all of those pure virtual functions inherited from class `Thing`; for instance it can't do much about `Draw()`. It should not repeat the declarations of those functions for which it can't give a definition. Virtual functions only get re-declared in those subclasses where they are finally defined.

Class `CircuitThing` can specify some of the processing that must be done when a `CircuitThing` gets written to or read from a file on disk. Obviously, it cannot specify everything; each specialized subclass has its own data to save. But `CircuitThing` can define how to deal with the common data like the identifier, location and label:

```
void CircuitThing::WriteTo(ostream& os) const
{
    // keyword virtual not repeated in definition
    os << fId << endl;
    os << fLocation.x << " " << fLocation.y << endl;
    os << fLabel << endl;
}

void CircuitThing::ReadFrom(istream& is)
{
    is >> fId;
    is >> fLocation.x >> fLocation.y;
    char buff[256];
    is.getline(buff, 255, '\n');
    delete [] fLabel; // get rid of existing label
    fLabel = new char[strlen(buff) + 1];
    strcpy(fLabel, buff);
}
```

These member functions can be used by the more elaborate `WriteTo()` and `ReadFrom()` functions that will get defined in subclasses. (Note the deletion of `fLabel`

and allocation of a new array; this is another of those places where it is easy to get a memory leak.)

The example illustrates that there are three possibilities for additional member functions:

```
int GetId() const { return this->fId }
virtual Rect    Area() const {
    return Rect(
        this->fLocation.x - 8, this->fLocation.y - 8,
        this->fLocation.x + 8, this->fLocation.y + 8);
    }
virtual double Current() const = 0;
```

Function `GetId()` is not a virtual function. Class `CircuitThing` defines an implementation (return the `fId` identifier field). Because the function is not virtual, subclasses of `CircuitThing` cannot change this implementation. You use this style when you know that there is only one reasonable implementation. for a member function.

*A non-virtual
member function*

Function `Area()` has a definition. It creates a rectangle of size 16x16 centred around the `fLocation` point that defines the centre of a `CircuitThing`. This might suit most specialized kinds of `CircuitThing`; so, to economise on coding, this default implementation can be defined at this level in the hierarchy. Of course, `Area()` is still a virtual function because that was how it was specified when first introduced in class `Thing` ("Once a virtual function, always a virtual function"). Some subclasses, e.g. class `Wire`, might need different definitions of `Area()`; they can override this default definition by providing their own replacement.

*A defined, virtual
member function*

Function `Current()` is an additional pure virtual function. The circuit simulation code will require all circuit elements know the current that they have flowing. But the way this gets calculated would be class specific.

*Another pure virtual
function*

Class `CircuitThing` declares some of the data members – `fId`, `fLabel`, and `fLocation`. There is a potential difficulty with these data members.

Access to members

These data members should not be `public`; you don't want the data being accessed from anywhere in the program. But if the data members are declared as `private`, they really are private, they will only be accessible from the code of class `CircuitThing` itself. But you can see that the various specialized subclasses are going to have legitimate reasons for wanting to use these variables. For example, all the different versions of `Draw()` are going to need to know where the object is located in order to do the correct drawing operations.

You can't use the "friend" mechanism to partially relax the security. When you define class `CircuitThing` you won't generally know what the subclasses will be so you can't nominate them as friends.

There has to be a mechanism to prevent external access but allow access by subclasses– so there is. There is a third level of security on members. In addition to `public` and `private`, you can declare data members and member functions as being

"protected" data

extra data members (and functions) protected. You would then also have to define the destructor as virtual.

The specification of the problem might disallow the user from dragging a wire or clicking on a wire to open a dialog box. This would be easily dealt with by making the `Area()` function of a `Wire` return a zero sized rectangle (rather than the fixed 16x16 rectangle used by other `CircuitThings`):

```
Rect Wire::Area() const
{
    return Rect(0, 0, 0, 0);
}
```

(The program identifies the `Thing` being selected by testing whether the mouse was located in the `Thing`'s area; so if a `Thing`'s area is zero, it can never be selected.) This definition of `Area()` overrides that provided by `CircuitThing`.

A `Wire` has to save all the standard `CircuitThing` data to file, and then save its extra data. This can be done by having a `Wire::WriteTo()` function that makes use of the inherited function:

```
void Wire::WriteTo(ostream& os)
{
    CircuitThing::WriteTo(os);
    os << fFirstEnd << " " << fSecondEnd << endl;
    ...
}
```

This provides another illustration of how inheritance structures may lead to small savings of code. All the specialized subclasses of `CircuitThing` use its code to save the identifier, label, and location.

23.6.3 BUT HOW DOES IT WORK?!

The example hierarchy illustrates that you can define a concept like `Thing` that can save itself to disk, and you can define many different specific classes derived from `Thing` that have well defined implementations – `TextParagraph::WriteTo()`, `Battery::WriteTo()`, `Wire::WriteTo()`. But the code for `Document` would be something like:

```
void Document::DoSave(ostream& out)
{
    out << thingList.Length() << endl;

    iterator il(thingList);
    il.First();
    while(!il.IsDone()) {
        Thing* thing_ptr = (Thing*) il.CurrentItem();
    }
}
```

```

        thing_ptr ->WriteTo(out);
        il.Next();
    }
}

```

The code generated for

```
thing_ptr ->WriteTo()
```

isn't supposed to invoke function `Thing::WriteTo()`. After all, this function doesn't exist (it was defined as `= 0`). Instead the code is supposed to invoke the appropriate specialized version of `WriteTo()`.

But which is the appropriate function? That is going to depend on the contents of `thingList`. The `thingList` will contain pointers to instances of class `TextParagraph`, class `Battery`, class `Switch` and so forth. These will be all mixed together in whatever order the user happened to have added them to the `Document`. So the appropriate function might be `Battery::WriteTo()` for the first `Thing` in the list, `Resistor::WriteTo()` for the second list element, and `Wire::WriteTo()` for the third. You can't know until you are writing the list at run-time.

The compiler can't work things out at compile time and generate the instruction sequence for a normal subroutine call. Instead, it has to generate code that works out the correct routine to use at run time.

virtual tables

The generated code makes use of tables that contain the addresses of functions. There is a table for each class that uses virtual functions; a class's table contains the addresses of its (virtual) member functions. The table for class `Wire` would, for example, contain pointers to the locations in the code segment of each of the functions `Wire::ReadFrom()`, `Wire::WriteTo()`, `Wire::Draw()` and so forth. Similarly, the virtual table for class `Battery` will have the addresses of the functions `Battery::ReadFrom()` and so on. (These tables are known as "virtual tables".)

In addition to its declared data members, an object that is an instance of a class that uses virtual functions will have an extra pointer data member. This pointer data member holds the address of the virtual table that has the addresses of the functions that are to be used in association with that object. Thus every `Wire` object has a pointer to the `Wire` virtual table, and every `Battery` object has a pointer to the `Battery` virtual table. A simple version of the scheme is illustrated in Figure 23.6

The instruction sequence generated for something like:

```
thing_ptr ->WriteTo()
```

involves first using the link from the object pointed to by `thing_ptr` to get the location of the table describing the functions. Then, the required function, `WriteTo()`, is "looked up" in this table to find where it is in memory. Finally, a subroutine call is made to the actual `WriteTo()` function. Although it may sound complex, the process requires only three or four instructions!

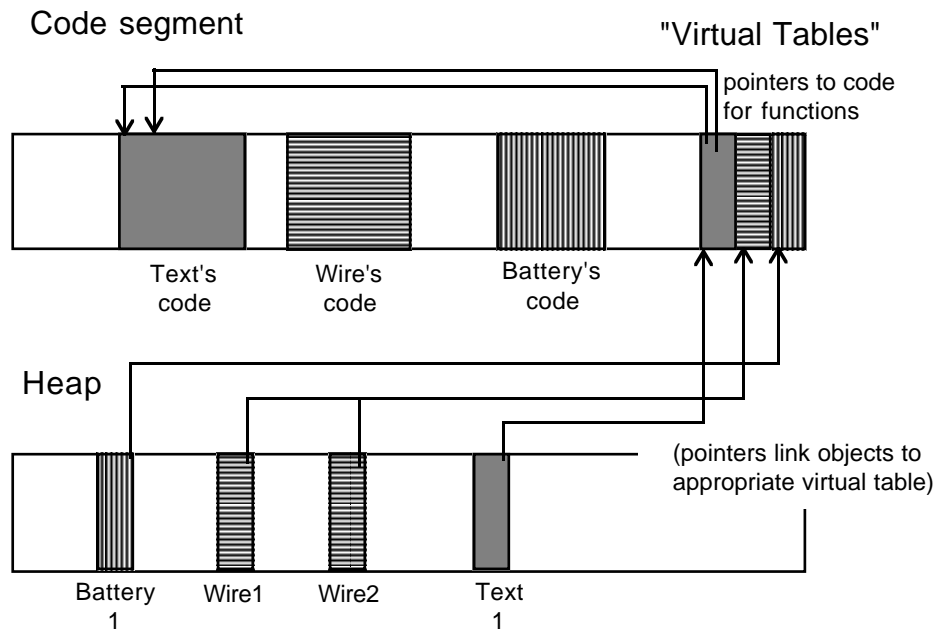


Figure 23.6 Virtual tables.

Function lookup at run time is referred to as "dynamic binding". The address of the function that is to be called is determined ("bound") while the program is running (hence "dynamically"). Normal function calls just use the machine's JSR (jump to subroutine) instruction with the function's address filled in by the compiler or linking loader. Since this is done before the program is running, the normal mechanism of fixing addresses for subroutine calls is said to use static binding (the address is fixed, bound, before the program is moving, or while it is static).

Dynamic binding

It is this "dynamic binding" that makes possible the simplification of program design. Things like `Document` don't have to have code to handle each special case. Instead the code for `Document` is general, but the effect achieved is to invoke different special case functions as required.

Another term that you will find used in relation to these programming styles is "polymorphism". This is just an anglicisation of two Greek words – poly meaning many, and morph meaning shape. A `Document` owns a list of `Things`; `Things` have many different shapes – some are text paragraphs, others are wires. A pointer like `thing_ptr` is a "polymorphic" pointer in that the thing it points to may, at different times, have different shapes.

Polymorphism

23.6.4 MULTIPLE INHERITANCE

You are not limited to single inheritance. A class can be derived from a number of existing base classes.

Multiple inheritance introduces all sorts of complexities. Most uses of multiple inheritance are inappropriate for beginners. There is only one form usage that you should even consider.

Multiple inheritance can be used as a "type composition" device. This is just a systematic generalization of the previous example where we had class `Thing` that represented the type "a drawable, storable, editable data item occupying an area of a window".

Instead of having class `Thing` as a base class with *all* these properties, we could instead factor them into separate classes:

```
class Storable {
public:
    virtual ~Storable() { }
    virtual void WriteTo(ostream&) const = 0;
    virtual void ReadFrom(istream&) const = 0;
    ...
};

void Drawable {
public:
    virtual ~Drawable() { }
    virtual void Draw() const = 0;
    virtual Rect Area() const = 0;
    ...
};
```

This allows "mix and match". Different specialized subclasses can derive from chosen base classes. As a `TextParagraph` is to be both storable and drawable, it can inherit from both base classes:

```
class TextParagraph : public Storable, public Drawable {
...
};
```

You might have another class, `Decoration`, that provides some pretty outline or shadow effect for a drawable item. You don't want to store `Decoration` objects in a file, they only get used while the program is running. So, the `Decoration` class only inherits from `Drawable`:

```
class Decoration : public Drawable {
...
};
```

As additional examples, consider class `Printable` and class `Comparable`:

```
class Printable {
public:
    virtual ~Printable() { }
    virtual void PrintOn(ostream& out) const = 0;
};

ostream& operator<<(ostream& o, const Printable& p)
{ p.PrintOn(o); return o; }
ostream& operator<<(ostream& o, const Printable *p_ptr)
{ p_ptr->PrintOn(o); return o; }

class Comparable {
public:
    virtual ~Comparable() { }
    virtual int Compare(const Comparable* ptr) const = 0;
    int Compare(const Comparable& other) const
        { return Compare(&other); }

    int operator==(const Comparable& other) const
        { return Compare(other) == 0; }
    int operator!=(const Comparable& other) const
        { return Compare(other) != 0; }
    int operator<(const Comparable& other) const
        { return Compare(other) < 0; }
    int operator<=(const Comparable& other) const
        { return Compare(other) <= 0; }
    int operator>(const Comparable& other) const
        { return Compare(other) > 0; }
    int operator>=(const Comparable& other) const
        { return Compare(other) >= 0; }
};
```

Class `Printable` packages the idea of a class with a `PrintOn()` function and associated global `operator<<()` functions. Class `Comparable` characterizes data items that compare themselves with similar data items. It declares a `Compare()` function that is a little like `strcmp()`; it should return -1 if the first item is smaller than the second, zero if they are equal, and 1 if the first is greater. The class also defines a set of operator functions, like the "not equals function" operator `!=()` and the "greater than" function `operator>()`; all involve calls to the pure virtual `Compare()` function with suitable tests on the result code. (The next chapter has some example `Compare()` functions.)

As noted earlier, another possible pure virtual base class would be class `Iterator`:

```
class Iterator {
public:
    virtual ~Iterator() { }
    virtual void First(void) = 0;
```

```
virtual void Next(void) = 0;  
virtual int  IsDone(void) const = 0;  
virtual void *CurrentItem(void) const = 0;  
};
```

This would allow the creation of a hierarchy of iterator classes for different kinds of data collection. Each would inherit from class `Iterator`.

Now inventing classes like `Storable`, `Comparable`, and `Drawable` is not a task for beginners. You need lots of experience before you can identify widely useful abstract concepts like the concept of storability. However you may get to work with library code that has such general abstractions defined and so you may want to define classes using multiple inheritance to combine different data types.

What do you gain from such use of inheritance as a type composition device?

Obviously, it doesn't save you any coding effort. The abstract classes from which you multiply inherit are exactly that – abstract. They have no data members. All, or most, of their member functions are pure virtual functions with no definitions. If any member functions are defined, then as in the case of class `Comparable`, these definitions simply provide alternative interfaces to one of the pure virtual functions.

You inherit, but the inheritance is empty. You have to define the code.

The advantage is not for the implementor of a subclass. Those who benefit are the maintenance programmers and the designers of the overall system. They gain because if a project uses such abstract classes, the code becomes more consistent, and easier to understand. The maintenance programmer knows that any class whose instances are to be stored to file will use the standard functions `ReadFrom()` and `WriteTo()`. The designer may be able to simplify the design by using collections of different kinds of objects as was done with the `Document` example.

23.6.5 USING INHERITANCE

There are many further complexities related to inheritance structures. One day you may learn of things like "private inheritance", "virtual base classes", "dominance" and others. You will discover what happens if a subclass tries to "override" a function that was not declared as `virtual` in the class that initially declared it.

But these are all advanced, difficult features.

The important uses of inheritance are those illustrated – capturing commonalities to simplify design, and using (multiple) inheritance as a type composition device. These uses will be illustrated in later examples. Most of Part V of this text is devoted to simple uses of inheritance.

24 Two more "trees"

Computer science students often have great difficulty in explaining to their parents why they are spending so much time studying "trees" and "strings". But I must impose upon you again. There are a couple more trees that you need to study. They are both just more elaborate versions of the binary tree lookup structure illustrated in Section 21.5.

The first, the "AVL" tree, is an "improved" binary tree. The code for AVL deals with some problems that can occur with binary trees which reduce the performance of the lookup structure. AVL trees are used for the same purpose as binary trees; they hold collections of keyed data in main memory and provide facilities for adding data, searching for data associated with a given key, and removing data. *AVL tree*

The second, the "BTree" tree, is intended for data collections that are too large to fit in main memory. You still have data records with a "key" field and other information; it is just that you may have hundreds of thousands of them. A BTree provides a means whereby most of the data are kept in disk files, but a fast search is still practical. The BTree illustrated is only slightly simplified; it is pretty close to the structures that are used to implement lookup systems for many large databases. *BTree*

These two examples make minor use of "inheritance" as presented in Section 23.6. The AVL tree is to store data items that are instances of some concrete class derived from class `KeyedItem`:

```
class KeyedItem {
public:
    virtual      ~KeyedItem() { }
    virtual long  Key(void) const = 0;
    virtual void  PrintOn(ostream& out) const { }
};
```

A `KeyedItem` is just some kind of data item that has a unique long integer key associated with it (it can also print itself if asked).

The BTree stores instances of a concrete class derived from class `KeyedStorableItem`:

```

class KeyedStorableItem {
public:
    virtual      ~KeyedStorableItem() { }
    virtual long  Key(void) const = 0;
    virtual void  PrintOn(ostream& out) const { }
    virtual long  DiskSize(void) const = 0;
    virtual void  ReadFrom(fstream& in) = 0;
    virtual void  WriteTo(fstream& out) const = 0;
};

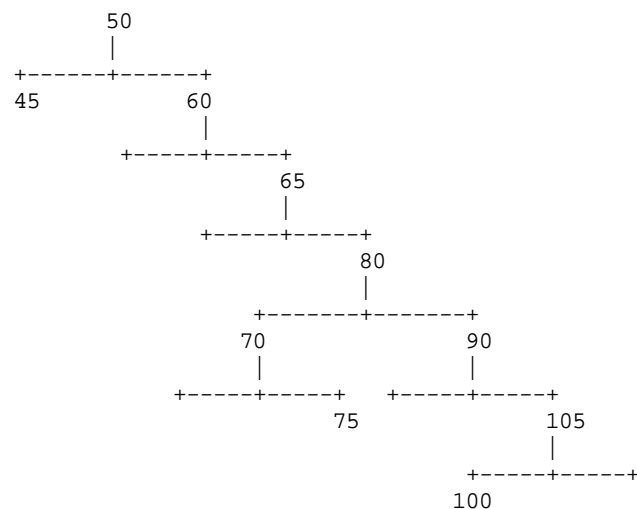
```

These data items must be capable of transferring themselves to/from disk files using binary transfers (`read()` and `write()` calls). On disk, the data items must all use the same amount of space. In addition to any other data that they possess, these items must have a unique long integer key value (disallowing duplicate keys simplifies the code).

24.1 AVL TREES

24.1.1 What's wrong with binary trees?

Take a look at a binary tree after a few "random" insertions. The following tree resulted when keyed data items were inserted with the keys in the following order: 50, 60, 65, 80, 70, 45, 75, 90, 105, 100:



The tree is a little out of balance. Most binary trees that grow inside programs tend to be imbalanced.

This imbalance does matter. A binary search tree is supposed to provide faster lookup of a keyed data item than does an alternative structure like a list. It is supposed

to give $O(\lg N)$ performance for searches, insertions, and deletions. But when a tree gets out of balance, performance decreases.

Fast lookup of keyed data items is a very common requirement. So the problems of simple binary search trees become important.

24.1.2 Keeping your balance

You can change the code so that the tree gets rearranged after every insertion and deletion. Figure 24.1 illustrates a couple of rearrangements that could be used to keep the tree balanced as those data items were inserted.

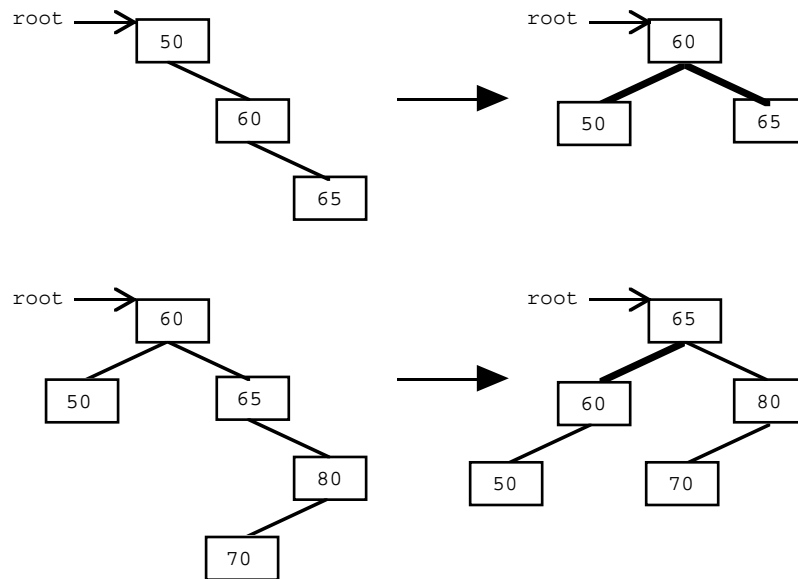


Figure 24.1 Rearranging a tree to maintain perfect balance.

Such rearrangements are possible. But a tree may take quite a lot of rearranging to get it balanced after some data item gets inserted. In some cases, you may have to alter almost all of the left- and right- subtree pointers. This makes the cost of rearrangement directly proportional to the number of items in the tree, i.e. $O(N)$ performance.

There is no point in trying to get a perfectly balanced tree if balancing cost are $O(N)$. Although rebalancing does keep search costs at $O(\lg N)$ you are interested in the overall costs, and thus $O(N)$ costs for rebalancing after insertions and deletions count against the $O(\lg N)$ searches.

However, it has been shown that a tree can be kept "more or less balanced". These more or less balanced trees have search costs that are $O(\lg N)$ and the cost of

rebalancing the tree until it is "more or less balanced" is also $O(\lg(N))$. (The analyses of the algorithms to demonstrate these costs is far too difficult for this introductory treatment).

There are many different schemes for keeping a binary search tree "more or less balanced". The best known was invented by a couple of Russians (Adelson-Velskii and Landis) back around 1960. They defined rules to characterize a "more or less balanced tree" and worked out the rearrangements that would be necessary if an insertion operation or a deletion operation destroyed the existing balance. The rearrangements are localized to the "vine" that leads from the root to the point where the change (insertion/ deletion) has just occurred and it is this that keeps the cost of rearrangements down to $O(\lg(N))$. A tree that satisfies their rules is called an AVL tree.

AVL tree The following definitions together characterize an AVL tree:

- Height of tree (or subtree):
The height of a binary tree is the length of the longest path from its root to a leaf.
- AVL property:
A node in a binary tree has the "AVL property" if the heights of the left and right subtrees are either equal or differ by 1.
- AVL tree:
An AVL tree is a binary tree in which every node has the AVL property.

Figure 24.2 illustrates some trees with examples of both AVL and non-AVL trees.

You can check a tree by starting at the leaf nodes. The "left and right subtrees" of a leaf node don't exist, or from a different perspective they are both size zero. Since they are both size zero they are equal so a leaf node has evenly balanced subtrees.

You climb up from a leaf node to its parent node and check its "left and right subtrees". If the node has two subtrees both with just leaves, then it is even. If it has only one leaf below it, it is either "left long" or "right long".

As you climb further up toward the root, you would need to keep a count of the longest path down through a link to its furthest leaf. As you reached each node, you would have to compare these longest paths down both the left and right subtrees from that node. If the longest paths are equal or differ by at most one, you can label the node as "even", or "left long", or "right long". If the lengths of the paths differ by more than one, as is the case with some of the nodes in the second pair of trees shown in Figure 24.2, then you have found a situation that violates the AVL requirements.

Checking the "AVL-ness" of an arbitrary tree might involve quite a lot of work. But if you are building up the tree, you can keep track of its state by just having an indicator on each node that says whether it is currently "even", "left long", or "right long". This information is sufficient to let you work out what rearrangements might be needed to maintain balance after a change like the addition or deletion of a node.

E = even
 L = Left subtree larger
 R = Right subtree larger

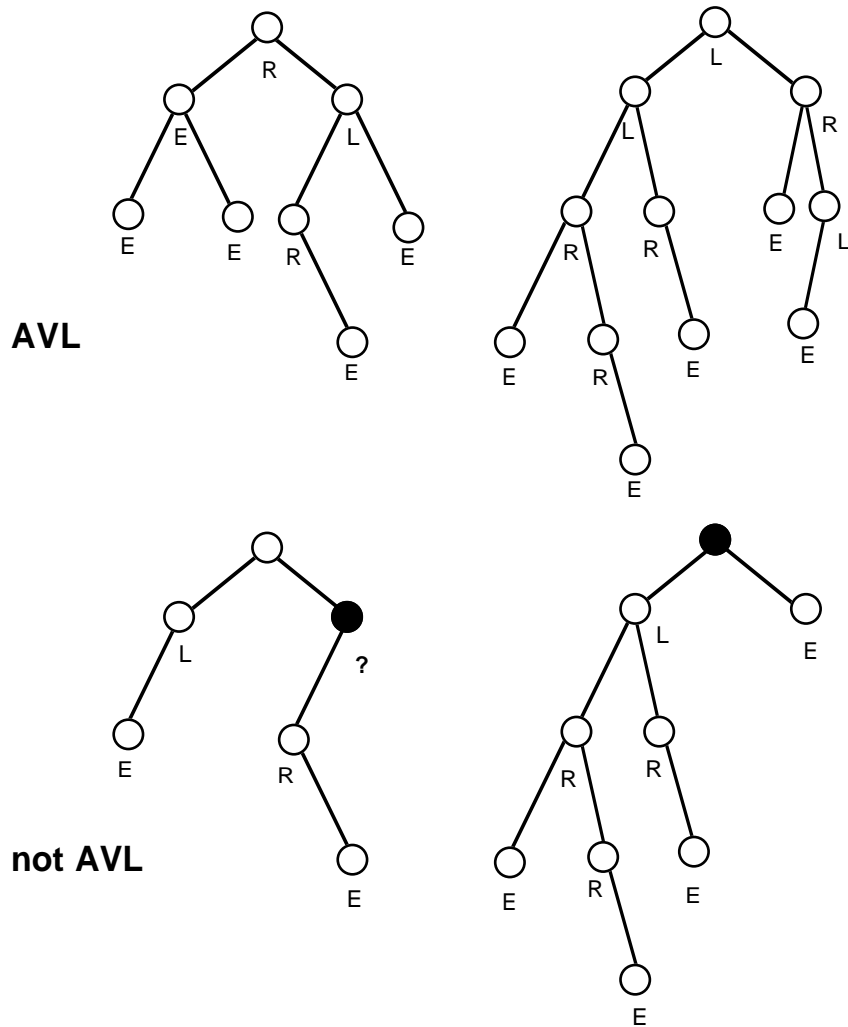


Figure 24.2 Example trees: AVL and not quite AVL.

Adding Nodes to an AVL tree

Figure 24.3 illustrates some of the possible situations that you might encounter when adding an extra node below an existing node. The numbers shown on the nodes

represent the keys for the data items associated with the tree node. Of course it is still essential to have the binary search tree property: data items whose keys are less than the key on a given node will be located in its left subtree, those whose keys are greater will be in its right subtree. (The key values will also be used as "names" for the nodes in subsequent discussions.)

The addition of a node can change the balance at every point on the path that leads from the root to the parent node where the new node gets attached. The first example shown in Figure 24.3 starts with all the existing nodes "even". The new data value must go to the left of the node 27; it was "even" but is going to become "left long". The value 6 has to go to the left of 19; so 19 which was "even" also becomes "left long".

The second example shown, the addition of 21, shows that additions sometimes restore local balance. Node 19 that was "left long" now gets back to "even". Node 27 is still "left long".

The next two examples shown in Figure 24.3 illustrate additions below node 6 that make nodes 27 and 19 both "left too long". They have lost their AVL properties. Some rearrangements are going to be performed to keep the tree "more or less balanced".

The final example shows another case where the tree will require rebalancing. Although this case does not need any changes in the immediate vicinity of the place where the new node gets added, changes are necessary higher up along the path to the root.

***Rearrangements to
the tree***

Adelson-Velskii and Landis explored all the possible situations that could arise when additions were made to a tree. Then, they worked out what local rearrangements could be made to restore "more or less balance" (i.e. the AVL property) in the immediate vicinity of an out of balance node.

The tree has to be reorganized whenever a node becomes "left too long" or "right too long". Obviously, there is a symmetry between the two cases and it is only necessary to consider one; we will examine the situation where a node is "left too long".

As illustrated in Figure 24.4, there are two variations. In one, the left subtree of the "left too long" node is itself "left long"; in the second variation, the node at the start of the left subtree is actually "right long". Adelson-Velskii and Landis sorted out the slightly different rearrangements of the tree structure that would be needed in these two cases. Their proposed local rearrangements are also shown in Figure 24.4.

***Local
rearrangements to fix
up an imbalanced
node***

The problem that has got to be resolved by these rearrangements is that the left branch of the tree below the current root (node 27) now has a height that is two greater than the right branch. The left branch must shrink, the right branch must grow. Since the tree must be kept ordered, the only way the right branch can grow and the left shrink is to push the current root node down into the right branch, replacing it at the root by an appropriate node from the left branch.

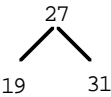
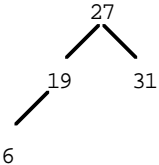
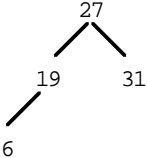
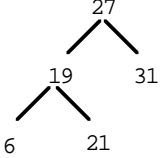
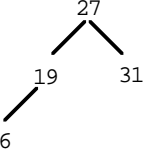
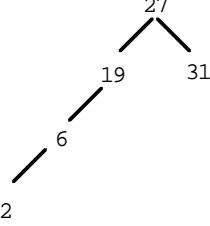
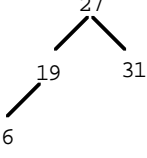
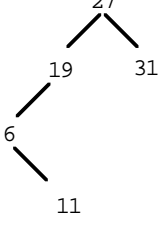
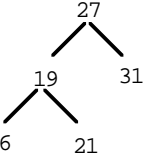
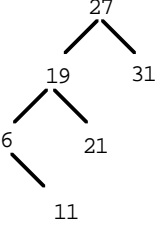
Existing Tree	Data added	Immediate result	AVL state
	6		Nodes 27 and 19 both "left long"
	21		Node 27 still "left long", 19 again "even"
	2		Nodes 27 and 19 both "left too long"! Tree must be rearranged.
	11		Nodes 27 and 19 both "left too long"! Tree must be rearranged.
	11		No problems for 6 or 19, but 27 is "left too long". Tree must be rearranged.

Figure 24.3 Effects of some additions to an AVL tree.

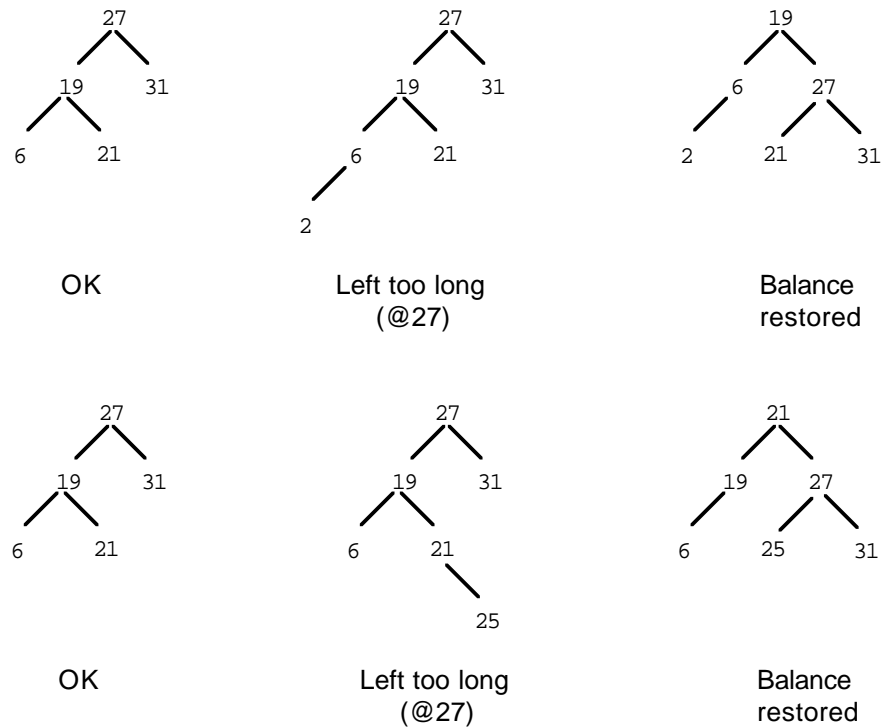


Figure 24.4 Manoeuvres to rebalance a tree after a node is added.

In the first case, the left subtree starting at node 19 has a small right subtree (just node 21). The tree can be rearranged by moving the current root node 27 down into the right tree, rehooking the small tree with node 21 to the left of node 27 and moving node 19 up to the root. The right subtree of the overall tree has grown by one (node 27 pushed into the subtree), and the left subtree has shrunk as node 19 moves upwards pulling its left subtree up with it. The tree is now balanced, all nodes are "even" except node 6 which is "left long". The order of nodes is maintained. Keys less than the new root value, 19, are down the left subtree, keys greater than 19 are in the right tree. Nodes with keys greater than 19 and less than 27 can be found by going first down the right tree from 19 to 27, then down the left tree below node 27.

In the second case, it is the right subtree below node 19 that is too large. This time the rearrangements must shorten this subtree while growing the right branch of the overall tree. Once again, node 27 gets pushed into the right subtree; this time being replaced by its left child's (19) right child (21). Any nodes attached below node 21 must be reattached to the tree. Things in its right subtree (e.g. 25) will have values greater than 21 and less than 27. This right subtree can be reattached as the left subtree of node 27 once this has been moved into position. The left subtree below node 21

(there is none in the example shown) would have nodes whose keys were less than 21 and greater than 19. This left subtree (if any) should be reattached as the right subtree below node 19 after node 21 is detached.

The tree is of course defined by pointer data members in the tree nodes. Rearrangements of the tree involve switching these pointers around. The principles are defined in the following algorithm outline. At the start, the pointer `t` is supposed to hold the address of the node that has got out of balance (node 27 in the example). This pointer `t` could be the "root pointer" for the entire tree; more commonly it will be either the "left subtree" pointer or the "right subtree" pointer of some other tree node. The value in this pointer gets changed because a subtree (if not the entire tree) is getting "re-rooted".

```
tLeft = t->left_subtree          // pointer to node 19
if(tLeft->balance is "left_long")
    // attach subtree starting at node 21 as left subtree of
    // node 27
    t->"left subtree link" =
        tLeft->"right subtree link"

    // attach subtree starting at node 27 as right subtree of
    // node 19
    tLeft->"right subtree link" = t;

    mark nodes referenced by t and tLeft as both now "even"

    // make node with 19 the new root
    change pointer t to refer to old left node

else
    // get pointer to node 21
    tLeftRight = tLeft->right_subtree /

    // left from 21, here NULL, gets attached to right of
    // 19
    tLeft->right_subtree = tLeftRight->left_subtree

    // make subtree starting at 19 as the left subtree of 21
    tLeftRight->left_subtree = tLeft;

    // make 21's right subtree (25), the new left subtree of 27
    t->left_subtree = tLeftRight->right_subtree

    // make subtree starting at 27 the new right subtree of 21
    tLeftRight->right_subtree = t;

    // Fix up balance records on nodes
    t->balance = (tLeftRight->balance == LEFT_LONG) ?
        RIGHT_LONG : EVEN;
    tLeft->balance = (tLeftRight->balance == RIGHT_LONG) ?
        LEFT_LONG : EVEN;
```

*First case shown in
Figure 24.4*

*Second case shown if
Figure 24.4*

```

tLeftRight->balance = EVEN

// Re-root the subtree, now starts with node 21
t = tLeftRight

```

**Organizing the
overall process**

The addition of a new node below an existing node may make that node "left long" or "right long" by changing the tree height. (When a node has one leaf below it, the addition of the other possible leaf does not change the tree's height, it simply puts the node back into even balance.) If the tree's height changes, this may necessitate rearrangement at the next level above where a node may have become "left too long" (or "right too long"). But it is possible that the change of height in one branch of a tree only produces an imbalance several levels higher up. For example, in the last example shown in Figure 24.3, the addition of node 11 did not cause problems at node 6, or at node 19, but did cause node 27 to become unbalanced.

The mechanism used to handle an insertion must keep track of the path from root to the point where a new node gets attached. Then after a new node is created, its data are filled in, and it gets attached, the process must work back up the path checking each node for imbalance, and performing the appropriate rebalancing rituals where necessary.

**Recursive driver
function**

The process of recording the path and then unwinding and checking the nodes is most easily handled using a recursive routine. It starts like the recursive insertion function shown for the simpler binary tree; the key for the new item, is compared with that in the current node and either the left or right branch is followed in the next recursive call. When there is no subtree, you have found the point where the new node is to be attached, so you build the node and hook it in. The difference from the simple recursive insertion function is that there is a lot of checking code that reexamines "balance" when the recursive call returns.

The algorithm is:

**Terminate recursion
when have position to
attach new node
Notify caller that tree
has grown**

```

insert(dataitem, link)
    if (the link is null)
        create a new node, fill in data
        set the link to point to the new node
        set a flag saying that the tree has grown larger
        return

    currentnode is tree node referenced by link

    if (the key value in the currentnode
        equals the key for the dataitem)
        report an error, duplicate keys are not allowed
        set flag to say that the tree is unchanged
        return

    if (the key value in the currentnode
        is smaller than the key for the dataitem)

```

<pre> recursively call insert, passing the dataitem and the left-link of currentnode if (tree size is now reported as changed) if current node was even, mark it left long leave "tree changed" flag set if current node was right long, mark it as even clear "tree changed" flag if current node was already left long do a local rearrangement clear "tree changed" flag return similar code for insertion in right subtree </pre>	<p><i>Recursive call down into appropriate subtree Fixup if tree has grown taller</i></p> <p><i>Do a local rearrangement if necessary</i></p>
--	---

Deletion of nodes

Deletions of nodes present two problems.

The first problem is identical to that encountered with the simple binary trees; it is easy to unhook a leaf node, or excise a node that only has one subtree, but you can't simply cut out a node that has two subtrees attached. The solution is identical to that used for the simpler binary tree. If a tree node has two subtrees, it is kept in the tree; its associated data are removed and replaced by data promoted from a node lower in the tree. The promoted data will be that with the largest key value smaller than the key of the data being removed (the "predecessor" of the deleted data). The node from which data was promoted is then removed from the tree.

***Deletion with
promotion***

The extra problem is of course that deletions may leave a node unbalanced. After a deletion is done, it is necessary to check back along the path to the root verifying that nodes are still balanced and performing any local rearrangement that might be needed for a node that has become unbalanced.

***Fixing up balance
after a deletion***

Naturally, the process is handled recursively. During the "inward" phase of recursion, the recursive function gets down to the node that must be removed (either the node with the deleted data, or the node from which data have been promoted). This inward recursion has function calls for each level, the local variables in the stack frame for each call define the various nodes traversed from the "root" to the node that is to be removed. Once found, the node can be excised and the tree marked as having its size changed.

***Recursive driver
function***

As the recursion is unwound, each node on the path gets its chance to consider the effects of the change in tree size on its balance. Sometimes, nodes will find themselves out of balance, and then there must be local rearrangements to the tree.

Because deletion is a fairly complex process, it has to be broken down into many separate functions. The basic driver function will be based on the following algorithm.

Code structure

The driver function will be given the key for the data item that is to be removed, and the root pointer. It involves a recursive search down through the tree. On each recursive call the argument *t* will be either the left or right subtree link from one of the tree nodes traversed.

<p><i>Hit null pointer, key wasn't present</i></p>	<pre>delete(bad_key, t) if(t is NULL) set flags to say tree not changed return</pre>
<p><i>If find key, use auxiliary function to do the delete</i></p>	<pre>if(bad_key equals key in node referenced by t) DeleteRec(t); return</pre>
<p><i>Otherwise recursively search down in left subtree</i></p>	<pre>if(bad_key < t->Key()) { Delete(bad_key, t->LeftLink())</pre>
<p><i>On unwind of recursion, do rebalancing</i></p>	<pre> if(fResizing == CHANGED_SIZE) Check_balance_after_Left_Delete(t); return</pre>
<p><i>Or search down right subtree</i></p>	<pre> Similar code dealing with right subtree</pre>

The auxiliary `DeleteRec()` function can sort out simple cases like a leaf node or a node with only one child, but will have to use other auxiliary functions to deal with more complex situations where data have to be replaced with information "promoted" from lower in the tree. Dealing with a node that has one child is simple, the link that lead to the node that is to be deleted is reset to point to the child. (A leaf, no children, doesn't have to be treated as a special case; the code handling nodes with one child also covers the case of no children.)

<p><i>Replace node with only one child by its sole child</i></p>	<pre>DeleteRec(t) if(t->RightLink() is NULL) x = t t = t->LeftLink() fResizing = CHANGED_SIZE delete x; else if(t->LeftLink() is NULL) similar</pre>
<p><i>Use auxiliary function to promote data from left subtree Left subtree may have shrunk, fix up</i></p>	<pre> else Del(t, t->LeftLink()) if(fResizing == CHANGED_SIZE) Check_balance_after_Left_Delete(t);</pre>

The auxiliary function `Del()` is given a pointer to the node that is being changed and, in the initial call, a pointer to the node's left subtree. It has to find the replacement

data that are to be promoted. The data will be that associated with the largest entry in this left subtree, i.e. the rightmost entry in the subtree. Naturally, `Del()` starts by recursively searching down to find the necessary data.

```
Del(t, r)
    if(r->RightLink() is not NULL)
        Del(t, r->RightLink())
        if(fResizing equals CHANGED_SIZE)
            Check_balance_after_Right_Delete(r);
    else {
        t->Replace(r->Data())

        // unlink the node from which data have been
        // promoted, replacing it by its left subtree
        // (if any exists)
        x = r
        r = r->LeftLink()

        // note tree size as changed
        // and get rid of discarded record
        fResizing = CHANGED_SIZE;
        delete x;
```

Code for the recursive search and the fixup as unwind recursion

Code that handles the promotion when data are found

The main issues still to be resolved are how to check the balance at a node after deletions in its left or right subtrees and how to fix things up if the balance is wrong.

Once again, Adelson-Velskii and Landis had to sort out all the possible situations and work out the correct rearrangements that would both keep the entries in the tree ordered, and the overall tree "more or less balanced".

AVL rearrangements for deletions

The checking part is relatively simple, the code is something like the following (which deals with the case where something has been removed from a node's right subtree):

```
Check_balance_after_Right_Delete(t)
    switch (t->Balance()) {
    case LEFT_LONG:
        // Right branch from current node was already
        // shorter than left branch, and it has shrunk.
        // Have to rebalance
        Rebalance_Right_Short(t);
        break;
    case EVEN:
        t->ResetBalance(LEFT_LONG);
        fResizing = UNCHANGED;
        break;
    case RIGHT_LONG:
        t->ResetBalance(EVEN);
        break;
    }
```

If the node had been "even", all that has happened is that it becomes "left long". This can simply be noted, and there is no need to consider changes at higher levels. If it had been "right long" it has now become "even". Its "right longedness" may have been balancing something else's "left longedness"; so it is possible that there will still be a need to make changes at higher levels. The real work occurs when you have a node that was already "left long" and whose right subtree has grown shorter. In that case, rebalancing operations are needed. There are symmetrically equivalent rebalancing operations for a node that was "right long" and whose left subtree has grown shorter.

The actual rearrangements are illustrated in Figure 24.5 for the case where a node was right long and whose left branch has shrunk.

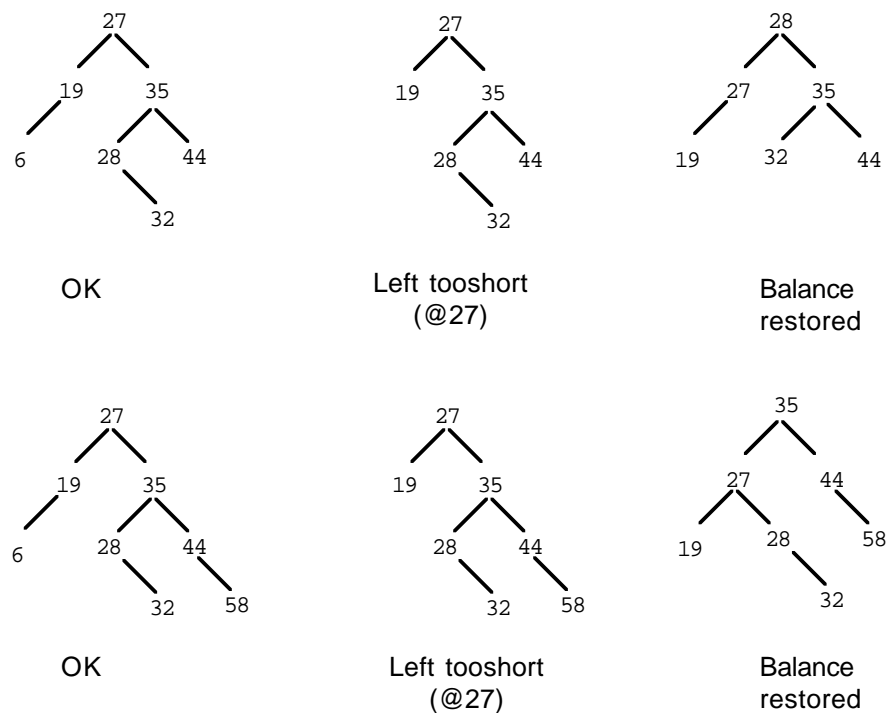


Figure 24.5 Rebalancing a tree after a deletion.

The rearrangements needed depend on the shape of the right subtree of the node that has become "left too short". If this right subtree is itself "left long" (the first example shown in Figure 24.5), then the tree is restored by pushing the current root down into the left branch (making that longer) and pulling a node up from the "left subtree" of the "right subtree" to make the new root for this tree (or subtree). If the right subtree is

evenly balanced (second example in Figure 24.5) or right long (not shown) then slightly different rearrangements apply.

Once again the rearrangements involve shifting pointers around and resetting the balance records associated with the nodes.

24.1.3 An implementation

The following code provides an example implementation of the AVL tree algorithms. The code implementing some functions has been omitted; as already noted, there are symmetrically equivalent "left" and "right" operations so only the code of one version need be shown.

The AVL tree is meant to be used to store pointers to any kind of object that is an instance of a class derived from abstract class `KeyedItem`. The header file should contain declarations for both `KeyedItem` and `AVLTree`. The implementation for class `AVLTree` uses an auxiliary class, `AVLTreeNode`, whose instances represent the tree nodes. This is essentially a private structure and is defined in the implementation file. Since class `AVLTree` has data members that are `AVLTreeNode*` pointers, there has to be a declaration of the form `"class AVLTreeNode;"` in the header file.

```
class KeyedItem {
public:
    virtual      ~KeyedItem() { }
    virtual long  Key(void) const = 0;
    virtual void  PrintOn(ostream& out) const { }
};

inline ostream& operator<<(ostream& out, const KeyedItem& d)
{ d.PrintOn(out); return out; }
inline ostream& operator<<(ostream& out, const KeyedItem* dp)
{ dp->PrintOn(out); return out; }

class AVLTreeNode;
```

*Definition of abstract
class (base class for
classes representing
data items)*

The public interface for class `AVLTree` is similar to that for the simple binary tree class. There are several private member functions that deal with issues like those rebalancing manoeuvres.

```
class AVLTree
{
public:
    AVLTree();
    ~AVLTree();

    int      NumItems(void) const;

    int      Add(KeyedItem* d);
```

class AVLTree

```

        KeyedItem    *Find(long key);
        KeyedItem    *Remove(long key);

private:
    void Insert1(KeyedItem* d, AVLTreeNode*& t);
    void Rebalance_Left_Long(AVLTreeNode*& t);
    void Rebalance_Right_Long(AVLTreeNode*& t);

    void Delete1(long bad_key, AVLTreeNode*& t);
    void Check_balance_after_Left_Delete(AVLTreeNode*& t);
    void Check_balance_after_Right_Delete(AVLTreeNode*& t);

    void Rebalance_Left_Short(AVLTreeNode*& t);
    void Rebalance_Right_Short(AVLTreeNode*& t);
    void DeleteRec(AVLTreeNode*& t);
    void Del(AVLTreeNode*& t, AVLTreeNode*& r);

    AVLTreeNode    *fRoot;
    int             fNum;

    KeyedItem      *fReturnItem;
    int             fResizing;
    int             fAddOK;
};

inline access function    inline int AVLTree::NumItems(void) const { return fNum; }

```

The principal data members are a pointer to the root of the tree and a count for the number of entries in the tree. The other three data members are essentially "work" variables for all those recursive routines that scramble around the tree; e.g. `fResizing` is the flag used to record whether there has been a change in the size of a subtree.

The tree does not "own" the data items that are inserted. The `Remove()` function returns a pointer to the data item associated with the "bad key". "Client code" that uses this AVL implementation can delete data items when appropriate. The destructor for the tree gets rid of all its `AVLTreeNode`s but leaves the data items untouched.

The implementation file starts with declarations of some integer flags and an enumerated type used to represent node balance. Then class `AVLTreeNode` is defined:

```

const    short    CHANGED_SIZE = 1;
const    short    UNCHANGED = 0;

enum eBALANCE { LEFT_LONG, EVEN, RIGHT_LONG };

Class AVLTreeNode
class AVLTreeNode {
public:
    AVLTreeNode(KeyedItem *d);

    AVLTreeNode*&    LeftLink(void);

```

```

        AVLTreeNode*&      RightLink(void);

        long               Key(void) const;
        eBALANCE           Balance(void) const;
        KeyedItem           *Data(void) const;

        void               Replace(KeyedItem *d);
        void               ResetBalance(eBALANCE newsetting);
    private:
        eBALANCE           fbalance;
        KeyedItem           *fData;
        AVLTreeNode         *fLeft;
        AVLTreeNode         *fRight;
};

```

An AVLTreeNode is something that has a balance factor, a pointer to some keyed data, and pointers to the AVLTreeNodes at the head of left and right subtrees. It provides three member functions that provide read access to data such as the balance factor, and two functions for explicitly changing the data associated with the node, or changing the balance.

In addition, there are the functions LeftLink() and RightLink(). These return references to the nodes left and right tree links. Because these functions return reference values, calls to these functions can appear on the left hand side of assignments. Although a little unusual, such functions help simplify the code of class AVLTree. Such coding techniques are somewhat sophisticated. You probably shouldn't yet attempt to write anything using such techniques, but you should be able to read and understand code that does.

Note functions that return reference values

All the member functions of class AVLTreeNode are simple; most can be defined as "inline".

```

AVLTreeNode::AVLTreeNode(KeyedItem *d)
{
    fbalance = EVEN;
    fLeft = fRight = NULL;
    fData = d;
}

inline eBALANCE AVLTreeNode::Balance(void) const
{ return fbalance; }

...
inline void AVLTreeNode::Replace(KeyedItem *d) { fData = d; }

...
inline AVLTreeNode*& AVLTreeNode::LeftLink(void)
{ return fLeft; }

```

Member functions of AVLTreeNode

The constructor for class AVLTree needs merely to set the root pointer to NULL and the count of items to zero. The destructor is not shown. It is like the binary tree

AVLTree

destructor illustrated at the end of Section 23.5.2. It does a post order traversal of the tree deleting each AVLTreeNode as it goes.

Constructor

```
AVLTree::AVLTree()
{
    fRoot = 0;
    fNum = 0;
}
```

The Find() function is just a standard search that chases down the left or right links as needed. It could be implemented recursively but because of its simplicity, an iterative version is easy:

AVLTree::Find()

```
KeyedItem* AVLTree::Find(long sought_key)
{
    AVLTreeNode *t = fRoot;
    for(; t != NULL; ) {
        if(t->Key() == sought_key) return t->Data();
        else
            if(t->Key() > sought_key) t = t->LeftLink();
        else
            t = t->RightLink();
    }
    return NULL;
}
```

The Add() and Remove() functions provide the client interface to the real working functions. They set up initial calls to the recursive routines, passing in the root pointer for the tree. Private data members are used rather than have the functions return their results; again this is just so as to slightly simplify the code in a few places.

AVLTree::Add() and AVLTree::Remove()

```
int AVLTree::Add(KeyedItem* d)
{
    fAddOK = 0;
    Insert1(d, fRoot);
    if(fAddOK)
        fNum++;
    return fAddOK;
}

KeyedItem *AVLTree::Remove(long bad_key)
{
    fReturnItem = NULL;
    Delete1(bad_key, fRoot);
    if(fReturnItem != NULL)
        fNum--;
    return fReturnItem;
}
```

The main driver routine for insertion is a straightforward implementation of the algorithm outlined earlier:

```
void AVLTree::Insert1(KeyedItem* d, AVLTreeNode*& t)
{
    if(t == NULL) {
        t = new AVLTreeNode(d);
        fResizing = CHANGED_SIZE;
        fAddOK = 1;
        return;
    }

    if(d->Key() == t->Key()) {
        // cout << "Duplicate entry ignored\n";
        fResizing = UNCHANGED;
        return;
    }

    if(d->Key() < t->Key()) {
        Insert1(d, t->LeftLink());
        if(fResizing == CHANGED_SIZE) {
            switch (t->Balance()) {
case LEFT_LONG:
                Rebalance_Left_Long(t);
                t->ResetBalance(EVEN);
                fResizing = UNCHANGED;
                break;
case EVEN:
                t->ResetBalance(LEFT_LONG);
                break;
case RIGHT_LONG:
                t->ResetBalance(EVEN);
                fResizing = UNCHANGED;
                break;
            }
        }
        return;
    }

    Similar code for right subtree
}
```

**Main driver routine
for insertion of extra
node
Item is not present,
make a node for it**

**Duplicates not
allowed**

**Insert small items in
left subtree**

**Rebalance if
necessary**

**Insert large items in
right subtree**

Functions like `Rebalance_Left_Long()` have a "reference to a `AVLTreeNode` pointer" as arguments. These functions may need to reset the pointer; this is why it gets passed by reference. The pointer used as an argument might be the tree's root pointer, `fRoot`, or the `fLeft` or `fRight` data member of some `AVLTreeNode` object.

*Rebalancing the tree
after an addition
Use balance of left
child to select
appropriate
rebalance manoeuvre*

```
void AVLTree::Rebalance_Left_Long(AVLTreeNode*& t)
{
    if((t->LeftLink()->Balance() == LEFT_LONG) {
        AVLTreeNode *tptr = t->LeftLink();
        t->LeftLink() = tptr->RightLink();
        tptr->RightLink() = t;
        t->ResetBalance(EVEN);
        t = tptr;
    }
    else {
        AVLTreeNode *tptr = t->LeftLink();
        AVLTreeNode *tptr2 = tptr->RightLink();

        tptr->RightLink() = tptr2->LeftLink();
        tptr2->LeftLink() = tptr;
        t->LeftLink() = tptr2->RightLink();
        tptr2->RightLink() = t;

        t->ResetBalance(
            (tptr2->Balance() == LEFT_LONG) ?
            RIGHT_LONG : EVEN);

        tptr->ResetBalance(
            (tptr2->Balance() == RIGHT_LONG) ?
            LEFT_LONG : EVEN);

        t = tptr2;
    }
}
```

The code highlighted in bold shows calls to the "reference returning" function `LeftLink()`. The first call is on the right hand side of an assignment so the compiler arranges to get the value from the `fLeft` field of the object pointed to by `t`. In the second case, the call is on the left of an assignment. The compiler gets the address of `t`'s `fLeft` data field, and then stores, in this location, the value obtained by evaluating `tptr->RightLink()`. The code highlighted in italics illustrates where the function is changing the value of the pointer passed by reference (in effect, "re-rooting" the current subtree).

The corresponding function `Rebalance_Right_Long()` is similar and so is not shown.

Deletion functions

The main driver routine for deletion and the functions for checking balance after left or right deletions are simple to implement from the outline algorithms given earlier. The `DeleteRec()` function (which removes nodes with one or no children and arranges for promotion of data in other cases) is:

```
void AVLTree::DeleteRec(AVLTreeNode*& t)
{
    fReturnItem = t->Data();
    if(t->RightLink() == NULL) {
```



```

        AVLTreeNode *x = t;
        t = t->LeftLink();
        fResizing = CHANGED_SIZE;
        delete x;
    }
    else
    if(t->LeftLink() == NULL) {
        AVLTreeNode *x = t;
        t = t->RightLink();
        fResizing = CHANGED_SIZE;
        delete x;
    }
    else {
        Del(t,t->LeftLink());
        if(fResizing == CHANGED_SIZE)
            Check_balance_after_Left_Delete(t);
    }
}

```

The Del() function deals with the processes of finding the data to promote and the replacement action:

```

void AVLTree::Del(AVLTreeNode*& t, AVLTreeNode*& r)
{
    if(r->RightLink() != NULL) {
        Del(t,r->RightLink());
        if(fResizing == CHANGED_SIZE)
            Check_balance_after_Right_Delete(r);
    }
    else {
        AVLTreeNode *x;
        t->Replace(r->Data());
        x = r;
        r = r->LeftLink();
        fResizing = CHANGED_SIZE;
        delete x;
    }
}

```

*Recursive search
down to replacement
data*

*Doing the
replacement*

There are symmetrically equivalent routines for rebalancing a node after deletions in its left or right subtrees. This is the code for the case where the right subtree has shrunk:

```

void AVLTree::Rebalance_Right_Short(AVLTreeNode*& t)
{
    AVLTreeNode* tptr = t->LeftLink();

    if(tptr->Balance() != RIGHT_LONG) {
        t->LeftLink() = tptr->RightLink();
        tptr->RightLink() = t;
    }
}

```

*Code to rebalance
after a deletion*

```

        if(tptr->Balance() == EVEN) {
            t->ResetBalance(LEFT_LONG);
            tptr->ResetBalance(RIGHT_LONG);
            fResizing = UNCHANGED;
        }
        else {
            t->ResetBalance(EVEN);
            tptr->ResetBalance(EVEN);
        }
        t = tptr;
    }
else {
    AVLTreeNode *tptr2 = tptr->RightLink();
    tptr->RightLink() = tptr2->LeftLink();
    tptr2->LeftLink() = tptr;
    t->LeftLink() = tptr2->RightLink();
    tptr2->RightLink() = t;
    t->ResetBalance((tptr2->Balance() == LEFT_LONG) ?
                    RIGHT_LONG : EVEN);
    tptr->ResetBalance((tptr2->Balance() == RIGHT_LONG) ?
                      LEFT_LONG : EVEN);
    t = tptr2;
    tptr2->ResetBalance(EVEN);
}
}

```

The functions not shown are all either extremely simple or are the left/right images of functions that have been given.

24.1.4 Testing!

Just look at the AVL algorithm! It has special cases for left subtrees becoming too long on their own left sides, and left subtrees becoming too long on their right subtrees, code for right branches that are getting shorter, and It has special cases where data elements must be promoted from other tree cells. These operations may involve searches down branches of trees. The tree has to be quite large before there is even a remote possibility of some these special operations being invoked.

The simpler abstract data types like the lists and the queues shown in Chapter 21 could be tested using small interactive programs that allowed the tester to exercise the various options like getting the length or adding an element. Such an approach to testing something like the AVL tree is certain to prove inadequate. When arbitrarily selecting successive addition and deletion operations, the tester simply won't pick a sequence that exercises some of the more exotic operations.

The approach to testing has to be more systematic. You should provide a little interactive program, like those in Chapter 21, that can be used for some preliminary tests. A second non-interactive test program would then be needed to thoroughly test

all aspects of the code. This second program would be used in conjunction with a "code coverage tool" like that described in Chapter 14.

Both the test programs would need some data objects that could be inserted into the tree. You would have to define a class derived from class `KeyedItem`, e.g. class `TextItem`:

```
class TextItem : public KeyedItem
{
public:
    TextItem(const char* info, long k);
    ~TextItem();
    long Key(void) const;
    void PrintOn(ostream& out) const;
private:
    char    *fText;
    long    fk;
};

TextItem::TextItem(const char *info, long k)
{
    fk = k;
    fText = new char[strlen(info) + 1];
    strcpy(fText, info);
}

TextItem::~TextItem()
{
    delete [] fText;
}

void TextItem::PrintOn(ostream& out) const
{
    out << "[ " << fText << ", " << fk << " ] ";
};

long TextItem::Key(void) const { return fk; }
```

TextItem – a class of object that can be put into an AVLTree

A `TextItem` object is just something that holds a long integer key and a string. The interactive test program can get the user to enter these data; the way the data are generated and used in the automated program is explained later.

The main function for an interactive test program, `HandTest()`, is shown below. It has the usual structure with a loop offering user commands.

```
AVLTree gTree;

void HandTest(void)
{
    KeyedItem* d;
    for(;;) {
```

Get command

```

char ch;
cout << "Action (a = Add, d = Delete, f = Find, "
        " p = Print Tree, q = Quit) : ";
cin >> ch;

switch (ch) {

```

An "add" command results in the creation of an extra `TextItem` that gets put in the tree. (The function `AVLTree::Add()` returns a success/failure indicator. A failure should only occur if an attempt is made to insert a record with a duplicate key. If the add operation fails, the "duplicate" record should be deleted.)

Adding TextItems

```

case 'a':
case 'A':

    {
    long key;
    char buff[100];
    cout << "Key : " ; cin >> key;
    cout << "Name : "; cin >> buff;
    d = new TextItem(buff, key);
    if(gTree.Add(d) != 0)
        cout << "Inserted OK" << endl;
    else {
        cout << "Duplicate " << endl;
        delete d;
    }
    }
break;

```

A "delete" command gets the key for the `TextItem` to be removed then invokes the tree's remove function. Function `AVLTree::Remove()` returns `NULL` if an item with the given key was not present. If the item was found, a pointer is returned. The item can then be deleted.

There would also be a "find" command (not shown, is trivial to implement), a "quit" command, and possibly a "print" command. During testing it would be useful to get the tree displayed so it might be worth adding an extra public member function `AVLTree::PrintTree()`. The algorithm required will be identical to that used for the simpler binary tree.

Deleting TextItems

```

case 'd':
case 'D':

    {
    long bad;
    cout << "Enter key of record to be removed";
    cin >> bad;
    d = gTree.Remove(bad);
    if(d == NULL)
        cout << "No such record" << endl;
    }

```

```

        else {
            cout << "Removing " << *d << endl;
            delete d;
        }
    }
    break;

case 'f':
case 'F':

    ...
    ...
    break;

case 'p':
case 'P':

    gTree.PrintTree();
    break;

case 'q':
case 'Q':

    return;

default:

    cout << "?" << endl;
}
}

```

Hand testing will never build up the large complex trees where less common operations, like promotion of data, get fully tested. You need code that performs thousands of insertion, find, and deletion operations on the tree and which checks that each operation returns the correct result.

This is not as hard as it might seem. Basically, you need a testing function that starts by loading some records into the tree and then "randomly" chooses to add more records, delete records, or search for records. The function will need a couple of control parameters. One determines the number of cycles (should be 10000 to 20000). The other parameter, `testsize`, determines the range used for keys; there is a limit, `kTESTMAX`, for this parameter. The use of the `testsize` parameter is explained below.

Mechanism for an automated test

```

void AutoTest()
{
    int testsize;
    int runsize;
    cout << "Enter control parameters for auto-test ";
    cin >> testsize >> runsize;
    assert(testsize > 1);
    assert(testsize < kTESTMAX);

    Initialize(testsize);
    for(int i=0; i < testsize / 2; i++)
        Add(testsize);

    for(int j=0; j < runsize; j++) {

```

```

        int r = rand() % 4;
        switch(r) {
        case 0:      Add(testsize); break;
        case 1:      Find(testsize); break;
        case 2:
        case 3:
            Remove(testsize); break;
        }
        cout << "Test complete, counters of actions: " << endl;
        for(i = 0; i < 6; i++)
            cout << i << ": " << gCounters[i] << endl;
    }

```

As you can see, the loop favours removal operations. This makes it likely that at some stage all records will be removed from the tree. There are often obscure special cases related to collections becoming empty and then being refilled so it is an aspect that you want to get checked.

Function `AutoTest()` uses the auxiliary functions, `Add()`, `Find()`, and `Remove()` to do the actual operations. These must be able to check that everything works correctly.

***Keeping track of
valid keys***

Correct operation can be checked by keeping track of the keys that have been allocated to `TextItem` records inserted in the tree. When creating a new `TextItem`, the test program gives it a "random" key within the permitted range:

```

TextItem *MakeATextItem(int testsize)
{
    int r = rand() % testsize;
    return new TextItem("XXXX", r);
}

```

The `Add()` function keeps track of the keys that it has allocated and for which it has inserted a record into the tree. (It only needs an array of "booleans" that record whether a key has been used):

```

const int kTESTMAX      = 5000;
short      gUsed[kTESTMAX];

```

If the same randomly chosen key has already been used, an addition operation should fail; otherwise it should succeed. The `Add()` function can check these possibilities. If something doesn't work correctly, the program can stop after generating some statistics on the tree (function `ReportProblem()`, not shown). If things seem OK, the function can increment a count of operations tested:

```

const int ADD_OK        = 0;
const int ADD_DUP       = 1;
const int FIND_OK       = 2;
const int FIND_EMPTY    = 3;

```

```

const int REMOVE_OK      = 4;
const int REMOVE_FAIL   = 5;

long      gCounters[6];      // record test operations

void Add(int testsize)
{
    TextItem *t = MakeATextItem(testsize);
    long k = t->Key();
    if(gUsed[k] == 0) {
        /* Should get a successful insert */
        if(gTree.Add(t) != 0) gCounters[ADD_OK]++;
        else {
            cout << "Got a 'duplicate' response when"
                  "should have been able to add"
                  << endl;
            ReportProblem(k);
            exit(1);
        }
        gUsed[k] = 1;
    }
    else {
        /* Should get a duplicate message */
        if(gTree.Add(t) == 0) {
            gCounters[ADD_DUP]++;
            delete t;
        }
        else {
            cout << "Failed to notice a duplicate" << endl;
            ReportProblem(k);
            exit(1);
        }
    }
}

```

"Fresh" key, Add() should work

Mark key in use

Already used key, Add() should fail

Of course, the `gUsed[]` and `gCounters[]` arrays have to be initialized. The `Initialize()` function is called at the start of the `AutoTest()` function:

```

void Initialize(int testsize)
{
    for(int i = 0; i < testsize; i++)
        gUsed[i] = 0;
    for(int j = 0; j < 6; j++)
        gCounters[j] = 0;
}

```

The functions `Find()` and `Remove()` can also use the information in the `gUsed[]` array. Function `Find()` (not shown) randomly picks a key, inspects the corresponding `gUsed[]` array to determine whether or not a record should be found, then attempts the `gTree.Find()` operation and verifies whether the result is as expected.

`Remove()` is somewhat similar. However, if it successfully removes a `TextItem`, it must also delete it and clear the corresponding entry in the `gUsed[]` array:

```
void Remove(int testsize)
{
    long k = rand() % testsize;
    KeyedItem *d = gTree.Remove(k);
    if(gUsed[k] == 0) {
        /* Remove operation should have failed */

        if(d == NULL) gCounters[REMOVE_FAIL]++;
        else {
            cout << "Removed a thing that wasn't there"
                  << endl;
            ReportProblem(k);
            exit(1);
        }
    }
    else {
        /* Remove operation should succeed */
        if(d != NULL) {
            gCounters[REMOVE_OK]++;
            delete d;
            gUsed[k] = 0;
        }
        else {
            cout << "Failed to find and remove a data"
                  << endl;
            ReportProblem(k);
            exit(1);
        }
    }
}
```

Trying to remove item with non-existent key

Trying to remove an item that should be present

Runs can be made with different values for the `testsize` parameter. Large values (3000 - 5000) result in complicated deep trees (after all, the first step involves filling the tree with `testsize/2` items). These trees have cases where data have to be promoted from remote nodes, leading to a long sequence of balance checks following the deletion.

Small values of the `testsize` parameter keep the tree small, force lots of "duplicate" checks, and make it more likely that all elements will be deleted from the tree at some stage in the processing.

The test program can use the `gCounters[]` counts to provide some indication as to whether the tests are comprehensive. But this is still not adequate. You may know that your tree survived 10,000 operations but you still can't be certain that all its functions have been executed.

Complex algorithms like the AVL code require testing with the code coverage tools. The code was run on a Unix system where the `tcov` tool (Chapter 14) was available.

Several different runs were performed and the final accumulated statistics were analyzed using tcov. A fragment of tcov's output is as follows:

	void AVLTree::Delete1(long bad_key, AVLTreeNode*& t)	<i>Output from code coverage tool</i>
122513 -> {		
10217 ->	if(t == NULL) {	
	fResizing = UNCHANGED;	
	return;	
	}	
112296 ->	if(bad_key == t->Key()) {	
4902 ->	DeleteRec(t);	
	return;	
	}	
107394 ->	if(bad_key < t->Key()) {	
56338 ->	Delete1(bad_key, t->LeftLink());	
	if(fResizing == CHANGED_SIZE)	
3070 ->	Check_balance_after_Left_Delete(t);	
3070 ->	return;	
	}	
51056 ->	Delete1(bad_key, t->RightLink());	
	if(fResizing == CHANGED_SIZE)	
2996 ->	Check_balance_after_Right_Delete(t);	
2996 ->	return;	
	}	
	void AVLTree::Check_balance_after_Left_Delete(
	AVLTreeNode*& t)	
4338 -> {		
	switch (t->Balance()) {	
	case LEFT_LONG:	
1480 ->	t->ResetBalance(EVEN);	
	break;	
	case EVEN:	
2256 ->	t->ResetBalance(RIGHT_LONG);	
	fResizing = UNCHANGED;	
	break;	
	case RIGHT_LONG:	
602 ->	Rebalance_Left_Short(t);	
	break;	
	}	
4338 -> }		

Such results give greater confidence in the code. (The tcov record, along with the test programs, form part of the "documentation" that you should provide if your task was to build a complex component like an AVL tree.)

Code coverage by hand

If you can't get access to something like `tcov`, you have to achieve something similar. This means adding conditionally compiled code. You will have to define a global array to hold the counters:

```
#ifdef TCOVING
int __my__counters[1000];
#endif
...
```

and you will have to edit every function, and every branch statement within a function, to increment a counter:

```
void AVL::Insert(
{
#ifdef TCOVING
    __my__counters[17]++;
#endif
```

Finally, you have to provide a function that prints contents of the table when the program terminates.

Unfortunately, this process is very clumsy. It is easy to make mistakes such as forgetting to associate a counter with some branch, or to have two bits of code that use the same counter (this is quite common if the main code is still being finalized at the same time as being tested). The printouts of counts aren't directly related to the source listings, the programmer has to read the two outputs together.

Further, the entire process of hand editing is itself error prone. Careless editing can easily cut a controlled statement from the `if()` condition that controls it!

The best solution is to use a code coverage tool on some other platform while pressing your IDE supplier for such a tool in the next version of the software. (The supplier should oblige, code coverage tools are as easy, or easier, to add to a compiler than the time profilers which are commonly available.)

Don't really trust it even if `tcov` says its tested

A final warning, even if `tcov` (or equivalent) says that you've tested all the branches in your code, you still can't be certain that it is correct. You really should try that exercise at the end of Chapter 21 where "live" listcells were "deleted" from a list and the program still ran "correctly" (or at least it ran long enough to convince any typical tester that it was working correctly).

Memory problems (leaks, incorrect deletions, other use of "deleted" data) are the reason for having an automated program that performs tens of thousands operations. If the tests are lengthy enough you have some chance of forcing memory bugs to manifest themselves (by crashing the program several minutes into a test run). Unfortunately, some memory related bugs are "history dependent" and will only show up with specific sequences of operations; such bugs are exceptionally difficult to find.

Testing can establish that your code has bugs; testing can not prove that a program is bug free. Despite that, it is better if code is extensively tested rather than left untested.

24.2 BTREE

24.2.1 Multiway trees

You are not restricted to "binary trees", there are alternative tree structures. In fact, there are numerous forms of "multiway tree". They all serve much the same role as an AVL tree. They are "lookup" structures for keyed data. These trees provide `Add()`, `Find()`, and `Remove()` functions. They provide a guarantee that their performance on all operations is close to $O(\lg(N))$ (with N the number of items stored in the tree).

These trees have more than one key in each of their tree nodes and, consequently, more than two links down to subtrees. The data inserted into a tree are kept ordered; data items with small keys are in "left subtrees", data with "middling" keys can be found down in other subtrees, and data with large keys tend to be in "right subtrees". The trees keep themselves "more or less balanced" by varying the number of data items stored in each node. The path from root to leaf is kept the same for all leaves in the tree (a major factor in keeping costs $O(\lg(N))$).

Although the structures of the nodes, and the forms for the trees, are radically different from those in the AVL tree, there are some similarities in the overall organization of the algorithms.

An operation like an insertion is done by recursively chasing down through the tree to the point where the extra data should go. The new item is added. Then, as the recursion unwinds, local "fix ups" are performed on each tree node so as to make certain that they all store "appropriate" numbers of data items and links.

What is an "appropriate" number of data items for a node? How are "fix ups" done when nodes don't have appropriate number of items? Each different form of multiway tree has slightly different rules with regard to these issues.

Deletions are also handled in much the same way as in AVL and binary trees. You search, using a recursive routine, for the item that is to be removed. Items can easily be cut out of "leaf nodes". Items that are in "internal nodes" (those with links down to subtrees) have to be replaced by data "promoted" from a leaf node lower in the tree (the successor, or predecessor, item with the key immediately after, or before, that of the item being removed). If data are promoted from another node, the original copy of the promoted data must then be cut out from its leaf node.

Once the deletion step has been done, the recursion unwinds. As in the case of insertion, the "unwinding" process must "fix up" each of the nodes on the path back to the root. Once again, different forms of multiway tree have slightly different "fix up" rules.

Of course, searches are fairly simple. You can chase down through the tree following the links. As nodes can have more than one key, there is an iterative search through the various keys in each node reached.

2-3 Trees

You will probably get to study different multiway trees sometime later in your computer science career. Here, only one simple version need be considered. It is usually called a "two-three" tree because each node can hold two keys and three links down to subtrees. It has similarities to, and can act as an introduction to the BTree which the real focus of this section.

For simplicity, these 2-3 trees will be shown storing just integer keys. If you were really implementing this kind of tree, the "integer key" fields used in the following discussion would be replaced by structures that comprised an integer key and a pointer to the real data item (as was done in the binary tree example in Chapter 21).

Figure 24.6 illustrates the form of a tree node for this simplified 2-3 tree, while Figure 24.7 illustrates an actual tree built using these nodes. The tree node has an array of two longs for the keys, an array of three pointers for the links to subtrees and a "flag" indicating whether it is a "2-node" (one key, two links used), or a "3-node" (both keys and all three links used). In leaf nodes, the link fields will be NULL.

Search The search algorithm is simple:

```
compare search key with 1st (only?) key in node
if equal
    report record as found
```

Node structure:

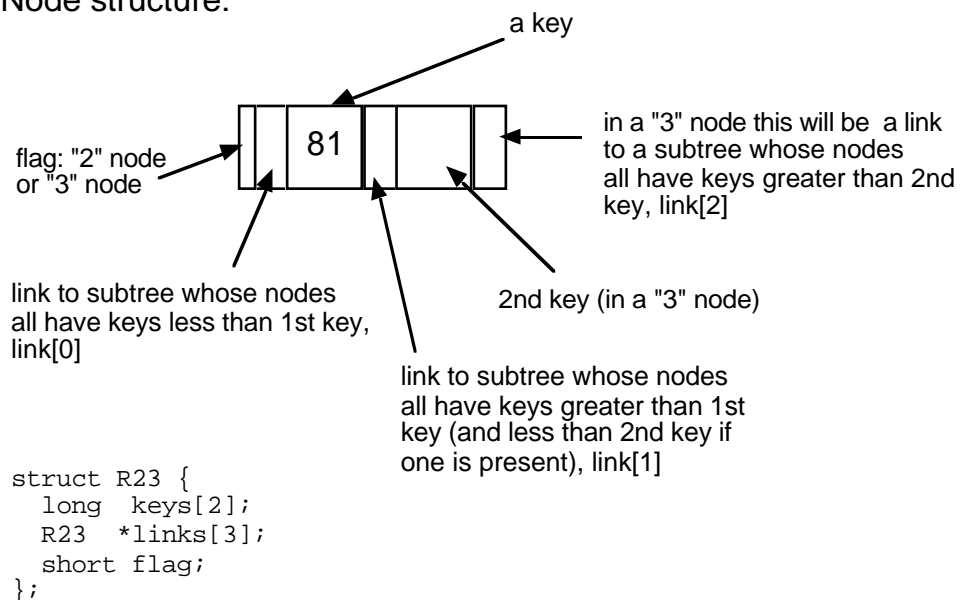


Figure 24.6 Structure of a node for a "2-3" multiway tree.

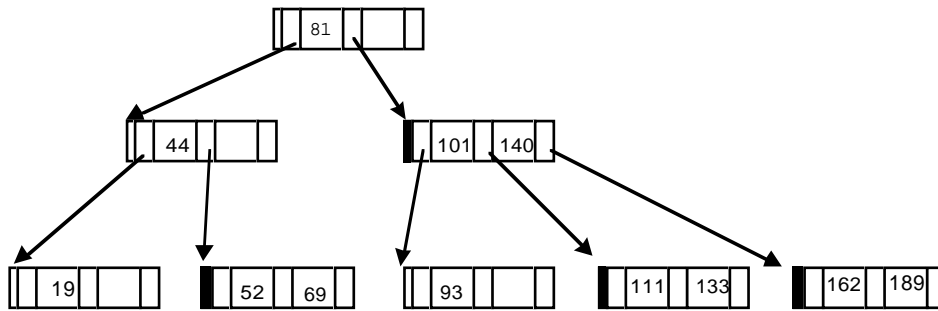


Figure 24.7 An example "2-3" tree.

```

else
  if less
    search down subtree 0
  else
    if greater
      if this is a 2-node then search down subtree 1
    else
      compare search key with 2nd key in node
      if equal
        report record as found
      else
        if less
          search down subtree 1
        else
          search down subtree 2

```

New keys are inserted into leaf nodes. In some cases this is easy. In the example tree shown in Figure 24.7, insertion of the key value 33 is easy. The search for the correct place goes left down `link[0]` from the node with key 81, and again left down `link[0]` from the node with key 44. The next node reached is a leaf node. This one has only one entry, 19, so there is room for the key 33. The key can be inserted and the node's flag changed to mark it as a "3-node" (two keys, potentially three links though currently all these links are `NULL`).

Insertion of the key value 71 would be more problematic. Its place is in the leaf node with the keys 52 and 69; but this node is already fully occupied. Insertion into a full leaf node is handled by "splitting the node". There will then be three keys, and two leaf nodes. The least valued of the three keys goes in the "left" node resulting from the split; the key with the largest value goes in the "right" node; while the middle valued key gets moved up one level to the parent node. This is illustrated in Figure 24.8.

The two leaf nodes are both "2-nodes", while their parent (the node that used to hold just key 44) now has two keys and three links and so it is now a "3-node".

The results of two additional insertions are illustrated in Figures 24.9 and 24.10.

Insertion

Splitting nodes to make room for another inserted key

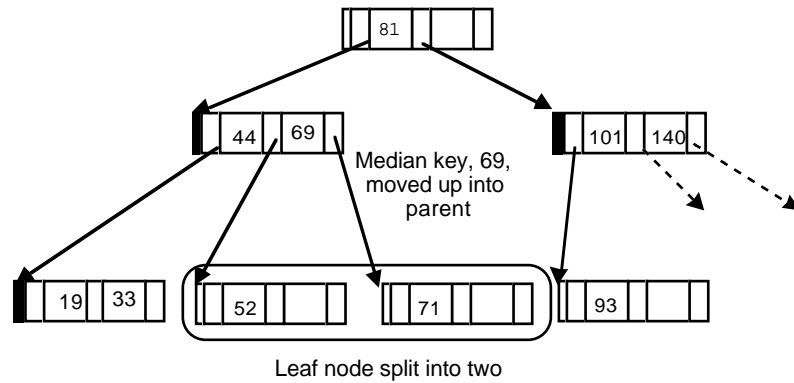


Figure 24.8 Splitting a full leaf node to accommodate another inserted key.

First, the key 20 is inserted. This should go into the leaf currently occupied by keys 19 and 33. Since this leaf is full, it has to be split. Key 19 goes in the left part, key 33 in the right part and key 20 (the median) has to be inserted into the parent. But the parent, the node with 44 and 69 is itself full. So, it too must be split. The left part will hold the smallest key (the 20) and have links down to the nodes with 19 and 33. The right part will hold the key 69 and links down to the nodes with keys 52 and 71. The median key, 44, must be pushed into the parent, the root node with the 81. This node has room; it gets changed from a 2-node to a 3-node. The resulting situation is shown in Figure 24.9.

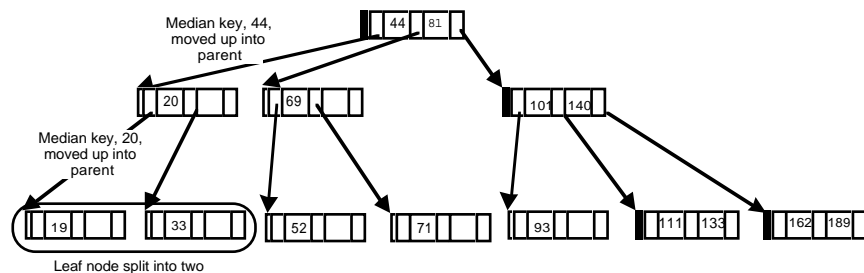


Figure 24.9 Another insertion, another split, and its ramifications.

Insertion of the next key value, 161, causes more problems. It should go in the leaf node where the values 162 and 189 are currently located. As this leaf is full, it must be split. The new key 161 can go in the left part; the large key 189 can go in the new right node; and the median key, 162, (and the link to the new right node) get passed back to be inserted into the parent node. But this node, the one with the 101 and 140 keys is also full. So it gets split. One part gets to hold the key 101 and links down to the node with 93 and the node with 111 and 133. The new part gets to hold the key 162 and its

links down to the node with 161 and the node with 189. The median value, 140, has to go in the parent node. But this is full. So, once again a split occurs. One node takes the 44, another takes the 140 and the median value, 81, has to be pushed up to the parent level.

There isn't a parent. The node with keys 44 and 81 used to be the root node of the tree. So, it is time to "grow a new root". The new root holds the 81 key. The result is as shown in Figure 24.10.

Growing a new root

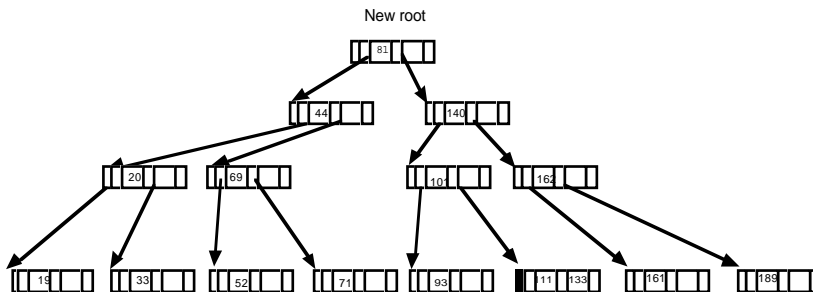


Figure 24.10 The tree grows a new root.

Trees in computer programs are always strange. Their branch points have "children" and their leaves have "parents". They grow downwards, so "up" means closer to the root not nearer to the leaves. These multiway trees add another aberrant behaviour; they grow at the root rather than at the ends of existing branches. It is done this way to keep those paths from root to leaf the same for all leaves; this is required as its part of the mechanism that guarantees that searches, insertions, (and deletions) have a cost that is proportional to $O(\lg(N))$.

BTrees

A BTree is simply a 2-3 tree on steroids. Its nodes don't have two keys and three links; instead its going to be something like 256 keys and 257 links. A fully populated BTree (one where all the nodes held the maximum possible number of keys) could hold 256 keys in a one level tree, around 60000 keys in a two level tree, sixteen million keys in a three level tree. Figure 24.11 gives an idea as to the form of a node and shape of a BTree. The node now has a count field rather than a flag; the count defines the number of keys in the node.

A BTree can be searched, and data can be inserted into a BTree, using algorithms very similar to those that have just been illustrated for the 2-3 tree. You can implement BTrees that work this way, where all the links are memory pointers, and the real data records are accessed using pointers that are stored in the nodes along with their key values.

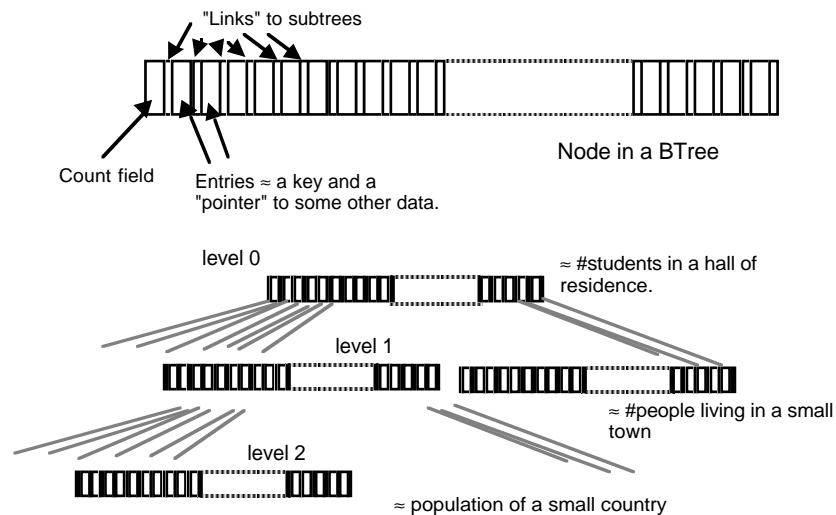


Figure 24.11 Features of a BTree.

But you don't have any really good reasons for using a memory resident BTree like that. If all your data fit in main memory, you might as well use something more standard like an AVL tree.

Exceeding memory limits

However, if you have a really large collection of keyed data records, it is likely that they won't all fit in memory. A large company (e.g. a utility like an electricity company) may have records on two million customers. Each of these customer records is likely to be a thousand bytes or more (name, address, payment records, ...). Now two thousand million bytes of data requires rather more "SIM" chips than fit in the average computer. You can't keep such data in memory instead they must be kept on disk.

This is where the BTree becomes useful. As will be explained more in the next two sections, it allows you to keep both your primary data records, and your search tree structure, out on disk. Only a few nodes from the tree and a single data record ever need be in primary memory. So you can have very large data collections, provided that you have sufficient disk space (and most PCs support disks with up to 4 gigabytes capacity).

24.2.2 A tree on a disk?

A binary file can always be treated as an array of bytes. If you know where a data record is located (i.e. the "array index" of its first byte) you can use a "seek" operation on a file to set the position for the next read or write operation. Then, provided you know the size of the data record, you can use a low level read or write operation to

transfer the necessary number of bytes. These operations have been illustrated previously with the examples in Chapters 18 (the customer records example), 19 and 23 (the different InfoStore examples).

This ability to treat a file as a byte array makes it practical to map something like a tree structure onto a disk file. We can start by considering simple binary trees that hold solely an integer key. A memory version of such a tree would use structures like the following:

```
struct binr {
    long      key;
    binr      *left_p;
    binr      *right_p;
};
```

with address pointers `left_p` and `right_p` holding the locations in memory of the first node in the corresponding subtree. If we want something like that on a disk file, we will need a record like the following:

```
struct dbinr {
    long      key;
    daddr_t   left;
    daddr_t   right;
};
```

The values in the `left` and `right` data members of a `dbinr` structure will be byte locations where a node is located in the disk file. These will be referred to below as "disk addresses", though they are more accurately termed "file offsets". The type `daddr_t` ("disk address type") is an alias for long integer. It is usually defined in one of the standard header files (`stdlib`, `unistd` or `unix`, or `sys_types`, or ...). If you can't locate the right header you can always provide the typedef yourself:

```
typedef long daddr_t;
```

Figure 24.12 illustrates how a binary tree might be represented in a disk file (the numbers used for file offsets assume that the record size is twelve bytes which is what most systems would allocate for a record with three long integer fields).

Storing the tree structure in a disk file

The first record inserted would go at the start of the file (disk address 0); in the example shown this was the record with key 45.. Initially, the first node would have -1s in its `left` and `right` link fields (-1 is not a valid disk address, this value serves the same role as `NULL` in a memory pointer representation of a tree).

The second record added had key 11. Its record gets written at the end of the existing file, so it starts at byte 12 of the file. The `left` link for the first node would be changed to hold the value 12. Similarly, addition of a record with key 92 results in a new node being created on disk (at location 24) and this disk address would then be written into the appropriate link field of the disk record with key 45.

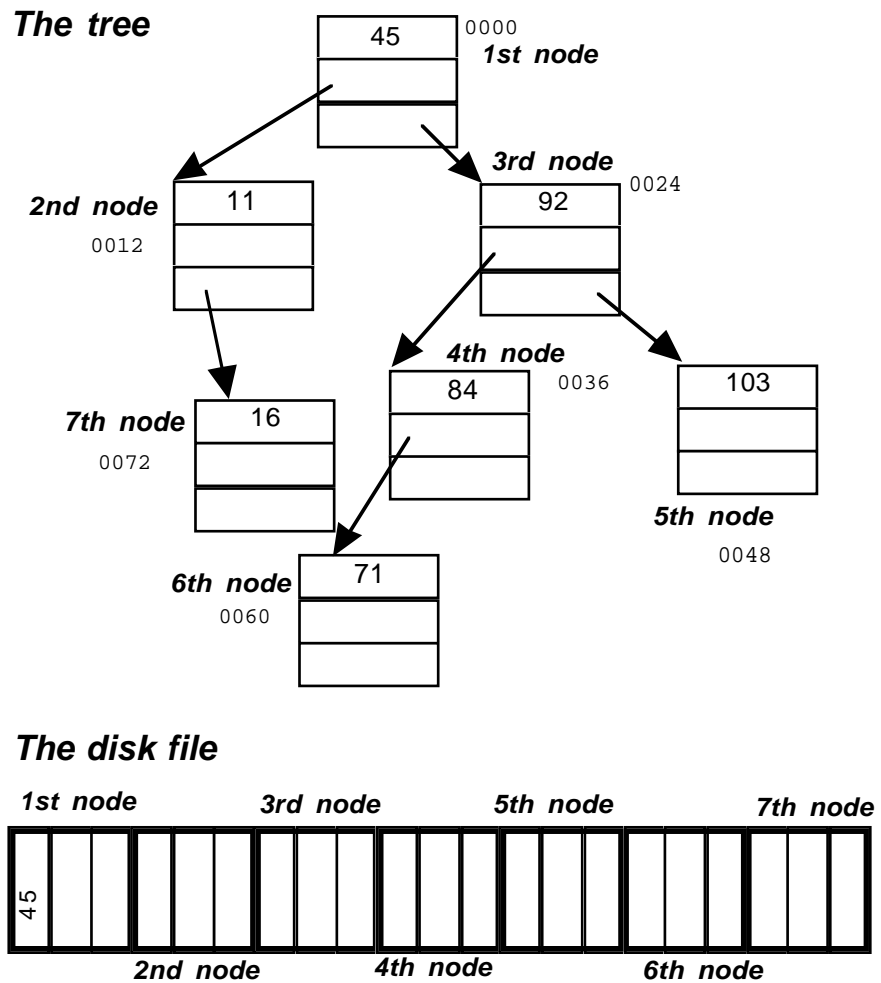


Figure 24.12 Mapping a binary tree onto records in a disk file.

You should have no difficulty in working out how the rest of the file gets built up and the links get set.

Such a tree on disk can be searched to determine whether it contains a record with a given key. The code would be something like the following:

```

fstream    treefile;
int search(long sought)
{
    // Assume that treefile has already been opened successfully
    dbinr arec;

```

```

// "root" node will be at location 0 of file
long diskpos = 0;
for(;;) {
    treefile.seekg(diskpos);
    treefile.read((char*)&arec, sizeof(dbinr));
    if(sought == arec.key) return 1;
    if(sought < arec.key)
        diskpos = arec.left;
    else diskpos = arec.right;
    if(diskpos == -1)
        return 0;
}
}

```

This is just another version of an iterative search on a binary tree (similar to the search function illustrated in Section 24.1 for searching an AVL tree). It is a relatively expensive version; each cycle of the loop involving disk transfer operations.

Normally, you would have data records as well as keys. You would use two files; one file stores the tree structure, the other file would store the data records (a bit like the index file and the articles file in the InfoStore example). If all the data records are the same size, there are no difficulties. The record structure for a tree node would be changed to something like:

Store data in a separate file

```

struct dbinr {
    long      key;
    daddr_t   dataloc;      // extra link to datafile
    daddr_t   left;
    daddr_t   right;
};

```

With the extra field being the location of the data associated with the given key; this would be an offset into the second data file.

The search routine to get the data record associated with a given key would be:

```

fstream      treefile;
fstream      datafile
int search(long sought, datarec& d)
{
    // Assume both files have already been opened successfully
    dbinr arec;
    long diskpos = 0;
    for(;;) {
        treefile.seekg(diskpos);
        treefile.read((char*)&arec, sizeof(dbinr));
        if(sought == arec.key) {
            datafile.seekg(arec.dataloc);
            datafile.read((char*)&d, sizeof(datarec));
            return 1;
        }
    }
}

```

```

        if(sought < arec.key)
            diskpos = arec.left;
        else diskpos = arec.right;
        if(diskpos == -1)
            return 0;
    }
}

```

The records are stored separately from the tree structure because you don't want to read each record as you move from tree node to tree node. You only want to read a data record, which after all might be quite large, when you have found the correct one.

It should be obvious that there are no great technical problems in mapping binary trees (or more elaborate things like AVL trees) onto disk files. But it isn't something that you would really want to do.

The iterative loops in the search, and the recursive call sequences involved in the insertion and deletion operations require many tree nodes to be read from the "tree file". As illustrated in Figure 24.13, the tree nodes are going to be stored in disk blocks that may be scattered across the disk. Each seek and read operation may involve relatively lengthy disk operations (e.g. as much as 0.02 seconds for the operating system to read in the disk block containing the next tree node).

If the trees are deep, then many of the operations will involve reading multiple blocks. After all, a binary tree that has a million keys in it will be twenty levels deep. Consequently each search operation on a disk based binary tree may require as many as twenty disk seeks to find the required tree node (and then one more seek to find the corresponding data).

A BTree with a million keys is going to be only three levels deep if its nodes have ≈ 250 keys. Searching such a tree will involve three seeks to get just three tree nodes, and then the extra seek to find the data. Three disk operations is much better than twenty. BTrees are just as easy to map onto disks as are other trees. But because of their shallow depths, their use doesn't incur so much of a penalty.

The structure for a BTree tree node, and the form of the BTree index file, are illustrated in Figure 24.14. The example node has space for only a few keys where a real BTree has hundreds. Small sized nodes are necessary in order to illustrate algorithms and when testing the implementation. Many of the more complex tree rearrangements occur only when a node becomes full or empty. You would have to insert several million items if you wanted to fill most of the nodes in a three level tree with 250 keys per node; that would make testing difficult. Testing is a lot easier if the nodes have only a few keys.

The node has a count specifying the number of key/location pairs that are filled, an array of these key/location pairs, and an array of links. The links array is one larger than the keys array and a BTree node either has no links (a leaf node) or has one more link than it has keys. The entries in the links array are again disk addresses; they are the addresses of the BTree nodes that represent subtrees hung below the current BTree node. A BTree node has the following data members:

■ Tree node of a few contiguous bytes

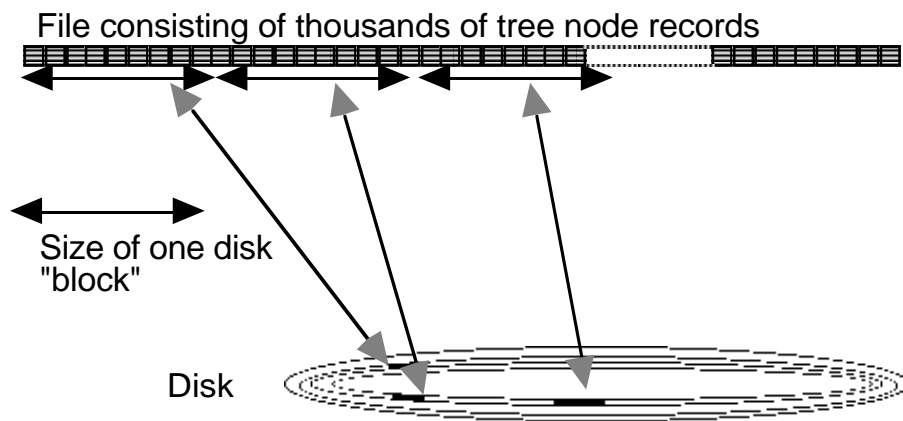
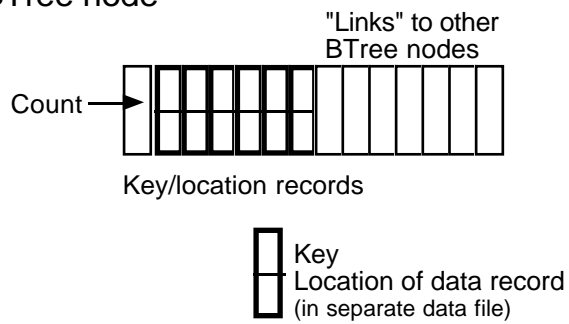


Figure 24.13 Mapping a file onto disk blocks.

BTree node



BTree file

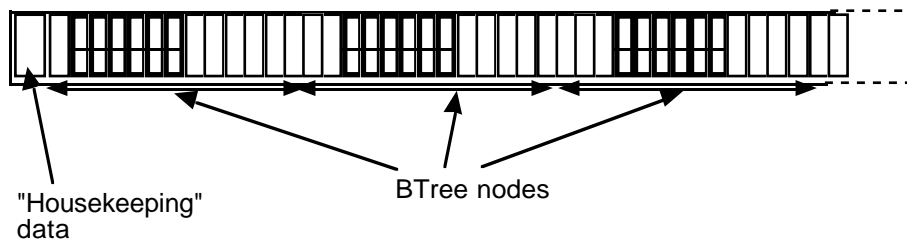


Figure 24.14 BTree nodes and the BTree file.

```

int          n_data;
KLRec        data[MAX]; /* 0..n_data-1 are filled */
daddr_t      links[MAX+1]; /* 0..n_data are filled */

```

where KLRec is defined as "struct KLRec { long fKey; daddr_t fLocation; };". (For simplicity, most of the later diagrams will omit the link to the data record and show solely the values of the keys in those key/location fields. Similarly, the disk addresses of subtrees, that would be in the link fields, are usually not shown.)

The BTree file consists mainly of BTree node records, but there is small amount of "housekeeping information" that has to be kept at the start of the file. In this slightly simplified implementation, the only housekeeping data used is a link (disk address) to the BTree node that represents the current root for the tree, and a count for the number of items.

class BTree "Client programmers" using a BTree will see it as basically something that owns an index file (the one with the BTree nodes) and a data file, and which provides Add(), Find(), and Remove() operations that efficiently transfer data records (instances of classes derived from KeyedStorableItem) to/from the data file. (For convenience, an "add" operation specifying an existing key should be treated as an "update"; the existing data record with the given key is overwritten by the new data.)

```

class BTree
{
public:
    BTree(const char* filename);
    ~BTree();

    int    NumItems(void) const;

    void    Add(KeyedStorableItem& d);
    int     Find(long key, KeyedStorableItem& rec);
    void    Remove(long key);
private:
    ...

    fstream    fTreeFile;
    fstream    fDataFile;
};

```

This implementation is simplified. Data items once written to disk will always occupy space. Deletion of a data item simply removes its key/location entry from the index file. Similarly, BTree nodes that become empty and get discarded also continue to occupy space on disk; once again, they are simply unlinked from the index structure.

A real implementation would employ some extra "housekeeping data" to keep track of deleted records and discarded BTree nodes. If there are "deleted records", the next request for a new record can be satisfied by reusing existing allocated space rather than by extending the data file. The implementation of this recycling scheme involves

keeping two lists, one of deleted data records and the other of discarded BTree nodes. The links of these lists are stored in the "deleted" data records of the corresponding disk files (i.e. a "list on a disk"). The starting points of the two "free lists" are included with the other "housekeeping information" at the start of the index file.

24.2.3 BTree: search, insertion, and deletion operations

All the more elaborate trees have rules that define how their nodes should be organized. You may find minor variations in different text books for the rules relating to BTrees. The rules basically specify:

Rules defining a BTree

- A BTree is a tree with nodes that have the data members previously illustrated:

```
int      n_data;      // number of keys in current node
KLRec    data[MAX];   // key-location pairs for data
daddr_t  links[MAX+1]; // links to subtrees
```

- If a node x is an internal node, it will hold $x.n_data$ keys and $(x.n_data + 1)$ links to subtrees. Every internal node contains one more link than it has keys.
- If a node is a leaf, all its link fields are "NULL". (i.e. the -1 value for "no disk address", NO_DADDR).
- The keys within a node are kept ordered: $x.data.fKey[0] < x.data.fKey[1] < \dots$
- The keys in a node separate the ranges of keys stored in subtrees. So $x.link[0]$ links to the start of a subtree containing records whose keys will all be less than $x.data.fKey[0]$; $x.link[1]$ points to a subtree containing records whose keys (k) are in the range $x.data.fKey[0] < k < x.data.fKey[1]$. The final link, $x.link[x.n_data]$, connects to a subtree with records having keys greater than $x.data.fKey[x.n_data]$.
- Every leaf is at the same depth.
- There are lower and upper bounds on the number of keys in a node. Apart from the root node, every node must contain at least $MAX/2$ keys and at most MAX keys.

The root node may contain fewer than $MAX/2$ keys.
- If an insertion or a remove operation results in a node that violates these conditions, the tree must be reorganized to make all nodes again satisfy these conditions.

Find

Searching the tree for a record with a given key is relatively simple. It involves just a slight generalization of the algorithm suggested earlier for searching a 2-3 tree.

You start by loading the root node of the tree (reading it from the index file into memory). Next you must search in the current node for the key. You stop when you find the key, or when you find a key greater than the value sought. If the key was found, the associated location information identifies where the data record can be found in the data file. The data record can then be loaded and the Find routine can return a success indicator.

If the key is not matched, the search should continue in a subtree (provided that there is a subtree). The search for the key will have stopped with an index set so that it either identifies the position of the matching key or the link that should be used to get to the BTreeNode at the start of the required subtree.

The driver routine is iterative. It keeps searching until either the key is found, or a "null" link (i.e. a -1 disk address) is encountered in a link field.

Some of the work is done the BTree::Find() function itself. But it is worth making a class BTreeNode to look after details of links and counts etc. A BTreeNode, once loaded from disk, can be asked to check itself for the key.

<p><i>Get and check a BTreeNode from disk</i></p> <p><i>If find key, get data from data file</i></p> <p><i>Otherwise, use link with disk address of subtree</i></p>	<pre> BTree::Find(long key, KeyedStorableItem& rec) initialize disk-address to hold address of root node (from "housekeeping information" in index file) while(disk-address is valid) { load a BTreeNode, current, from the specified disk-address ask current node to search itself for "key" if (key was found) get associated data location, loc GetDataRecord(rec, loc); return success otherwise use identified link disk-address = current.links[index]; } report key not present </pre>
---	---

*Searching inside a
node*

The BTreeNode object would have to find the required key, returning a success or failure indicator. It would also have to set an index value identifying the position of the key (or of the link down to the subtree where the required key might be located). Since the keys in a node are ordered, you should use binary search. For simplicity, a linear search is shown in the following implementation. The loop checks successive keys, incrementing index each time, until either the key is found or all keys have been checked. (You should work through the code and convince yourself that, if the key is not present, the final value of index will identify the link to the correct subtree).


```

int BTreeNode::SearchInNode (long keysought, int& index)
{
    for (index = 0; index < n_data; index++)
        if (data[index].fKey == keysought)
            return 1;
        else if (data[index].fKey > keysought)
            return 0;
    return 0;
}

```

Add

The algorithm is essentially the same as that illustrated for the 2-3 tree:

```

recursively...

    chase down through links until find position where
    record should go

    if find a record with same key, replace old data record
    else "insert" new record into node
        if record fits, return success
        else
            split the node
                have MAX+1 records (MAX already
                in full node, and the extra
                record being inserted)
                leave MAX/2 with lowest keys in
                existing node
                put MAX/2 with highest keys in
                new node
                return the median (middle value)
    as unwind recursion:
        check if given a median record to insert, if get
        one then insert (with again possible split...)

```

This is another case where a substantial number of auxiliary functions are needed to handle the various different aspects of the work. The implementation given in the next section uses the following functions for class BTree:

```

BTree::Add(KeyedStorableItem& d);           // Interface

BTree::DoAdd(...);                          // Main recursion

BTree::Split(...);                          // Splitting nodes
BTree::SplitInsertLeft(...);                // auxiliary splitting
BTree::SplitInsertMiddle(...);              // functions
BTree::SplitInsertRight(...)

```

as well as functions in the auxiliary BTreeNode class:

```
BTreeNode::SearchInNode(...)           // Check for key
BTreeNode::InsertInNode(...)           // Simple insertion
BTreeNode::NotFull()                   // Check if full
```

Add() The Add() function is the client interface. It sets up the initial call to the main DoAdd() recursive function. It also has to deal with the special case of growing a new root for the tree (as illustrated for the 2-3 tree in Figure 24.10):

```
Add
    invoke DoAdd()
        passing it as arguments the new data record, and
        the disk address of the current root of the tree

    test "work flag" returned by DoAdd(),
    if work flag is set create new root as follows
        BTreeNode new_root;
        fill in number of keys as 1,
        insert (median) KLRec returned by DoAdd()
        insert two links, link[0] to link to current root
            link[1] to link to disk address that
                DoAdd() reports for newly created node
        Save the new root node to the index file
        Update the "root" info. in the housekeeping part of
            the index file
```

Recursive DoAdd() function The recursive function DoAdd() is the most complex. It has three aspects. There is an inward recursion aspect; this chases down subtree links through the tree. When the recursion process is complete and the correct point for the record has been found in the tree, the data get saved to disk. The final aspect is organizing the "fix up" operations as recursion is unwind.

Several BTreeNode on the stack Each recursive call to DoAdd() will load another BTreeNode into the stack. BTreeNodes are going to be a few hundred to a few thousand bytes in size. Since the maximum limit of the recursion is defined by the depth of the tree (which won't be large), this stacking up of the BTreeNodes will not use excessive memory. When the insertion point is found, the BTreeNodes in the stack are those that define the path back to the root. These are the nodes that may need to be "fixed up" if a node was full and had to be split.

Inward recursion to find place for record Inwards recursion aims to get to the point in the tree where the new data should go. Since there are now many subtrees below each tree node, one aspect of the inward recursion process is a search through the current node to find the appropriate link to follow for a given key value.

Terminating recursion, by replacing a data record The inwards recursive phase terminates on either of two conditions. There is the special case of finding an existing record with the key. In this case, the data are

replaced in the data file and a flag is set to indicate that no work is necessary as the recursion unwinds.

The other terminating condition is that the recursive call has been made with a "null" disk address passed as an argument. This means that recursion has reached the bottom of the tree. The new data record should be added to the data file. Its disk address and its key get placed in a `KLRec`. This is returned to the preceding level of recursion for processing.

Terminating recursion by inserting a new record

The final aspect of `DoAdd()` is the mechanism for unwinding recursion. This starts by checking a "work flag" returned by the recursive call. If the flag is not set, function `DoAdd()` can simply return; but if the flag is set then a `KLRec` and link value have to be inserted into the `BTreeNode` in the current stack frame and the updated node must be written back to disk.

Unwinding recursion and fixing up records

Most of the remaining complexities relate to insertions into a `BTreeNode`. These operations are outlined after the complete `DoAdd()` algorithm.

Naturally, like all recursive routines, the termination conditions for `DoAdd()` come first. So the actual structure of the function involves the termination tests, then the setting up of a further recursive call, and finally, after the recursive call, the fix-up code. The algorithm for `DoAdd()` is as follows:

```
DoAdd( arguments include record to be inserted and disk address
        of next node from index file, ...)
    Check "disk address" argument passed in call

    if (disk address is NO_DADDR) {
        Save new record to data file
        Update housekeeping info (number of records etc)
        Return details of key for new data record and
            its location in data file, and set flag
            to indicate fix up required
        return;
    }

    Load a BTreeNode from the index file
        into current stack frame

    Ask current node to search for given key
    if(found) {
        Find location in data file for record with
            this key
        Replace with new data
        Set flag to say fix up not required
        return;
    }

    // If key not found, 'index' variable will have been set
    // so as to identify link to subtree
    DoAdd(newData, current.links[index],
        ...);
```

*Termination conditions
Check whether "off end of tree"*

Load another tree node into memory

Check for key

If find key, terminate by replacing data

Recursive call

*Check "work flag"
on return from
recursion*

```
if (fix up flag not set)
    return;
```

Do fix up operations:

```
if (current Btree node is not full)
```

Simple insertion

```
    Insert details of key/disk location into
    current node
```

Or splitting of node

```
else {
    Split the current node,
    Get some key/location records transferred
    to a new tree node and add this to
    index file
    Get a "median record" to be passed by for
    insertion by caller
    Set flag to indicate caller must do fix up.
}
```

```
Have changed current BTreeNode by adding to it, or
by splitting it, so write it to disk
```

*Insertion into a
partially filled
BTreeNode*

Insertion of an extra key into a partially filled BTreeNode is the simplest case, see Figure 24.15. The BTreeNode can be given details of the new key (more strictly, a KLRec, key/location pair, defining the data item), and the position where this is to go in the node's array of KLRecs. Existing entries with keys that are greater in value should be moved to the right in the array to make room; the new KLRec entry can be inserted and the count of entries incremented. Every time a BTreeNode is changed, it has to be written back to the index file.

Simple insert into a "leaf node":



Figure 24.15 Simple insertion into a partially filled node.

If the insertion is into an "internal" BTreeNode, then there will be an extra link that has to be inserted in addition to the KLRec. This extra link is the disk address of an extra BTreeNode that results from a "split" operation at a lower level..

The BTreeNode can have a single member function that deals with insertions of a KLRec and associated extra link (the extra link argument will be -1 in the case of insertion into a leaf node):

```
void BTreeNode::InsertInNode(KLRec& info, daddr_t diskpos,
                             int index)
{
    for (int i = n_data - 1; i >= index; i--) {
        data[i+1] = data[i];
        links[i+2] = links[i+1];
    }
    data[index] = info;
    links[index+1] = diskpos;
    n_data++;
}
```

*Move existing entries
right to make room*

*Insert key/location
pair and link to
subtree*

If a BTreeNode already has MAX keys, then it has to be split, just like a full 2-3 node would get split. There will be a total of MAX+1 keys (the MAX keys already in the node and the extra one). Half (those with the lowest values) are left in the existing node; half (those with the greatest values) are copied into a newly created BTreeNode, and one, the median value, gets moved up into the parent BTreeNode.

*Insertion into a full
BTreeNode – splitting
the node*

The basic principles are as illustrated in Figure 24.16. This shows an insertion into a full leaf node (all its subtree links are -1). An extra BTreeNode is created (shown as "node02"). Half the data are copied across. Both nodes are marked as half empty. Then, both would be written to disk (the new BTreeNode, "node02", going at the end of the index file, the original BTreeNode, "node01", being overwritten on disk with the updated information). The median key, and the disk address for the new "node02" would then have to be inserted into the BTreeNode that is the parent of node01 (and now of node02 as well).

There are really three slightly different versions of this split process. In the first, the extra key has a low value and it gets inserted into the left (original) node. In the second, the new key is the median value; it gets moved to the parent. Finally, there is the situation where the new key is large and it belongs in the right (new) node. The reshuffling processes that move data around are slightly different for the three cases and so are best handled by separate auxiliary functions.

The overall BTree::Split() function has to be organized along the following lines:

```
Split
given: a BTreeNode to be split "node_to_split"
      an extra KLRec key/data-location pair
      an extra link (disk address of a BTreeNode that
      is to become a subtree of "node_to_split")
```

Splitting a full node:

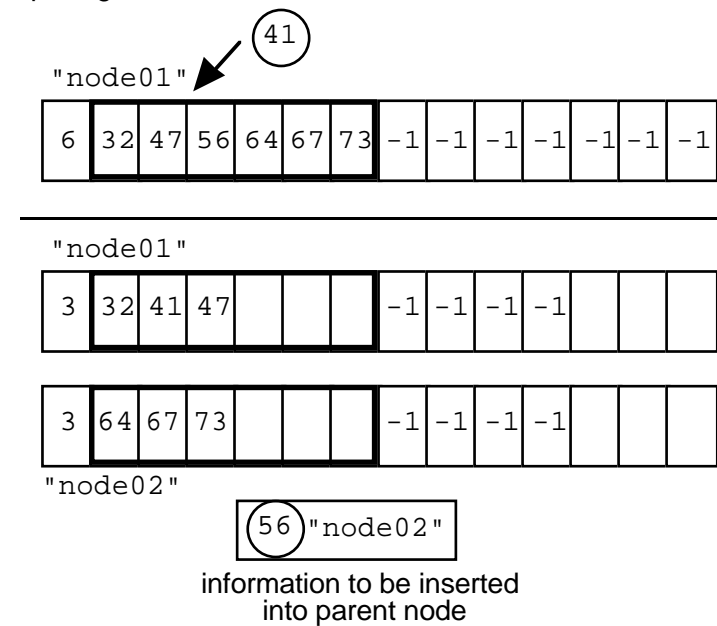


Figure 24.16 Splitting a BTreeNode.

```

        and an index specifying where new entry to go.
    if (index > MIN)
        use an auxiliary "Insert Right" function to
            get new key into right node
    else
    if (index == MIN)
        use an auxiliary "Insert Middle" function to
            split the node, sharing the existing entries
            between old and new parts
            keeping the new key as "median"
    else
        use an auxiliary "Insert Left" function to
            get new key into left node
    return
        Median value to get put into parent node
        disk address of the newly created node.

```

The auxiliary functions have loops that shuffle KLRec records and links between the old and new BTreeNode records. Though the code is fairly simple in structure, there are lots of niggling little details relating to which link values end up in the link arrays of the two nodes.

The basic operations are as shown in Figure 24.17 for the case of inserting the new data in the right hand node; the process is:

Insert in Right

```

given: a BTreeNode to be split "node_to_split"
      an extra KLRec key/data-location pair
      an extra link (disk address of a BTreeNode that
        is to become a subtree of "node_to_split"
      an index specifying where new entry to go

BTreeNode newNode; // BTreeNode created on stack
for (i = MAX-1, j = MIN-1; i >= index; i--, j--) {
    newNode.links[j+1] = nodetosplit.links[i+1];
    newNode.data[j] = nodetosplit.data[i];
}
newNode.links[j+1] = extralink;
newNode.data[j] = extradata;
for (j--; i > MIN; i--, j--) {
    newNode.links[j+1] = nodetosplit.links[i+1];
    newNode.data[j] = nodetosplit.data[i];
}
newNode.links[0] = nodetosplit.links[MIN+1];
newNode.n_data = nodetosplit.n_data = MIN;
return_diskpos = MakeNewDiskBNode(newNode);

return
    Median value to get put into parent node
    (nodetosplit.data[MIN])
    disk address of the newly created node.
    (return_diskpos)

```

Copy1

Insertion

Copy2

Clean up

The operations all take place on a temporary BTreeNode created in the stack (as a local variable of the "insert in right" function). When this has been filled in successfully, an auxiliary function gets it into the data file (at the end of the file) and returns its disk address for future reference.

The KLRec with the median valued key, and the address of the extra BTreeNode, are passed back to the calling level of the recursion. There they have to be inserted into the parent, which may again split. The process is identical in concept to that shown in Figures 24.8, 24.9 and 24.10 for the 2-3 trees.

As also illustrated previously for the 2-3 trees, if there is no parent node, a new root node has to be created for the tree. This process is illustrated in Figure 24.18.

The figure shows a BTree index file that initially has a single full node containing the keys 20, 33, 45, 56, 67, and 79 (links to data records in the datafile are not shown). There are no subtrees; so all the BTree structure link fields are -1; and the count field is 6. The BTreeNode is assumed to be 80 bytes in size; starting at byte 4 (after a minimal housekeeping record that contains solely the byte address of the first record).

Insertion of key 37 will force the record to split as there would now be seven keys and these nodes have a maximum of six. The three highest keys (56, 67, and 79) are

**Initially a single full
BTreeNode**

shifted into a new BTreeNode (see Figure 24.18). This would start at byte 84 of the disk file. The node would be assembled in a structure on the stack and then be written to the disk file.

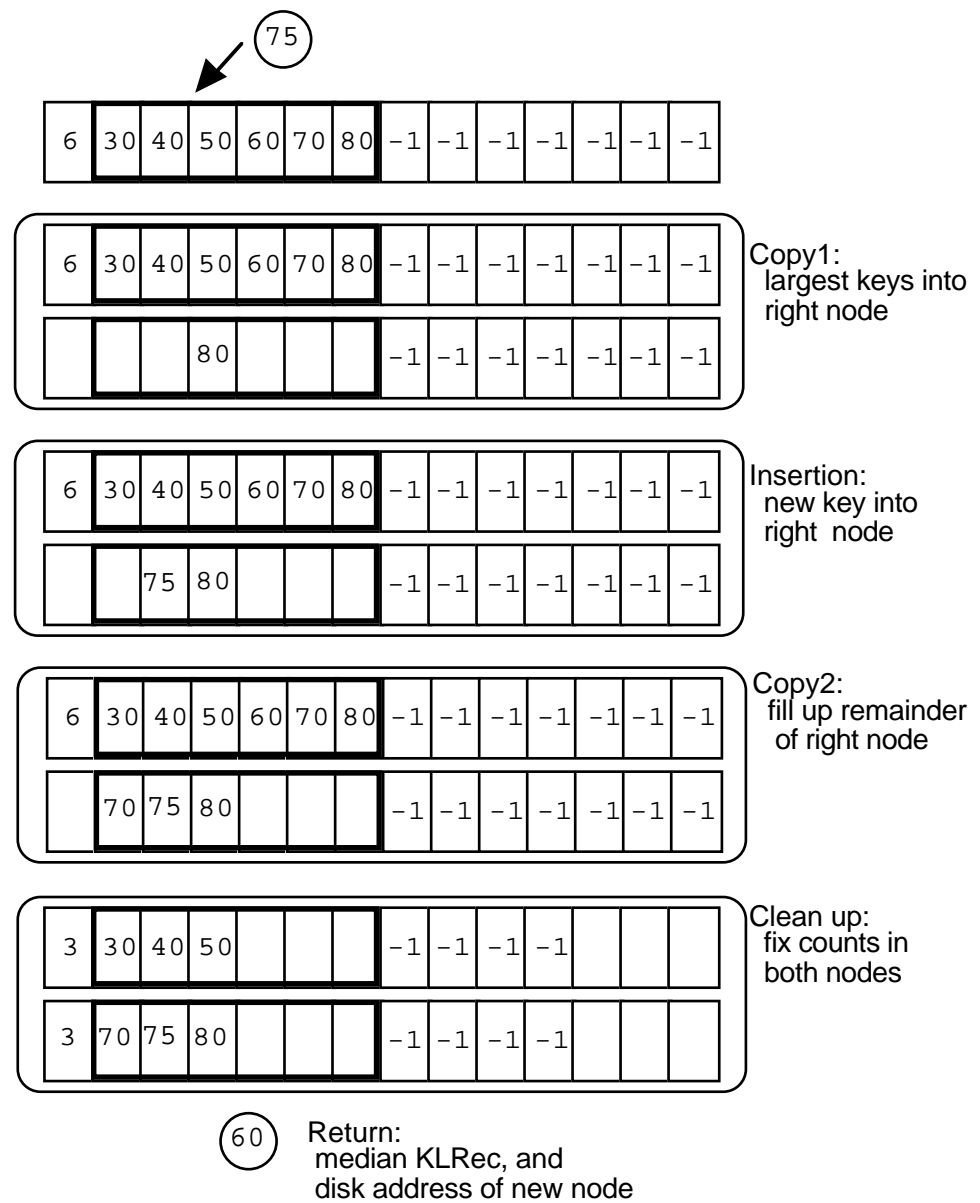
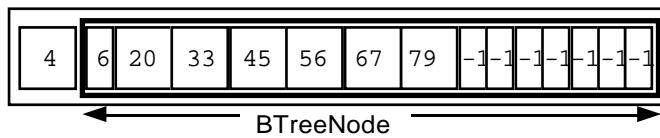
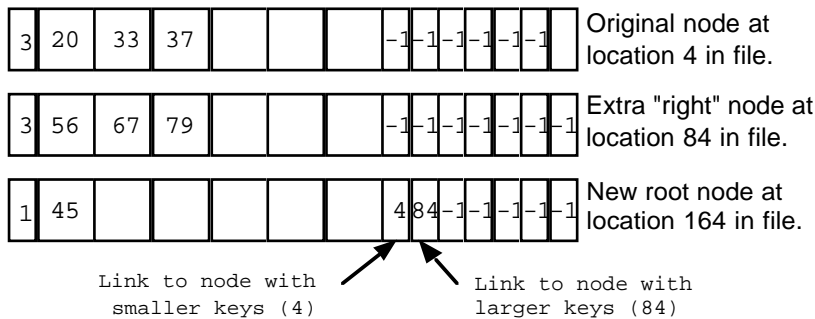


Figure 24.17 Reorganizing a pair of BTreeNodes after a "split".

Index file with housekeeping data and one BTreeNode:



Insertion of 37, splits node:



Final file contains three BTreeNodes:

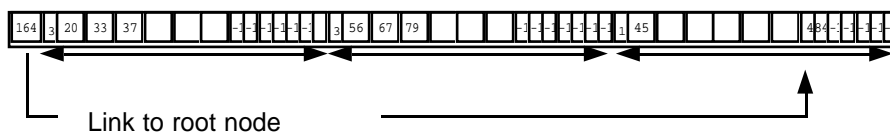


Figure 24.18 Splitting leading to a new root node.

The three lowest keys (20, 33, 37) would go in the original BTreeNode starting at byte 4 of the file. This BTreeNode would first be composed in memory and then would overwrite the existing record on disk.

The median, with key 45, would have to go into a new root BTreeNode. It would need links down to the original BTreeNode at 4 (link[0], all the keys less than 45) and the BTreeNode just created at location 84 in the file (link[1], all the keys greater than 45). The new root BTreeNode would get written to the file starting at byte 164.

Finally, the housekeeping data at the front of the file would be updated to hold the disk address of the new root node.

Remove

As already noted, records are removed in much the same way as in AVL trees. A recursive routine hunts down through the tree to find the key for the record that is to be removed. (If the key is not found, the routine returns some failure indication.) As it

recurses down through the tree, the routine loads `BTreeNode`s from the file into the stack, as in previous examples these define the path back to the root and are the nodes that may need to be fixed up.

If the key is found in an internal node, data must be promoted from a leaf node (the implementation in the next section promotes the successor key – the smallest key greater than the key to be deleted). Once the promotion has been done, the original promoted data must be deleted. So the routine further recurses down through the tree until it has the leaf node from where data were taken.

All actual deletions take place on leaf nodes (either because the key to be deleted was itself in a leaf node, or because a key was taken from a leaf node to replace a key in an internal node). A deletion reduces the number of keys in the node. The remaining keys are rearranged to close up the space left by the key that was removed. If there are still at least $\text{MAX}/2$ keys in the leaf, then essentially everything is finished. The node can be written back to the file. Recursion can simply unwind (if an internal node was modified by having data replaced with promoted data, then it gets written to the file during the unwinding process).

***Deficient nodes need
"fixing up"***

The difficulties arise when a node gets left with less than $\text{MAX}/2$ keys. Such a node violates the BTree conditions (unless it happens to be the root node); it is termed a "deficient node". A deficient node can't do anything to "fix itself up". All it can do is report to its parent node. This is achieved, in the recursive procedure, by a node that detects deficiency setting a return flag; the flag is checked at the next level above as the recursion unwinds.

If a parent node sees that a child node has become deficient, it can "fix up" that child node by shifting data from "sibling nodes". There are a couple of different situations that must be handled. These are illustrated in Figures 24.19 and 24.20 .

***Moving data from a
sibling node***

Figure 24.19 illustrates a "move" operation. The initial tree (shown in Pane 1 of Figure 24.19) would have five nodes (only four are shown). The root has three keys (300, 400, and 500) and four links down to subtrees. The first subtree (not shown) would contain the keys less than 300. The second subtree, node n1, contains keys greater than 300 and less than 400. The third subtree (in `link[2]`) has the keys between 400 and 500. The final subtree has the keys greater than 500.

Node n2 has exactly $\text{MAX}/2$ keys. If one of its keys is removed, e.g. 440, it is left "deficient" (Pane 2 of Figure 24.19). It cannot do anything to fix itself up. But, it can report to its problem to its parent node (which in this case is the root node).

The root node can examine the sibling nodes (n1 and n3) on either side of the node that has just become deficient. Node n1 has four keys ($> \text{MAX}/2$). The deficiency in node n2 could be made up by "transferring a key" from n1. That would in this case leave both nodes n1 and n2 with exactly $\text{MAX}/2$ keys.

But of course, you can't simply transfer a key across between nodes because the keys also have to be in order. There has to be a key in the root node such that it is greater than all keys in its left subtree and smaller than all keys in its right subtree.

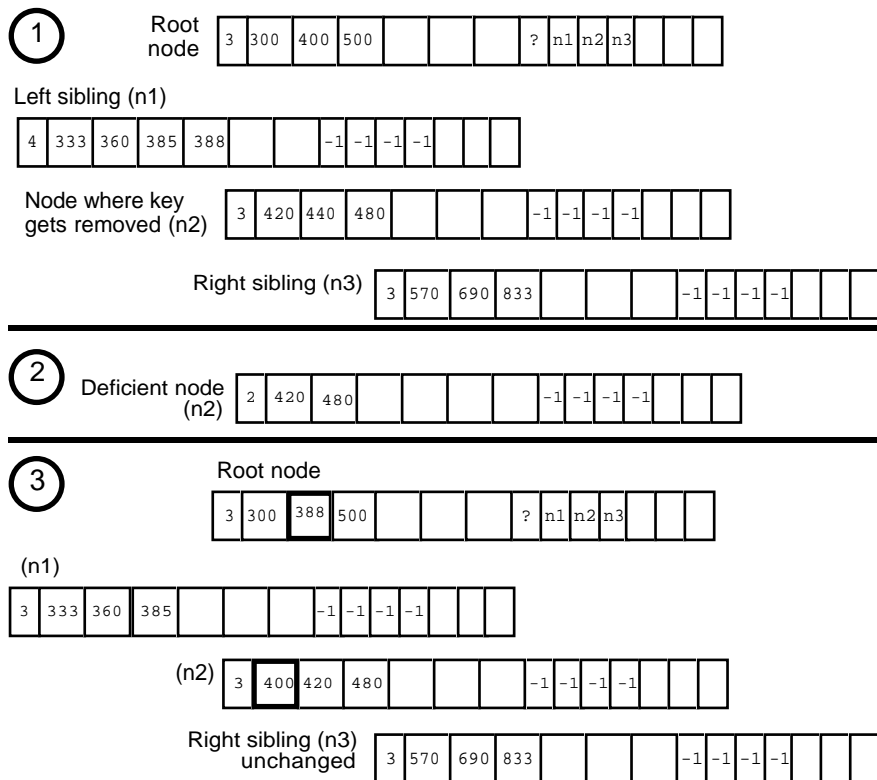


Figure 24.19 Moving data from a sibling to restore BTree property of a "deficient" BTreeNode.

The "transfer of a key" actually involves taking the largest key in a left sibling (or smallest key in a right sibling) and using this to replace a key in the parent. The key from the parent is then used to restore the deficient node to having at least the minimum number of keys.

In the example shown (Pane 3 of Figure 24.19), the key 388 is taken from node n1 (leaving it with three keys) and moved up into the parent (root) node where it replaces the key 400. Key 400 is moved down into node n2.

Everything has been restored. The link down "between" keys 300 and 388 (`link[1]` of the root node) leads to all keys in this range (i.e. to node n1 with keys 333, 360 and 385). The link down between keys 388 and 500 leads to the "subtree" (i.e. node n2) with keys between 388 and 500 (keys 400, 420, and 480). All nodes continue to satisfy the BTree requirements on their minimum number of keys.

For a "move" or transfer to take place, at least one of the siblings of a deficient node must have more than $\text{MAX}/2$ keys. Move operations can take a key from the left sibling or the right sibling of a deficient node. Of course, if the deficient node is on `link[0]`

of its parent then it has no left sibling and a move can only occur from a right sibling. If the deficient node is in the last subtree link of its parent, only a move from a left sibling is possible. If a node has both left and right sibling, and both siblings have more than $\text{MAX}/2$ keys, then either can be used. In such cases, it is best to move a key from the sibling that has the most keys.

Combine operations

Sometimes, both siblings have just the minimum $\text{MAX}/2$ keys. In such situations, "move" operations cannot be used. Instead, there is another way of "fixing up" the node. This alternative way combines all the existing keys in the deficient node and one of its siblings into a single node and ceases to use one of the `BTreeNode`s in the file. Figure 24.20 illustrates a combine operation.

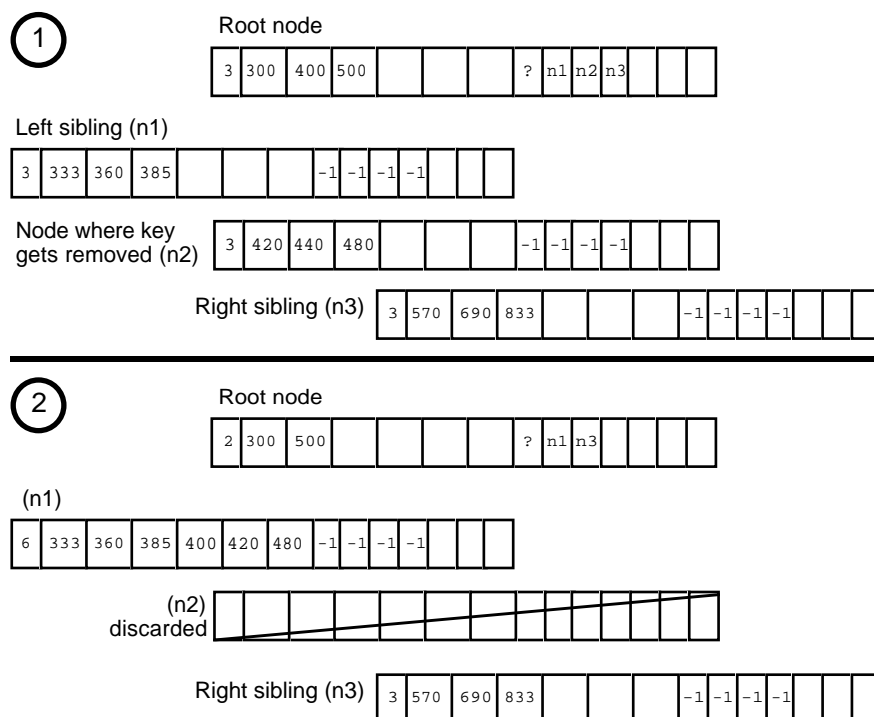


Figure 24.20 Combining `BTreeNode`s after removing a key.

Since a `BTreeNode` has to be removed, there will be one fewer link down from the parent. Since all the links in a `BTreeNode` must either be `NULL` or links down to subtrees, this means that the keys in the parent node have to be squeezed up a bit. There will be exactly $\text{MAX}/2$ keys from a sibling, $\text{MAX}/2 - 1$ keys from the node that became deficient. These are combined along with one key from the parent to produce a full `BTreeNode`.

In the example shown in Figure 24.20, the initial state has each of the three nodes n_1 , n_2 , and n_3 with three keys. Removal of key 440 from n_2 leaves it deficient. The parent can not shift a key from either n_1 or n_3 for that would leave the donor deficient. So, instead, key 400 from the parent, and the remaining keys 420 and 480 are shifted into n_1 , filling it up so that it has six keys. (Alternative rearrangements are possible; for example, key 500 from the parent and the three keys from node n_3 could be shifted into n_2 to fill it up and leave n_3 empty. It doesn't matter which rearrangement is used.)

The `BTreeNode` that becomes empty, n_2 in Figure 24.20, is "discarded". It is no longer linked into the tree structure. (In a simple implementation, it becomes "dead space" in the file; it continues to occupy part of the disk even though it isn't again used.)

As shown in Figure 24.20, the parent node now only has three links down. One goes to a node (not shown) with keys less than 300. The second is to the full node n_2 with all the keys between 300 and 500. The third link is to the node, n_3 , with the keys greater than 500. This leaves the parent node with just two keys.

Since the parent provides one key, it may become "deficient". If the parent becomes deficient, it has to report this fact to its parent during the unwinding of the recursion. A move or combine operation would then be necessary at that level. Removal of a key from a leaf can in some circumstances cause changes at every node on the path back to the root.

The root node is allowed to have fewer than $\text{MAX}/2$ keys. If the root node is involved in a combine operation, it ends up with fewer keys. Of course, it is possible to end up with no keys in the current root! Figure 24.21 illustrates such an occurrence.

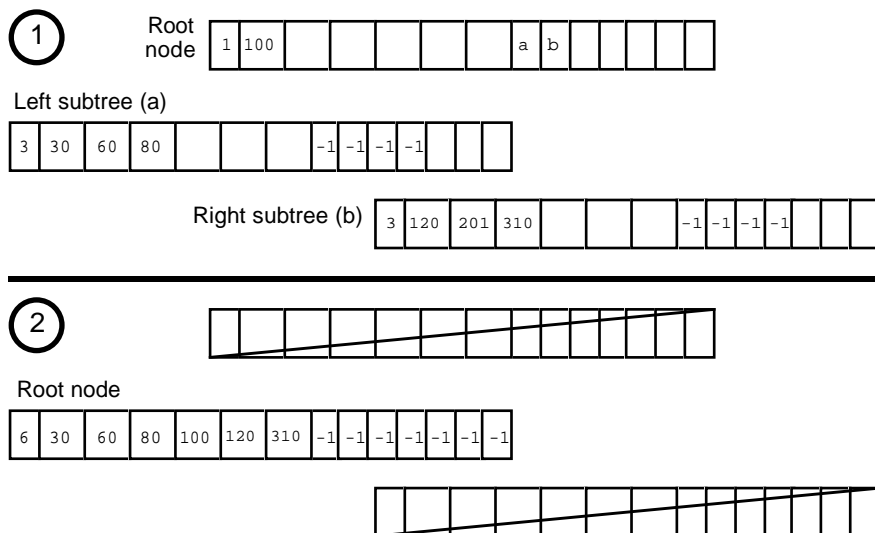


Figure 24.21 Removal of a key may lead to a change of root node.

In this case, two `BTreeNode`s get discarded and the tree is "re-rooted" on the node that originally formed the top of the left subtree. The "housekeeping" information at the start of the BTree index file would have to be updated to reflect such a changed.

Overall removal process

Removal of a record thus involves:

- finding the leaf node with the key (if the key is in an internal node, a promotion operation must be done and then the original copy of the promoted data must be found in a leaf node);
- removal of the key from the leaf;
- unwinding the recursive search, performing "fix ups" on any nodes that become deficient;
- nodes get "fixed up" by parents performing move or combine operations affecting the deficient node, a sibling, and the parent node.

Unlike `Remove()` for the AVL tree which returns the address of a data record in memory, `BTree::Remove(...)` simply performs the action. (The data record is in the data file. It doesn't actually get destroyed; the reference to it in the index is removed making it inaccessible. Again, the space it occupies becomes "dead space" on disk).

Obviously, many auxiliary functions have to be used in the implementation of `BTree::Remove()`. The implementation given in the following section has the following functions:

```
BTree::Remove(long key);           // interface
BTree::DoRemove(...);             // main recursive routine
BTree::DeleteKeyInNode(...);       // removal of key from node
BTree::DeleteInLeaf(...);         // special case, leaf node
BTree::Successor(...);            // get data to replace key in
                                   // internal node
BTree::Restore(...);              // Organize fix up of deficient
                                   // child node
BTree::MergeOrCombineRight(...);   // Merge child and right sibling
BTree::MergeOrCombineLeft(...);    // Merge child and left sibling
BTree::MoveRight(...);            // Move key out of left node
BTree::MoveLeft(...);             // Move key out of right node
BTree::Combine(...);              // Combined deficient node and
                                   // sibling
```

In addition, the implementation relies on several functions of class `BTreeNode`:

```
BTreeNode::SearchInNode(...);      // Finding key
BTreeNode::InsertAtLeft(...);      // Shifting keys into node
BTreeNode::InsertAtRight(...);
BTreeNode::ShiftLeft(...);         // Moving keys around
BTreeNode::Compress(...);
BTreeNode::Deficient(...);         // Checking status of node
BTreeNode::MoreThanMinFilled(...);
```

Function `Remove()` provides the interface; its only argument is the key corresponding to the data record that must be deleted. The function has to check that a tree exists, someone might try to delete data before any have been entered. If there is a BTree to search, the `Remove()` function should load in the root node and set up the call to the main recursive `DoRemove()` function. When `DoRemove()` returns, a check should be made for the special case of needing a new root node (the situation illustrated in Figure 24.21).

Remove() driver function

```
Remove(key)
    if (there is no tree!)
        return;

    Load current root_node

    DoRemove(key, root_node);
    if (root_node has no keys left)
        set housekeeping data to record new root
    else
        save root_node back on disk
```

Function `DoRemove()` is a recursive function with some parallels to the `DoAdd()` function already considered. It has to recursively chase down through the tree to find the key. There has to be a check to stop recursion. As recursion unwinds, any necessary fix up operations are performed.

Main recursive DoRemove()

At each recursive level, the `BTreeNode` to be worked on has already been loaded at the previous level (e.g. `Remove()` loads the root node). Since the `BTreeNode` is already on the stack, it can be checked for the key; if the key is present the auxiliary `DeleteKeyInNode()` function is called to remove it (this will involve a further recursive call to `DoRemove()` if the node is an internal node). When the deletion has been done, function `DoRemove()` can return. Function `DoRemove()` returns a flag indicating whether it has left a node deficient.

Termination of recursion and handling of delete

If the key was not found, the function has to set up a further recursive call using the appropriate link down to a subtree. If it encounters a "null" link, this means that the specified key was not present, in that case the function can simply return. Usually, there will be a valid disk address in the link. The `BTreeNode` at this location in the index file should be loaded onto the stack and the recursive call gets made.

Setting up a recursive call

The unwinding process uses the auxiliary function `Restore()` to fix up any deficient nodes. Other nodes that may have been modified just get written back to the index file.

Unwinding recursion

```
DoRemove(long bad_key and reference to a BTreeNode on the
stack)
    Get node to search for the key
    if (found)
        Delete bad_key from this node
        update count of keys in housekeeping records
```

Termination of recursion

	return indication of whether node was left deficient
Setting up recursive call	Get link to subtree if(subtree == NO_DADDR) return 0; Load next BTreeNode onto stack
Recursive call	repairsneeded = DoRemove (bad_key, nextNode);
Unwinding recursion and fixing up	if (repairsneeded) Restore(...); else SaveBTreeNode(nextNode, subtree); return indication of whether node was left deficient
Deletion of a key	The function DeleteKeyInNode() sorts out how to delete a key. The auxiliary function DeleteInLeaf() deals with the easy case of deletion in a leaf (leaves are easy to recognize, all subtree links are "null"). (Deletion in a leaf is trivial; all higher valued keys are moved one place left so overwriting the key that has to be removed. The count field for the node is then decremented.) If the node is an internal node, the auxiliary function Successor() is employed to get the key/location pair of next higher key (the lowest valued key in the right subtree). Then, DoRemove() must be called recursively to get rid of the original copy of the promoted key/location pair. DeleteKeyInNode(node to work on and index of key to be deleted if (subtree links are null) DeleteInLeaf(aNode, index); return;
Promotion of successor	Replace entry with data promoted from Successor(right subtree); Load node at top of right subtree Use DoRemove to remove promoted data from right subtree Fixup
Restore() function	The Restore() function has to determine whether the deficient node is the leftmost child (in which case can only merge with right sibling), or the rightmost child (can only merge with a left sibling), or an intermediate case with both left and right siblings. If both siblings exist, the merge operation should use the sibling with more keys. The checks involve loading the sibling nodes into memory. Restore(parent node, deficient node, index of parent's link down to deficient node) if(index == 0)


```

        MergeOrCombineRight(...);
    else
        if(index == parent.n_data)
            MergeOrCombineLeft(...);
        else
            Load left sibling into a temporary BTreeNode

            int left_num = temp.n_data;

            Load right sibling into a temporary BTreeNode

            int right_num = temp.n_data;

            if(left_num >= right_num)
                MergeOrCombineLeft(...);
            else
                MergeOrCombineRight(...);

```

The "MergeOrCombine Left / Right" functions check the occupancy of the selected sibling node. If it has sufficient keys, a "move" is done; otherwise the more complex combine operation is performed. The `MoveLeft()` function is similar in organization. **MergeOrCombine**

The `MoveRight()` operation takes a `KLRec` (key/data location pair) from the parent and the rightmost link down from the left node and inserts these into the deficient node (the `BTreeNode` does the actual insertion, it moves all existing entries one place right then inserts the extra data). Then the rightmost `KLRec` is moved from the left node up into the parent. **MoveRight()**

Figures like 24.19 were simplified in that the nodes operated on were leaf nodes. Often they will be internal nodes with links down to lower levels. Thus, to the right of key 388 there would be a link down to a subtree with all keys greater than 388 and less than 400. This link down would have to be moved into `link[0]` of the deficient node.

```

MoveRight(parent node, two siblings, and index position)
    Get KLRec from parent at indexed position
    Get last down link from left node
    Get other BTreeNode to insert KLRec and link
    Replace parent KLRec with rightmost data from left node
    decrement count field in left node

```

The `Combine()` shifts a `KLRec` from the parent and remaining data from the other node.

24.2.4 An implementation

The header file, `BTree.h`, would contain the declaration for the pure abstract class `KeyedStorableItem` along with the main class `BTree`:

```

class KeyedStorableItem {
public:
    virtual      ~KeyedStorableItem() { }
    virtual long  Key(void) const = 0;
    virtual void  PrintOn(ostream& out) const { }
    virtual long  DiskSize(void) const = 0;
    virtual void  ReadFrom(fstream& in) = 0;
    virtual void  WriteTo(fstream& out) const = 0;
};

inline ostream& operator<<(ostream& out,
                           const KeyedStorableItem& d)
{ d.PrintOn(out); return out; }
inline ostream& operator<<(ostream& out,
                           const KeyedStorableItem* p_d)
{ p_d->PrintOn(out); return out; }

```

A `KeyedStorableItem` is essentially something that can report its key and transfer itself to/from a disk file.

The `BTree` code is parameterized according to the number of keys in each node. This number needs to be small during testing but should be enlarged for a production version of the code.

Class `BTree` makes use of `BTreeNode` objects and `KLRec` objects and its functions have pointers and references of these types. They are basically a detail of the implementation so their definitions go in the ".cp" implementation file. The types however must be declared in the header:

```

#define      MIN      3
#define      MAX      (2 * MIN)

class      BTreeNode;
struct     KLRec;

```

Class `BTree` has a simple public interface. The constructor takes a string that will be the "base name" for the index and data files (e.g. if the given name is "test", the files used will be "test.ndx" and "test.dat").

**Public interface of
class *BTree***

```

class BTree
{
public:
    BTree(const char* filename);
    ~BTree();

    int    NumItems(void) const;

    void    Add(KeyedStorableItem& d);
    int     Find(long key, KeyedStorableItem& rec);
    void    Remove(long key);

```

All the complexity exists in the private implementation part.

The implementation needs some structure to represent the "housekeeping data". In this simple implementation, this consists of the root address and a count of items in the collection. The declarations of the various auxiliary functions come after the declaration of this housekeeping structure.

The first group of member functions deal with disk transfers. The BTree object can look after reading and writing its housekeeping information and its BTreeNode's. (The nodes could have been made responsible for reading and writing themselves, it doesn't make much difference). KeyedStorableItem objects are responsible for their own data transfers, but class BTree is responsible for the files. It is the BTree object that must perform the seek operations used to position the read/write file pointers before another object is asked to transfer itself. The implementations for most of these functions are simple, just a seek followed by a request to some other object to read or write itself.

*Auxiliary private
member functions for
disk transfers*

The Get and Save functions use existing entries in the files. The MakeNew functions add extra entries at the end of files (an extra BTreeNode or an extra data record as appropriate). In a more sophisticated implementation, the MakeNew functions could be enhanced to reuse "dead space" left where data records or BTreeNode's have been deleted.

```
private:

    struct HK {
        daddr_t      fRoot;
        long         fNumItems;
    };

    /* Disk i/o group */
    void    GetBTreeNode(BTreeNode& bnrec, daddr_t diskpos);
    void    SaveBTreeNode(BTreeNode& bnrec, daddr_t diskpos);
    void    GetDataRecord(KeyedStorableItem& datarec,
                        daddr_t diskpos);
    void    SaveDataRecord(KeyedStorableItem& datarec,
                        daddr_t diskpos);
    daddr_t    MakeNewDiskBNode(BTreeNode& bnode);
    daddr_t    MakeNewDataRecord(KeyedStorableItem& data);

    void      SaveHK(void);
    void      LoadHK(void);
```

The next declarations will be of all the auxiliary functions needed for the Add operation followed by all the auxiliary functions needed for the Remove operation. The main recursive DoAdd() function has a complex argument list because it must pass back data defining any new information that needs to be inserted into a node (the reference arguments like return_diskpos).

**Auxiliary functions
for Add**

```

void    DoAdd(KeyedStorableItem& newData, daddr_t filepos,
           int& return_flag, KLRec& return_KLRec,
           daddr_t& return_diskpos);
void    Split(BTreeNode& nodetosplit,
           KLRec& extradata, daddr_t extralink,
           int index, KLRec& return_KLRec,
           daddr_t& return_diskpos);
void    SplitInsertLeft(BTreeNode& nodetosplit,
           KLRec& extradata, daddr_t extralink,
           int index, KLRec& return_KLRec,
           daddr_t& return_diskpos);
void    SplitInsertMiddle(BTreeNode& nodetosplit,
           KLRec& extradata, daddr_t extralink,
           int index, KLRec& return_KLRec,
           daddr_t& return_diskpos);
void    SplitInsertRight(BTreeNode& nodetosplit,
           KLRec& extradata, daddr_t extralink,
           int index, KLRec& return_KLRec,
           daddr_t& return_diskpos);

```

**Auxiliary functions
for Remove()**

```

int      DoRemove(long badkey, BTreeNode& cNode);
void     DeleteKeyInNode(BTreeNode& aNode, int index);
KLRec    Successor(daddr_t subtree);
void     DeleteInLeaf(BTreeNode& leaf, int index);

void     Restore(BTreeNode& parent, BTreeNode& deficient,
               int index);
void     MergeOrCombineRight(BTreeNode& parent,
               BTreeNode& deficient, int index);
void     MergeOrCombineLeft(BTreeNode& parent,
               BTreeNode& deficient, int index);

void     MoveRight(BTreeNode& parent, BTreeNode& left,
               BTreeNode& right, int index);
void     MoveLeft(BTreeNode& parent, BTreeNode& left,
               BTreeNode& right, int index);

void     Combine(BTreeNode& parent, BTreeNode& left,
               BTreeNode& right, int index);

```

Data Members

Once all the auxiliary functions have been declared, the data members can be specified. A BTree object needs somewhere to store its housekeeping information in memory, two input/output file streams, and records of the size of the files.

```

HK          fHouseKeeping;
fstream     fTreeFile;
fstream     fDataFile;
long        fTreefile_size;
long        fDatafile_size;
};

```

The implementation file would start with the full declarations for struct KLRec (given earlier) and class BTreeNode:

```
class BTreeNode {
    friend class BTree;
private:

    int          SearchInNode(long keysought, int& index);
    void         InsertInNode(KLRec& data, daddr_t, int);
    void         InsertAtLeft(KLRec& data, daddr_t downlink);
    void         InsertAtRight(KLRec& data, daddr_t downlink);
    void         ShiftLeft(void);
    void         Compress(int index);

    int          NotFull() { return (n_data==MAX) ? 0 : 1; }
    int          Deficient() { return (n_data<MIN) ? 1 : 0; }
    int          MoreThanMinFilled()
                { return (n_data>MIN) ? 1 : 0; }

    int          n_data;
    KLRec        data[MAX]; /* 0..n_data-1 are filled */
    daddr_t      links[MAX+1]; /* 0..n_data are filled */
};
```

Class BTree is made a friend of BTreeNode so that code of member functions of class BTree can work directly with things like the n_data count or the links[] array.

Implementation of class BTreeNode

The member functions of class BTreeNode are all simple (some are just inline functions in the class declaration). Most of the rest involve iterative loops running through the entries. Functions InsertInNode() and Search() were both shown earlier.

The InsertAtLeft() and InsertAtRight() functions are used during move operations when fixing up deficient nodes. The InsertAtLeft() moves existing data over to make room, adds the new data and increments the counter. The InsertAtRight() function (not shown) is simpler; it merely adds the extra data in the first unused positions and then increments the counter.

```
void BTreeNode::InsertAtLeft(KLRec& info, daddr_t downlink)
{
    for(int i = n_data - 1; i >= 0; i--) {
        data[i+1] = data[i];
        links[i+2] = links[i+1];
    }
    links[1] = links[0];
    data[0] = info;
    links[0] = downlink;
```

```

        n_data++;
    }

```

Function `ShiftLeft()` moves `KLRec` and link entries leftwards after the leftmost entry has been removed. It gets used to tidy up a node after its least key has been removed during a "move" operation required to fix up a deficient sibling. Function `Compress()` is used to tidy up a parent node after one of its keys (that identified by argument `index`) has been removed as part of a Combine operation.

```

void BTreeNode::ShiftLeft(void)
{
    n_data--;
    links[0] = links [1];
    for(int i = 0; i < n_data; i++) {
        data[i] = data[i+1];
        links[i+1] = links[i+2];
    }
}

void BTreeNode::Compress(int index)
{
    n_data--;
    for(int i = index; i < n_data; i++) {
        data[i] = data[i+1];
        links[i+1] = links[i+2];
    }
}

```

Implementation of class `BTree`'s constructor and destructor

The constructor has to make up the names for the index and data files and then open them for both input and output. If the files can not be opened, the program should terminate (you could make the code "throw an exception" instead, see Chapter 29).

Opening the files

```

BTree::BTree(const char* filename)
{
    char buff[100];
    strcpy(buff,filename);
    strcat(buff, ".ndx");
    fTreeFile.open(buff, ios::in | ios::out);
    if(!fTreeFile.good()) {
        cerr << "Sorry, can't open BTree index file." << endl;
        exit(1);
    }
    strcpy(buff,filename);
    strcat(buff, ".dat");
    fDataFile.open(buff, ios::in | ios::out);
    if(!fDataFile.good()) {

```



```

        fDataFile.seekg(diskpos);
        datarec.ReadFrom(fDataFile);
    }

daddr_t BTree::MakeNewDiskBNode(BTreeNode& bnode)
{
    SaveBTreeNode(bnode, fTreefile_size);
    daddr_t diskpos = fTreefile_size;
    fTreefile_size += sizeof(BTreeNode);
    return diskpos;
}

daddr_t BTree::MakeNewDataRecord(KeyedStorableItem& data)
{
    SaveDataRecord(data, fDatafile_size);
    daddr_t diskpos = fDatafile_size;
    fDatafile_size += data.DiskSize();
    return diskpos;
}

```

Implementation of class BTree::Find()

The Find() function is given a key and a reference to a KeyedStorableItem (of course this will really be a reference to an instance of some class derived from class KeyedStorableItem). If Find() can find the key in the index file, it arranges for the KeyedStorableItem to load itself. Function Find() returns a 0/1 indicator (1 for success, 0 for failure i.e. key not present).

The function is a straightforward implementation of the iterative tree walk algorithm shown earlier:

```

int BTree::Find(long key, KeyedStorableItem& rec)
{
    daddr_t diskpos = fHouseKeeping.fRoot;
    while( diskpos != NO_DADDR) {
        int index;
        BTreeNode current;
        GetBTreeNode(current, diskpos);
        if (current.SearchInNode(key, index)) {
            daddr_t loc = current.data[index].fLocation;
            GetDataRecord(rec, loc);
            return 1;
        }
        diskpos = current.links[index];
    }
    return 0;
}

```


Implementation of class BTree::Add() and related functions

The Add() function itself is a straightforward implementation of the algorithm given earlier:

```
void BTree::Add(KeyedStorableItem& d)
{
    KLRec          rec_returned;
    daddr_t        filepos_returned;
    int            workflag;
    DoAdd(d, fHouseKeeping.fRoot, workflag,
          rec_returned, filepos_returned);
    if(workflag != 0) {
        BTreeNode new_root;
        new_root.n_data = 1;
        new_root.links[0] = fHouseKeeping.fRoot;
        new_root.data[0] = rec_returned;
        new_root.links [1] = filepos_returned;
        fHouseKeeping.fRoot = MakeNewDiskBNode(new_root);
    }
}
```

Make initial call to DoAdd() passing root node

Create new root if necessary

The recursive DoAdd() function has two input arguments and three output arguments. The input arguments are the new KeyedStorableItem (which is passed by reference) and the disk address of the next BTreeNode that is to be considered. The output arguments (all naturally passed by reference) are the flag variable (whose setting will indicate if any fix up operation is needed) together with any KLRec and disk address that needed to be returned to the caller.

```
void BTree::DoAdd(KeyedStorableItem& newData, daddr_t filepos,
                  int& return_flag, KLRec& return_KLRec,
                  daddr_t& return_diskpos)
{
    int            index;
    BTreeNode      current;
    long           key = newData.Key();

    return_flag = 0;

    if (filepos == NO_DADDR) {
        return_KLRec.fKey = key;
        return_KLRec.fLocation = MakeNewDataRecord(newData);
        return_diskpos = NO_DADDR;
        return_flag = 1;
        fHouseKeeping.fNumItems +=1;
        return;
    }
}
```

Create new data record

<i>Loading node and checking for key</i>	<pre> GetBTreeNode(current, filepos); int found = current.SearchInNode(key, index); if(found) { daddr_t location = current.data[index].fLocation; SaveDataRecord(newData, location); return; } </pre>
<i>Replace old data with new data</i>	
<i>Recursive call</i>	<pre> KLRec rec_coming_up; daddr_t diskpos_coming_up; int need_insert_or_split; DoAdd(newData, current.links[index], need_insert_or_split, rec_coming_up, diskpos_coming_up); if (need_insert_or_split == 0) return; </pre>
<i>Insert into current node</i>	<pre> if (current.NotFull()) current.InsertInNode(rec_coming_up, diskpos_coming_up, index); else { </pre>
<i>Or split current node</i>	<pre> Split(current, rec_coming_up, diskpos_coming_up, index, return_KLRec, return_diskpos); return_flag = 1; } SaveBTreeNode(current, filepos); } </pre>

Node splitting The algorithm for `Split()` was given earlier. Its implementation is simple as it merely needs to sort out whether the extra data are to be inserted belong in the existing (left) node, a new right node, or should be returned to the parent node. The different `SplitInsert` functions get called as required. The algorithm for `SplitInsertRight()` was given earlier. The example implementation code shown here is for the other two cases.

<i>Move contents of top half of node into new node</i>	<pre> void BTree::SplitInsertMiddle(BTreeNode& nodetosplit, KLRec& extradata, daddr_t extralink, int index, KLRec& return_KLRec, daddr_t& return_diskpos) { BTreeNode newNode; int i, j; for (i = MAX-1, j = MIN-1; i >= MIN; i--, j--) { newNode.links[j+1] = nodetosplit.links[i+1]; newNode.data[j] = nodetosplit.data[i]; } } </pre>
--	---

```

    }

    newNode.links[0] = extralink;
    newNode.n_data = nodetosplit.n_data = MIN;
    return_KLRec = extradata;
    return_diskpos = MakeNewDiskBNode(newNode);
}

void BTree::SplitInsertLeft(BTreeNode& nodetosplit,
    KLRec& extradata, daddr_t extralink, int index,
    KLRec& return_KLRec, daddr_t& return_diskpos)
{
    BTreeNode newNode;
    int i, j;
    for (i = MAX-1, j = MIN-1; i >= MIN; i--, j--) {
        newNode.links[j+1] = nodetosplit.links[i+1];
        newNode.data[j] = nodetosplit.data[i];
    }
    newNode.links[0] = nodetosplit.links[MIN];
    return_KLRec = nodetosplit.data[MIN-1];

    for (i--; i >= index; i--) {
        nodetosplit.links[i+2] = nodetosplit.links[i+1];
        nodetosplit.data[i+1] = nodetosplit.data[i];
    }

    Slot in the new information.
    nodetosplit.links[index+1] = extralink;
    nodetosplit.data[index] = extradata;
    newNode.n_data = nodetosplit.n_data = MIN;
    return_diskpos = MakeNewDiskBNode(newNode);
}

```

Save new node

Copy data across to new node

Pick value to be passed back
Shift values across to make room

Save new node

Implementation of class BTree::Remove() and related functions

Function Remove() itself is simple. As explained earlier, it merely needs to set up the initial recursive call and check for the (uncommon!) case of a need to change the root when the existing root becomes empty:

```

void BTree::Remove(long key)
{
    if(fHouseKeeping.fRoot == NO_DADDR)
        return;

    BTreeNode root_node;
    GetBTreeNode(root_node, fHouseKeeping.fRoot);

    (void) DoRemove(key, root_node);
}

```

```

        if (root_node.n_data == 0)
            fHouseKeeping.fRoot = root_node.links [0];
        else
            SaveBTreeNode(root_node, fHouseKeeping.fRoot);
    }

```

(The housekeeping data don't have to be saved immediately. They are saved by the destructor that closes the BTree files.)

DoRemove() The DoRemove() function implements the algorithm given earlier:

```

int BTree::DoRemove(long bad_key, BTreeNode& cNode)
{
    int          index;
    int          found = cNode.SearchInNode(bad_key, index);
    if(found) {
        DeleteKeyInNode(cNode, index);
        fHouseKeeping.fNumItems--;
        return cNode.Deficient();
    }

    daddr_t subtree = cNode.links[index];
    if(subtree == NO_DADDR)
        return 0;

    BTreeNode    nextNode;
    int           repairsneeded;
    GetBTreeNode(nextNode, subtree);
    repairsneeded = DoRemove(bad_key, nextNode);
    if (repairsneeded)
        Restore(cNode, nextNode, index);
    else
        SaveBTreeNode(nextNode, subtree);
    return cNode.Deficient();
}

```

Recursive call

The result returned by the function indicates whether the given node has become deficient. If it is deficient, then the caller will discover that "repairs (are) needed".

DeleteKeyInNode() The DeleteKeyInNode() function shows the details of setting up the mechanism to find a key to promote followed by the call back to DoRemove() to get rid of the original copy of this key.

```

void BTree::DeleteKeyInNode(BTreeNode& aNode, int index)
{
    if (aNode.links [index+1] == NO_DADDR) {
        DeleteInLeaf(aNode, index);
        return;
    }
}

```

Check for simple case, deletion in leaf

```

daddr_t subtree = aNode.links[index + 1];
aNode.data[index] = Successor(subtree);

long promotedskey = aNode.data[index].fKey;
BTreeNode      nxtNode;
GetBTreeNode(nxtNode, subtree);

int repairsneeded = DoRemove(promotedskey, nxtNode);
if (repairsneeded)
    Restore (aNode, nxtNode, index+1);
else
    SaveBTreeNode(nxtNode, subtree);
}

```

Promote data from right subtree

Removal of original of promoted data

The `DeleteKeyInLeaf()` function is trivial (shift higher keys left inside node, decrement count) and so is not shown.

The `Successor()` function involves an iterative search that runs down the left links as far as possible. The function returns the `KLRec` (key/data location pair) for the next key larger than that in the call to `Remove()`. *Successor*

```

KLRec BTree::Successor(daddr_t subtree)
{
    BTreeNode      aNode;
    while(subtree != NO_DADDR) {
        GetBTreeNode(aNode, subtree);
        subtree = aNode.links[0];
    }
    return aNode.data[0];
}

```

The `Restore()` function (which chooses which sibling gets used to move data or combine with the deficient node) is simple to implement from the algorithm given earlier. *Restore()*

The function `MergeOrCombineLeft()` illustrates the implementation for one of the two `MergeOrCombine` functions. The "right" function is similar. *MergeOrCombine Left()*

```

void BTree::MergeOrCombineLeft(BTreeNode& parent,
                               BTreeNode& deficient, int index)
{
    BTreeNode      left_nbr;
    daddr_t        left_daddr = parent.links[index-1];

    GetBTreeNode(left_nbr, left_daddr);
    if(left_nbr.MoreThanMinFilled()) {
        MoveRight (parent, left_nbr, deficient, index-1);
        SaveBTreeNode(left_nbr, left_daddr);
        SaveBTreeNode(deficient, parent.links[index]);
    }
    else {

```

```

        Combine(parent, left_nbr, deficient, index-1);
        SaveBTreeNode(left_nbr, left_daddr);
    }
}

```

MoveLeft() The explanation given in the previous section included an algorithm for MoveRight(); this is the implementation for MoveLeft():

```

void BTree::MoveLeft(BTreeNode& parent, BTreeNode& left,
                    BTreeNode& right, int index)
{
    KLRec rec_from_parent = parent.data[index];
    daddr_t downlink = right.links[0];
    left.InsertAtRight(rec_from_parent, downlink);
    parent.data[index] = right.data[0];
    right.ShiftLeft();
}

```

Combine() Function Combine() removes all data from the node given as argument right, shifting these values along with information from the parent down into the left node:

```

void BTree::Combine(BTreeNode& parent, BTreeNode& left,
                  BTreeNode& right, int index)
{
    KLRec rec_from_parent = parent.data[index];
    daddr_t downlink = right.links[0];
    left.InsertAtRight(rec_from_parent, downlink);

    for(int j = 0; j < right.n_data; j++)
        left.InsertAtRight(right.data[j], right.links[j+1]);

    parent.Compress(index);
}

```

24.2.5 Testing

The problems involved in testing the BTree code, and their solution, are exactly the same as for the AVL tree. The BTree algorithms are complex. There are many special cases. Things like promoting a key from a leaf several levels down in the tree are only going to occur once the tree has grown quite large. Operations like deleting the current root node are going to be exceedingly rare. You can't rely on simple interactive testing.

Instead, you use the technique of a driver program that invokes all the basic operations tens of thousands of times. The driver program has to be able to test the success of each operation, and terminate the program if it detects something like a supposedly deleted item being "successfully" found by a later search. A code coverage

tool has to be used in conjunction with the driver to make certain that every function has been executed.

The driver needed to test the BTree can be adapted from that used for the AVL tree. There are a few changes. For example, insertion of a "duplicate" key is not an error, instead the old data are overwritten. Data records are not dynamically created in main memory. Instead, the program can use a single data record in memory filling it in with data read from the tree during a search operation, or setting its data before an insert.

EXERCISES

- 1 Complete the implementation and testing of the AVL class.
- 2 Complete the implementation and testing of the BTree class.
- 3 Add a mechanism for "recycling" the space occupied by "dead" BTreeNodes.

25 Templates

Many of the ideas and examples presented in the last few chapters have illustrated ways of making code "general purpose". The collection classes in Chapter 21 were general purpose storage structures for any kind of data item that could reasonably be handled using `void*` (pointer to anything) pointers. Classes like `Queue` and `DynamicArray` don't depend in any way on the data elements stored and so are completely general purpose and can be reused in any program. Class `AVL` and class `BTree` from Chapter 24 are a little more refined. However they are again fairly general purpose, they work with data items that are instances of any concrete class derived from their abstract classes `Keyed` and `KeyedStorable`.

Templates represent one more step toward making code truly general purpose. The idea of a template is that it defines operations for an unbounded variety of related data types.

You can have "template functions". These define manipulations of some unspecified generic form of data element. The function will depend on the data elements being capable of being manipulated in particular ways. For example, a function might require that data elements support assignment (`operator=()`), equality and inequality tests (`operator==()`, `operator!=()`), comparisons (`operator<()`), and stream output (`operator<<(ostream&, ?&)` global operator function). But any data element, whether it be an `int` or an `Aircraft`, that supports all these operations is equally readily handled by the "function".

Template functions

Most often, template classes are classes whose instances look after other data elements. Collection classes are preeminent examples, their instances provide storage organization of other data elements. The member functions of these template classes are like individual template functions; they describe, in some general way, how to manipulate data elements. They work with any kind of data that supports the required operations.

Template classes

25.1 A GENERAL FUNCTION AND ITS SPECIALIZATIONS

Example, an idea for a general function To make things concrete, consider a simple example – the function `larger_of_two()`. This function takes two data elements as arguments, and returns the larger. In outline form, its code is as follows:

Outline code

```
Thing larger_of_two(Thing& thing1, Thing& thing2)
{
    if(thing1 > thing2)
        return thing1;
    else
        return thing2;
}
```

Versions for different data You can imagine actual implementations for different types of data:

Instance of outline for short integer data

```
short larger_of_two(short& thing1, short& thing2)
{
    if(thing1 > thing2)
        return thing1;
    else
        return thing2;
}
```

or

Instance of outline for double data

```
double larger_of_two(double& thing1, double& thing2)
{
    if(thing1 > thing2)
        return thing1;
    else
        return thing2;
}
```

or even

Instance of outline for "Boxes"

```
Box larger_of_two(Box& thing1, Box& thing2)
{
    if(thing1 > thing2)
        return thing1;
    else
        return thing2;
}
```

which would be practical provided that class `Box` defines an `operator>()` function and permits assignment e.g.:

```
long Box::Size() { return fwidth*fheight*fbreadth; }
```

```
int Box::operator>(Box& other) { return Size() > other.Size(); }

...
Box b1;
Box b2;
...
Box bigBox = larger_of_two(b1,b2);
```

*Assignment operator
needed to use result
of function*

The italicised outline version of the function is a "template" for the other special purpose versions. The outline isn't code that can be compiled and used. The code is the specialized versions made up for each data type.

Here, the different specializations have been hand coded by following the outline and changing data types as necessary. That is just a tiresome coding exercise, something that the compiler can easily automate.

Of course, if you want the process automated, you had better be able to explain to the compiler that a particular piece of text is a "template" that is to be adapted as needed. This necessitates language extensions.

25.2 LANGUAGE EXTENSIONS FOR TEMPLATES

25.2.1 Template declaration

The obvious first requirement is that you be able to tell the compiler that you want a "function definition" to be an outline, a template for subsequent specialization. This is achieved using template specifications.

For the example `larger_of_two()` function, the C++ template would be:

```
template<class Thing>
Thing larger_of_two(Thing& thing1, Thing& thing2)
{
    if(thing1 > thing2)
        return thing1;
    else
        return thing2;
}
```

The compiler reading the code first encounters the keyword `template`. It then knows that it is dealing with a declaration (or, as here, a definition) of either a template function or template class. The compiler suspends the normal processes that it uses to generate instruction sequences. Even if it is reading a definition, the compiler isn't to generate code, at least not yet. Instead, it is to build up some internal representation of the code pattern.

template keyword

Following the `template` keyword, you get template parameters enclosed within a `<` begin bracket and a `>` end bracket. The outline code will be using some name(s) to

Template parameters

represent data type(s); the example uses the name `Thing`. The compiler has to be warned to recognize these names. After all, it is going to have to adapt those parts of the code that involve data elements of these types.

You can have template functions and classes that are parameterized by more than one type of data. For example, you might want some "keyed" storage structure parameterized according to both the type of the data element stored and the type of the primary key used to compare data elements. (This might allow you to have data elements whose keys were really strings rather than the usual integers.) Here, multiple parameters are considered as an "advanced feature". They will not be covered; you will get to use such templates in your later studies.

Once it has read the `template <...>` header, the compiler will consume the following function or class declaration, or (member) function definition. No instructions get generated. But, the compiler remembers what it has read.

25.2.2 Template instantiation

The compiler uses its knowledge about a template function (or class) when it finds that function (class) being used in the actual code. For example, assuming that class `Box` and template function `larger_of_two()` have both been defined earlier in the file, you could have code like the following:

```

int main()
{
    Box b1(6,4,7);
    Box b2(5,5,8);
    ...
    Box bigbox = larger_of_two(b1,b2);
    cout << "The bigger box is "; bigbox.PrintOn(cout);
    ...
    double d1, d2;
    cout << "Enter values : "; cin >> d1 >> d2;
    ...
    cout << "The larger of values entered was : " <<
        larger_of_two(d1, d2) << endl;
    ...
}

```

Need version for Box

Need version for double

At the point where it finds `bigbox = larger_of_two(b1,b2)`, the compiler notes that it has to generate the version of the function that works for boxes. Similarly, when it gets to the output statement later on, it notes that it must generate a version of the function that works for doubles.

When it reaches the end of the file, it "instantiates" the various versions of the template.

Figure 25.1 illustrates the basic idea (and is close to the actual mechanism for some compilers).

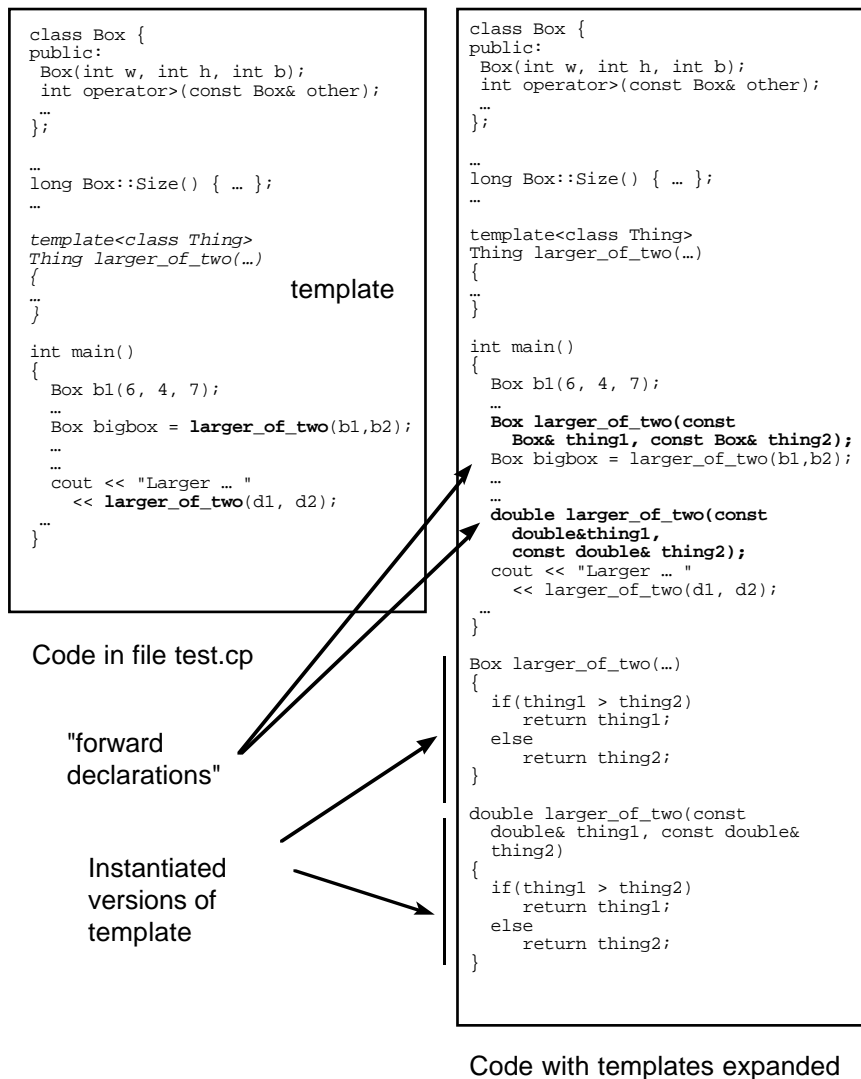


Figure 25.1 Illustration of conceptual mechanism for template instantiation.

The scheme shown in Figure 25.1 assumes that the compiler does a preliminary check through the code looking for use of templates. When it encounters uses of a template function, the compiler generates a declaration of a version specialized for the particular types of data used as arguments. When it gets to the end of the file, it uses the template to generate the specialized function, substituting the appropriate type (e.g. double) for the parametric type (Thing).

***Watch out for
compiler specific
mechanisms***

The specialized version(s) of the function can then be processed using the normal instruction generation mechanisms.

Templates are relatively new and the developers of compilers and linkers are still sorting out the best ways of dealing with them. The scheme shown in Figure 25.1 is fine if the entire program, with all its classes and template functions, is defined in a single file.

The situation is a little more complex when you have a program that is made up from many separately compiled and linked files. You may get references to template functions in several different files, but you probably don't want several different copies of "double larger_of_two(const double& d1, const double& d2)" – one copy for each file where this "function" is implicitly used.

***Compiler "pragmas"
to control
instantiation***

Compiler and linker systems have different ways of dealing with such problems. You may find that you have to use special compiler directives ("pragmas") to specify when and how you want template functions (and classes) to be instantiated. These directives are system's dependent. You will have to read the reference manuals for your IDE to see what mechanism you can use.

Another problem that you may find with your system is illustrated by the following:

```
template<class Whatsit>
void SortFun(Whatsit stuff[], int num_whatsists)
{
    // a standard sort algorithm
    ...
}

int main()
{
    long array1[10], array2[12], array3[17];
    int n1, n2, n3;
    // Get number of elements and data for three arrays
    ...
    // Sort array1
    SortFun(array1, n1);
    ...
    // Deal with array2
    SortFun(array2, n2);
    ...
}
```

Although in each case you would be sorting an array of long integers, the template instantiation mechanism might give you three versions of the code. One would be specialized for sorting an array of long integers with ten elements; the second would handle arrays with twelve elements; while the third would be specialized for arrays with seventeen elements.

This doesn't matter that much, it just wastes a little memory.

25.3 SPECIALIZED INSTANTIATIONS OF TEMPLATE CODE

Why does the compiler have to generate the specialized versions of template?

You should realize while that the code for the specialized data types may embody a similar pattern, the instruction sequences cannot be identical. All versions of the example `larger_of_two()` function take two address arguments (as previously explained references are handled internally as addresses). Thus, all versions will start with something like the following instruction sequence where the addresses of the two data elements are loaded into registers:

```
load a0 (address_register0) with stack_frame[...]
                                Thing& thing1
load a1 (address_register1) with stack_frame[...]
                                Thing& thing2
```

After that they have differences. For example, the generated code for the version with short integers will be something like:

```
load (integer register) r0 with two byte integer at a0
load (integer register) r1 with two byte integer at a1
compare r0, r1
branch if greater to pickthing1
// thing2 must be larger
store contents of r1 into 2-byte return field in stack frame
rts
pickthing1:
store contents of r0 into 2-byte return field in stack frame
rts
```

The code for the version specialized for doubles would be similar except that it would be working with 8-byte (or larger) data elements that get loaded into "floating point registers" and it would be a floating point comparison instruction that would be used.

There would be more significant differences in the case of `Boxes`. There the generated code would be something like:

```
push a0 onto stack
push a1 onto stack
call Box::operator>()
load r0 with contents of return-value from function
clean up stack
test r0
branch if greater to pickthing1
call routine to copy ... bytes into return area of stackframe
rts
...
```

The idea maybe the same, but the realization, the instantiation differs.

25.4 A TEMPLATE VERSION OF QUICKSORT

The Quicksort function group from Chapter 13 provides a slightly more realistic example than `larger_of_two()`.

The Quicksort function illustrated earlier made use of two auxiliary functions. There was a function `Partition()` that shuffled data elements in a particular subarray and an auxiliary `SelectionSort()` function that was used when the subarrays were smaller than a given limit size.

The version in Chapter 13 specified integer data. But we can easily make it more general purpose by just recasting all the functions as templates whose arguments (and local temporary variables) are of some generic type `Data`:

```

const int kSMALL_ENOUGH = 15;

Template
SelectionSort
    template<class Data>
    void SelectionSort(Data d[], int left, int right)
    {
        for(int i = left; i < right; i++) {
            int min = i;
            for(int j=i+1; j<= right; j++)
                if(d[j] < d[min]) min = j;
            Data temp = d[min];
            d[min] = d[i];
            d[i] = temp;
        }
    }

Template Partition
function
    template<class Data>
    int Partition(Data d[], int left, int right)
    {
        Data val =d[left];
        int lm = left-1;
        int rm = right+1;
        for(;;) {
            do
                rm--;
            while (d[rm] > val);

            do
                lm++;
            while( d[lm] < val);

            if(lm<rm) {
                Data tempr = d[rm];
                d[rm] = d[lm];
                d[lm] = tempr;
            }
        }
        else

```

```

        return rm;
    }
}

template<class Data>
void Quicksort(Data d[], int left, int right)
{
    if(left < (right-kSMALL_ENOUGH)) {
        int split_pt = Partition(d, left, right);
        Quicksort(d, left, split_pt);
        Quicksort(d, split_pt+1, right);
    }
    else SelectionSort(d, left, right);
}

```

Template QuickSort

The following test program instantiates two versions of the Quicksort group:

```

const int kBIG = 350;
int      data[kBIG];
double   data2[kBIG*2];

int main()
{
    int i;
    for(i=0; i < kBIG; i++)
        data[i] = rand() % 15000;

    Quicksort(data, 0, kBIG-1);

    for(i = 0; i < 100; i++) {
        cout << data[i] << ", ";
        if(5 == (i % 6)) cout << endl;
    }
    cout << "-----" << endl;

    for(i=0; i < kBIG*2; i++)
        data2[i] = (rand() % 30000)/7.0;

    Quicksort(data2, 0, kBIG-1);
    for(i = 0; i < 100; i++) {
        cout << data2[i] << ", ";
        if(5 == (i % 6)) cout << endl;
    }

    cout << "-----" << endl;

    ...
}

```

*Need Quicksort for integer**and a Quicksort for doubles*

This test program runs successfully producing its two lots of sorted output:

```

54,          134,          188,          325,          342,          446,

```

524,	585,	609,	610,	629,	639,
...					
3526,	3528,	3536,	3540,	----	
13.142,	25.285,	41.142,	49.571,	54.142,	58.142,
...					
1129.714,	1134.142,	1135.857,	1140,	----	

Beware of silly errors

But you must be a little bit careful. How about the following program:

```
char *msgs[10] = {
    "Hello World",
    "Abracadabra",
    "2,4,6,8 who do we appreciate C++ C++ C++",
    "NO",
    "Zanzibar",
    "Zurich",
    "Mystery",
    "help!",
    "!pleh",
    "tenth"
};

int main()
{
    int i;
    for(i = 0; i < 10; i++)
        cout << msgs[i] << ", " << endl;
    cout << "----" << endl;

    Quicksort(msgs, 0,9);
    for(i = 0; i < 10; i++)
        cout << msgs[i] << ", " << endl;
    cout << "----" << endl;

    return 0;
}
```

The output this time is as follows:

```
Hello World,
Abracadabra,
2,4,6,8 who do we appreciate C++ C++ C++,
NO,
Zanzibar,
Zurich,
Mystery,
help!,
!pleh,
"tenth",
```

```

----
Hello World,
Abracadabra,
2,4,6,8 who do we appreciate C++ C++ C++,
NO,
Zanzibar,
Zurich,
Mystery,
help!,
!pleh,
"tenth",
----

```

Problems! They aren't sorted; they are still in the original order! Has the sort routine got a bug in it? Maybe there is something wrong in the `SelectionSort()` part as its the only bit used with this small set of data.

A sort that didn't sort?

Another trial, one that might be more informative as to the problem:

What is really happening?

```

int main()
{
    int i;

    char *msgs2[10];
    msgs2[0] = msgs[3]; msgs2[1] = msgs[2];
    msgs2[2] = msgs[8]; msgs2[3] = msgs[9];
    msgs2[4] = msgs[7]; msgs2[5] = msgs[0];
    msgs2[6] = msgs[1]; msgs2[7] = msgs[4];
    msgs2[8] = msgs[6]; msgs2[9] = msgs[5];
    for(i = 0; i < 10; i++)
        cout << msgs2[i] << ", " << endl;
    cout << "-----" << endl;

    Quicksort(msgs2, 0,9);
    for(i = 0; i < 10; i++)
        cout << msgs2[i] << ", " << endl;

    return 0;
}

```

Shuffle the data

This produces the output:

```

NO,
2,4,6,8 who do we appreciate C++ C++ C++,
!pleh,
tenth,
help!,
Hello World,
Abracadabra,
Zanzibar,
Mystery,

```

*Sorted back into the
order of definition!*

```
Zurich,
----
Hello World,
Abracadabra,
2,4,6,8 who do we appreciate C++ C++ C++,
NO,
Zanzibar,
Zurich,
Mystery,
help!,
!pleh,
tenth,
```

This time the call to `Quicksort()` did result in some actions. The data in the array `msgs2[]` have been rearranged. In fact, they are back in exactly the same order as they were in the definition of the array `msgs[]`.

What has happened is that the `Quicksort` group template has been instantiated to sort *an array of pointers to characters*. No great problems there. A pointer to character is internally the same as an unsigned long. You can certainly assign unsigned longs, and you can also compare them.

*Sorting by location in
memory!*

We've been sorting the strings according to where they are stored in memory, and not according to their content! Don't get caught making silly errors like this.

You have to set up some extra support structure to get those strings sorted. The program would have to be something like the following code. Here a struct `Str` has been defined with associated `operator>()` and `operator<()` functions. A `Str` struct just owns a `char*` pointer; the comparison functions use the standard `strcmp()` function from the string library to compare the character strings that can be accessed via the pointers. A global `operator<<(ostream&, ...)` function had also to be defined to allow convenient stream output.

```
struct Str {
    char* mptr;
    int operator<(const Str& other) const
        { return strcmp(mptr, other.mptr) < 0; }
    int operator>(const Str& other) const
        { return strcmp(mptr, other.mptr) > 0; }
};

ostream& operator<<(ostream& out, const Str& s) {
    out << s.mptr; return out;
}

Str msgs[10] = {
    "Hello World",
    ...
    "tenth"
};
```

```

int main()
{
    int i;

    Quicksort(msgs, 0,9);
    for(i = 0; i < 10; i++)
        cout << msgs[i] << ", " << endl;

    return 0;
}

```

This program produces the output:

```

!pleh,
2,4,6,8 who do we appreciate C++ C++ C++,
Abracadabra,
Hello World,
Mystery,
NO,
Zanzibar,
Zurich,
help!,
tenth,

```

which is sorted. (In the standard ASCII collation sequence for characters, digits come before letters, upper case letter come before lower case letters.)

25.5 THE TEMPLATE CLASS "BOUNDED ARRAY"

Some languages, e.g. Pascal, have "bounded arrays". A programmer can define an array specifying both a lower bound and an upper bound for the index used to access the array. Subsequently, every access to the array is checked to make sure that the requested element exists. If an array has been declared as having an index range 9 to 17 inclusive, attempts to access elements 8 or 18 etc will result in the program being terminated with an "array bounds violation" error.

Such a structure is easy to model using a template `Array` class. We define a template that is parameterized by the type of data element that is to be stored in the array. Subsequently, we can have arrays of characters, arrays of integers, or arrays of any user defined structures.

What does an `Array` own? It will have to remember the "bounds" that are supposed to exist; it needs these to check accesses. It will probably be convenient to store the number of elements, although this could information could always be recomputed. An `Array` needs some space for the actual stored data. We will allocate this space on the heap, and so have a pointer data member in the `Array`. The space will be defined as an

An Array owns ...

An Array does ...

array of "things" (the type parameter for the class). Of course it is going to have to be accessed as a zero based array, so an `Array` will have to adjust the subscript.

An `Array` object will probably be asked to report its lower and upper subscript bounds and the number of elements that it owns; the class should provide access functions for these data. The main thing that an `Array` does is respond to the `[]` (array subscripting) operator.

The `[]` is just another operator. We can redefine `operator[]` to suit the needs of the class. We seem to have two ways of using elements of an array (e.g. `Counts` – an array of integers), as "right values":

```
n = Counts[6];
```

and as "left values":

```
Counts[2] += 10;
```

(The "left value" and "right value" terminology refers to the position of the reference to an array element relative to an assignment operator.) When used as a "right value" we want the value in the array element; when used as a "left value" we want the array element itself.

Reference to an array element

Both these uses are accommodated if we define `operator[]` as returning a reference to an array element. The compiler will sort out from context whether we need a copy of the value in that element or whether we are trying to change the element.

Assertions

How should we catch subscripting errors? The easiest way to reproduce the behaviour of a Pascal bounded array will be to have `assert()` macros in the accessing function that verify the subscript is within bounds. The program will terminate with an assertion error if an array is out of bounds.

Template class declaration

The template class declaration is as follows:

```
template<class DType>
class Array {
public:
    Array(int low_bnd, int high_bnd);
    ~Array();

    int    Size() const;
    int    Low() const;
    int    High() const;

    DType& operator[](int ndx);
    void   PrintOn(ostream& os);
private:
```

```

    void operator=(const Array &a);
    Array(const Array &a);
    DType *fData;
    int    flow;
    int    fhigh;
    int    fsize;
};

```

As in the case of a template function, the declaration starts with the keyword `template`, and then in `< >` brackets, we get the parameter specification. This template is parameterized by the type `DType`. This will represent the type of data stored in the array – `int`, `char`, `Aircraft`, or whatever.

The class has a constructor that takes the lower and upper bounds, and a destructor. Actual classes based on this template will be "resource managers" – they create a structure in the heap to store their data elements. Consequently, a destructor is needed to get rid of this structure when an `Array` object is no longer required.

Constructor

There are three simple access functions, `Size()` etc, and an extra – `PrintOn()`. This may not really belong, but it is useful for debugging. Function `PrintOn()` will output the contents of the array; its implementation will rely on the stored data elements being streamable.

Access functions

The `operator[]()` function returns a "reference to a `DType`" (this will be a reference to integer, or reference to character, or reference to `Aircraft` depending on the type used to instantiate the template).

Array subscripting

The declaration declares the copy constructor and `operator=()` function as `private`. This has been done to prevent copying and assignment of arrays. These functions will not be defined. They have been declared as `private` to get the compiler to disallow array assignment operations.

Copying of array disallowed

You could if you preferred make these functions public and provide definitions for them. The definitions would involve duplicating the data array of an existing object.

The data members are just the obvious integer fields needed to hold the bounds and a pointer to `DType` (really, a pointer to an array of `DType`).

Template class definition

All of the member functions must be defined as template functions so their definitions will start with the keyword `template` and the parameter(s).

The parameter must also be repeated with each use of the scope qualifier operator (`::`). We aren't simply defining the `Size()` or `operator[]()` functions of class `Array` because there could be several class `Arrays` (one for integers, one for `Aircraft` etc). We are defining the `Size()` or `operator[]()` functions of the class `Array` that has been parameterized for a particular data type. Consequently, the parameter must modify the class name.

The constructor is straightforward. It initializes the array bounds (first making certain that the bounds have been specified from low to high) and creates the array on the heap:

Constructor

```
template<class DType>
Array<DType>::Array(int low_bnd, int high_bnd)
{
    assert(low_bnd < high_bnd);
    fsize = high_bnd - low_bnd;
    fsize += 1; // low to high INCLUSIVE
    flow = low_bnd;
    fhigh = high_bnd;

    fData = new DType[fsize];
}
```

Destructor and simple access functions The destructor simply gets rid of the storage array. The simple access functions are all similar, only `Size()` is shown:

```
template<class DType>
Array<DType>::~~Array()
{
    delete [] fData;
}

template<class DType>
int Array<DType>::Size() const
{
    return fsize;
}
```

Output function Function `PrintOn()` simply runs through the array outputting the values. The statement `os << fData[i]` has implications with respect to the instantiation of the array. Suppose you had class `Point` defined and wanted a bounded array of instances of class `Point`. When the compiler does the work of instantiating the `Array` for `Points`, it will note the need for a function operator `<< (ostream&, const Point&)`. If this function is not defined, the compiler will refuse to instantiate the template. The error message that you get may not be particularly clear. If you get errors when instantiating a template class, start by checking that the data type used as a parameter does support all required operations.

```
template<class DType>
void Array<DType>::PrintOn(ostream& os)
{
    for(int i=0;i<fsize;i++) {
        int j = i + flow;
        os << "[" << setw(4) << j << "]\t:";
        os << fData[i];
    }
}
```

```

        os << endl;
    }
}

```

The operator[]() function sounds terrible but is actually quite simple:

Array indexing

```

template<class DType>
DType& Array<DType>::operator[](int ndx)
{
    assert(ndx >= flow);
    assert(ndx <= fhigh);

    int j = ndx - flow;

    return fData[j];
}

```

The function starts by using assertions to check the index given against the bounds. If the index is in range, it is remapped from [flow ... fhigh] to a zero based array and then used to index the fData storage structure. The function simply returns fData[j]; the compiler knows that the return type is a reference and it sorts out from left/right context whether it should really be using an address (for left value) or contents (for right value).

Template class use

The following code fragments illustrate simple applications using the template Array class. The first specifies that it wants an Array that holds integers. Note the form of the declaration for Array a; it is an Array, parameterized by the type int.

*Code fragments using
Array template*

```

#include <iostream.h>
#include <iomanip.h>
#include <ctype.h>
#include <assert.h>

template<class DType>
class Array {
public:
    ...
};

// Definition of member functions as shown above
...

int main()
{

```

Define an Array of integers

```
Array<int>    a(3,7);

cout << a.Size() << endl;

a[3] = 17; a[4] = 21; a[5] = a[3] + a[4];
a[6] = -1; a[7] = 123;

a.PrintOn(cout);

return 0;
}
```

The second example uses an `Array<int>` for counters that hold letter frequencies:

```
int main()
{
    Array<int> letter_counts('a', 'z');
    for(char ch = 'a'; ch <= 'z'; ch++)
        letter_counts[ch] = 0;
    cout << "Enter some text, terminate with '.'" << endl;

    cin.get(ch);
    while(ch != '.') {
        if(isalpha(ch)) {
            ch = tolower(ch);
            letter_counts[ch] ++;
        }
        cin.get(ch);
    }

    cout << endl << "-----" << endl;
    cout << "The letter frequencies are " << endl;
    letter_counts.PrintOn(cout);
    return 0;
}
```

(The character constants 'a' and 'z' are perfectly respectable integers and can be used as the low, high arguments needed when constructing the array.)

These two examples have both assumed that the template class is declared, and its member functions are defined, in the same file as the template is employed. This simplifies things, there are no problems about where the template should be instantiated. As in the case of template functions, things get a bit more complex if your program uses multiple files. The mechanisms used by your IDE for instantiating and linking with template classes will be explained in its reference manual..

25.6 THE TEMPLATE CLASS QUEUE

The collection classes from Chapter 21 are all candidates for reworking as template classes. The following is an illustrative reworking of class `Queue`. The `Queue` in Chapter 21 was a "circular buffer" employing a fixed sized array to hold the queued items.

The version in Chapter 21 held pointers to separately allocated data structures. So, its `Append()` function took a `void*` pointer argument, its `fData` array was an array of pointers, and function `First()` returned a `void*`.

The new version is parameterized with respect to the data type stored. The declaration specifies a `Queue` of `Obj`s, and an `Obj` is anything you want it to be. We can still have a queue that holds pointers. For example, if we already have separately allocated `Aircraft` objects existing in the heap we can ask for a queue that makes `Obj` an `Aircraft*` pointer. But we can have queues that hold copies of the actual data rather than pointers to separate data. When working with something like `Aircraft` objects, it is probably best to have a queue that uses pointers; but if the data are of a simpler data type, it may be preferable to make copies of the data elements and store these copies in the `Queue`'s array. Thus, it is possible to have a queue of characters (which wouldn't have been very practical in the other scheme because of the substantial overhead of creating single characters as individual structures in the heap).

*Choice of data copies
or data pointers*

Another advantage of this template version as compared with the original is that the compiler can do more type checking. Because the original version of `Queue::Append()` just wanted a `void*` argument, any address was good enough. This is a potential source of error. The compiler won't complain if you mix up all addresses of different things (`char*`, `Aircraft*`, addresses of functions, addresses of automatic variables), they are all just as good when all that is needed is a `void*`.

Type safety

When something was removed from that queue, it was returned as a `void*` pointer that then had to be typecast to a useable type. The code simply had to assume that the cast was appropriate. But if different types of objects were put into the queue, all sorts of problems would arise when they were removed and used.

This version has compile time checking. The `Append()` function wants an `Obj` – that is an `Obj` that matches the type of the instantiated queue. If you ask for a queue of `Aircraft*` pointers, the only thing that you can append is an `Aircraft*`; when you take something from that queue it will be an `Aircraft*` and the compiler can check that you use it as such. Similarly, a queue defined as a queue of `char` will only allow you to append a `char` value and will give you back a `char`. Such static type checking can eliminate many errors that result from careless coding.

The actual declaration of the template is as follows:

```
template<class Obj>
class Queue {
public:
    Queue();

    void    Append(Obj newobj);
    Obj     First();
```

```

        int    Length() const;
        int    Full() const;
        int    Empty() const;

private:
        Obj    fdata[QSIZE];
        int    fp;
        int    fg;
        int    fcount;
};

```

The definitions of the member functions are:

Access functions

```

template<class Obj>
inline int Queue<Obj>::Length() const { return fcount; }

template<class Obj>
inline int Queue<Obj>::Full() const {
    return fcount == QSIZE;
}

template<class Obj>
inline int Queue<Obj>::Empty() const {
    return fcount == 0;
}

```

Constructor

```

template<class Obj>
Queue<Obj>::Queue()
{
    fp = fg = fcount = 0;
}

```

The Append() and First() functions use assert() macros to verify correct usage:

```

template<class Obj>
void Queue<Obj>::Append(Obj newobj)
{
    assert(!Full());
    fdata[fp] = newobj;
    fp++;
    if(fp == QSIZE)
        fp = 0;
    fcount++;
    return;
}

template<class Obj>
Obj Queue<Obj>::First(void)
{
    assert(!Empty());
}

```

```
    Obj temp = fdata[fg];
        fg++;
    if(fg == QSIZE)
        fg = 0;
    fcount--;
    return temp;
}
```

The following is an example code fragment that will cause the compiler to generate an instance of the template specialized for storing char values:

```
int main()
{
    Queue<char> aQ;
    for(int i=0; i < 100; i++) {
        int r = rand() & 1;
        if(r) {
            if(!aQ.Full()) {
                char ch = 'a' + (rand() % 26);
                aQ.Append(ch);
                cout << char(ch - 'a' + 'A');
            }
        }
        else {
            if(!aQ.Empty()) cout << aQ.First();
        }
    }
    return 0;
}
```

26 Exceptions

With class based programs, you spend a lot of time developing classes that are intended to be used in many different programs. Your classes must be reliable. After all, no one else is going to use your classes if they crash the program as soon as they run in to data that incorrect, or attempt some input or output transfer that fails.

You should aim to build defences into the code of your classes so that bad data won't cause disasters. So, how might you get given bad data and what kind of defence can you build against such an eventuality?

"Bad data" most frequently take the form of inappropriate arguments passed in a call to a member function that is to be executed by an instance of one of your classes, or, if your objects interact directly with the user, will take the form of inappropriate data directly input by the user.

You can do something about these relatively simple problems. Arguments passed to a function can be checked using an `assert()` macro. Thus, you can check that the pointers supposed to reference data structures are not `NULL`, and you can refuse to do things like sort an array with -1234 data elements (e.g. `assert(ptr1 != NULL); assert((n>1) && (n<SHRT_MAX))`). Data input can be handled through loops that only terminate if the values entered are appropriate; if a value is out of range, the loop is again cycled with the user prompted to re-enter the data.

Simple checks on data

Such basic checks help. Careless errors in the code that invokes your functions are soon caught by `assert()` macros. Loops that validate data are appropriate for interactive input (though they can cause problems if, later, the input is changed to come from a file).

But what happens when things go really wrong? For example, you hit "end of file" when trying to read input, or you are given a valid pointer but the referenced data structure doesn't contain the right information, or you are given a filename but you find that the file does not exist.

Code like:

```
void X::GetParams(const char fname[])
{
```

```
ifstream in(fname, ios::in | ios::nocreate);
assert(in.good());
```

or

```
ifstream in(fname, ios::in | ios::nocreate);
if(!in.good()) {
    cout << "Sorry, that file can't be found. Quitting."
          << endl;
    exit(1);
}
```

catches the error and terminates via the "assertion failed" mechanism or the explicit `exit()` call.

Terminating the program – maybe the only thing to do?

In general, terminating is about all you can do if you have to handle an error at the point that it is detected.

You don't know the context of the call to `X::GetParams()`. It might be an interactive program run directly from the "console" window; in that case you could try to handle the problem by asking for another filename. But you can't assume that you can communicate with the user. Function `X::GetParams()` might be being called from some non-interactive system that has read all its data from an input file and is meant to be left running to generate output in a file that gets examined later.

But other parts of code might have been able to deal with error!

You don't know the context, but the person writing the calling code does. The caller may know that the filename will have just been entered interactively and so prompting for a new name is appropriate when file with a given name cannot be found. Alternatively, the caller may know that a "params" file should be used if it can be opened and read successfully; if not, a set of "default parameters" should be used in subsequent calculations.

Error detection and error handling separated by long call chain

It is a general problem. One part of the code can detect an error. Knowledge of what to do about the error can only exist in some separate part of the code. The only connection between the two parts of the code is the call chain; the code that can detect an error will have been called, directly or indirectly, from the code that knows what to do about errors.

The call chain between the two parts of the code may be quite lengthy:

```
caller --- (knows what to do if error occurs)
functionA
  procedure B
    recursive function X
      recursive function X (again)
        recursive function X (again again)
          procedure Y
            function Z
              Z can detect error
```


While that might be an extreme case, there is often a fairly lengthy chain of calls separating originator and final routine. (For example, when you use menu File/Save in a Mac or PC application, the call chain will be something like `Application::HandleEvent()`, `View::HandleMenu()`, `Document::HandleMenu()`, `Document::DoSaveMenu()`, `Document::DoSave()`, `Document::OpenFile()`; error detection would be in `OpenFile()`, error handling might be part way up the chain at `DoSaveMenu()`.)

If you don't want the final function in the call chain, e.g. function `Z`, to just terminate the program, you have to have a scheme whereby this function arranges for a description of the error to be passed back up the call chain to the point where the error can be handled.

Pass an error report back up the call chain

The error report has to be considered by each of the calling functions as the call chain is unwound. The error handling code might be at any point in the chain. Intermediate functions, those between the error handler and the error detector, might not be able to resolve the error condition but they might still need to do some work if an error occurs.

To see why intermediate functions might need to do work even when they can't resolve an error, consider a slight elaboration of the `X::GetParams()` example. The `X::GetParams()` function now creates two separately allocated, semi-permanent data structures. These are a `ParamBlock` structure that gets created in the heap and whose address is stored in the assumed global pointer `gParamData`, and some form of "file descriptor" structure that gets built in the system's area of memory as part of the process of "constructing" the `ifstream`. Once it has built its structures, `GetParams()` calls `X::LoadParamBlock()` to read the data:

Need to tidy up as call chain unwinds.

```
void X::GetParams(const char *fname)
{
    ifstream in(fname, ios::in | ios::nocreate);

    some code that deals with cases where can't open the file

    gParamData = new ParamBlock;
    ...
    LoadParamBlock();
    ...
}

void X::LoadParamBlock(ifstream& in)
{
    for(int i=0;i<kPARAMS;i++) {
        code to try to read next of parameters
        ...
        code to deal with problems like
            unexpected end of file
            invalid data
    }
}
```

If `LoadParamBlock()` runs into difficulties, like unexpected end of file (for some reason, the file contains fewer data items than it really should), it will need to pass back an error report.

The function `GetParams()` can't deal with the problem; after all, the only solution will be to use another file that has a complete set of parameters. So `X::GetParams()` has to pass the error report back to the function from where it was called.

However, if `X::GetParams()` simply returns we may be left with two "orphaned" data structures. The file might not get closed if there is an abnormal return from the function. The partly filled `ParamBlock` structure won't be of any use but it will remain in the heap.

The mechanism for passing back error reports has to let intermediate functions get a chance to do special processing (e.g. delete that useless `ParamBlock` structure) and has to make sure that all normal exit code is carried out (like calling destructors for automatic objects – so getting the file closed).

*Describing errors
that can be handled*

A function that hopes to deal with errors reported by a called function has to associate with the call a description of the errors that can be handled. Each kind of error that can be handled may have a different block of handling code.

*Code to handle
exceptional
conditions*

Obviously, the creation of an error report, the unwinding of the call chain, and the eventual handling of the error report can't use the normal "return from function" mechanisms. None of the functions has defined its return type as "error report". None of the intermediate functions is terminating normally; some may have to execute special code as part of the unwinding process.

The error handling mechanism is going to need additional code put into the functions; code that only gets executed under exceptional conditions. There has to be code to create an error report, code to note that a function call has terminated abnormally with an error report, code to tidy up and pass the error report back up the call chain, code to check an error report to determine whether it is one that should be dealt with, and code to deal with chosen errors.

The C language always provided a form of "emergency exit" from a function that could let you escape from problems by jumping back up the call chain to a point where you could deal with an error (identified by an integer code). But this `setjmp()`, `longjmp()` mechanism isn't satisfactory as it really is an emergency exit. Intermediate functions between the error detector and the error handler don't get given any opportunity to tidy up (e.g. close files, delete unwanted dynamic structures).

*Language extensions
needed*

A programming language that wants to support this form of error handling properly has to have extensions that allow the programmer to specify the source for all this "exception handling" code. The implementation has to arrange that error reports pass back up the call chain giving each function a chance to tidy up even if they can't actually deal with the error.

C++ has added such "exception handling". (This is a relatively recent addition, early 1990s, and may not be available in older versions of the compilers for some of the IDEs). Exception handling code uses three new constructs, `throw`, `try`, and `catch`:

- `throw`
Used in the function that detects an error. This function creates an error report and "throws" it back up the call chain.
- `try`
Used in a function that is setting up a call to code where it can anticipate that errors might occur. The `try` keyword introduces a block of code (maybe a single function call, maybe several statements and function calls). A `try` block has one or more `catch` blocks associated with it.
- `catch`
The `catch` keyword specifies the type of error report handled and introduces a block of code that has to be executed when such an error report gets "thrown" back at the current function. The code in this block may clear up the problem (e.g. by substituting the "default parameters" for parameters from a bad file). Alternatively, this code may simply do special case tidying up, e.g. deleting a dynamic data structure, before again "throwing" the error further back up the call chain.

Rather than "error report", the usual terminology is "exception". A function may *throw an exception*. Another function may *catch an exception*.

26.1 C++'S EXCEPTION MECHANISM

As just explained, the `throw` construct allows the programmer to create an exception (error report) and send it back up the call chain. The `try` construct sets up a call to code that may fail and result in an exception being generated. The `catch` construct specifies code that handles particular exceptions.

The normal arrangement of code using exceptions is as follows. First, there is the main driver function that is organizing things and calling the code that may fail:

*Driver with try {...}
and catch {...}*

```
void GetSetup()
{
    // Create objects ...
    gX = new X;
    ...
    // Load parameters and perform related initialization,
    // if this process fails substitute default params
    try {
        LoadParams();
        ...
    }
    catch (Error e) {
        // File not present, or data no good,
        // copy the default params
        gParamData = new ParamBlock(defaultblock);
        ...
    }
}
```

```

        // better log error for reference
        cerr << "Using default params because ";
        if(e.Type() == Error::eBADFILE)
            cerr << "Bad file" << endl;
        else
            if(e.Type() == Error::eBADCONTENT)
                ...
            }
        ...
    }

```

This driver has the `try` clause bracketing the call that may fail and the following `catch` clause(s). This one is set to catch all exceptions of type `Error`. Details of the exception can be found in the variable `e` and can be used in the code that deals with the problem.

Intermediate functions

There may be intermediate functions:

```

void LoadParams()
{
    NameStuff      n;
    n.GetFileName();
    ...
    gX->GetParams(n.Name());
    ...
    return;
}

```

As this `LoadParams()` function has no `try catch` clauses, all exceptions thrown by functions that it calls get passed back to the calling `GetSetup()` function.

As always, automatics like `NameStuff` are initialized by their constructors on entry and their destructors, if any, are executed on exit from the function. If `NameStuff` is a resource manager, e.g. it allocates space on the heap for a name, it will have a real destructor that gets rid of that heap space. The compiler will insert a call to the destructor just before the final `return`. If a compiler is generating code that allows for exceptions (most do), then in addition there will be code that calls the destructor if an exception passes back through this intermediate `LoadParams()` function.

Functions that throw exceptions

The `X::GetParams()` function may itself throw exceptions, e.g. if it cannot open its file:

```

void X::GetParams(const char *fname)
{
    ifstream in(fname, ios::in | ios::nocreate);

    if(!in.good())
        throw(Error(Error::eBADFILE));
}

```

It may have to handle exceptions thrown by functions it calls. If it cannot fully resolve the problem, it re-throws the same exception:

```

    gParamData = new ParamBlock;

    // File open OK, load the params
    try {
        LoadParamBlock(in);
    }
    catch(Error e) {
        // Rats! Param file must be bad.
        // Get rid of param block and inform caller
        delete gParamData;
        gParamData = NULL;
        // Pass the same error back to caller
        throw;
    }
}

```

The function `X::LoadParamBlock()` could throw an exception if it ran into problems reading the file:

```

void X::LoadParamBlock(istream& in)
{
    for(int i=0;i<kPARAMS;i++) {
        code to try to read next of parameters
        if(!in.good())
            throw(Error(Error::eBADCONTENT));
    }
}

```

But what are these "exceptions" like the `Errors` in these examples?

Things to throw

Essentially, they can be any kind of data element from a simple `int` to a complex struct. Often it is worthwhile defining a very simple class:

```

class Error{
public:
    enum ErrorType { eBADFILE, eBADCONTENT };
    Error(ErrorType whatsbad) { this->fWhy = whatsbad; }
    ErrorType Type() { return this->fWhy; }
private:
    ErrorType fWhy;
};

```

(You might also want a `Print()` function that could be used to output details of errors.)

You might implement your code for class `X` using exceptions but your colleagues using your code might not set up their code to catch exceptions. Your code finds a file that want open and so throws an `eBADFILE` Error. What now?

Suppose there are no catchers

The exception will eventually get back to `main()`. At that point, it causes the program to terminate. It is a bit like an assertion failing. The program finishes with a system generated error message describing an uncaught exception.

26.2 EXAMPLE: EXCEPTIONS FOR CLASS NUMBER

Class `Number`, presented in Section 19.3, had to deal with errors like bad inputs for its string constructor (e.g. "1419070"), divide by zero, and other operations that could lead to overflow. Such errors were handled by printing an error message and then terminating the program by a call to `exit()`. The class could have used exceptions instead. A program using `Numbers` might not be able to continue if overflow occurred, but an error like a bad input might be capable of resolution.

We could define a simple class whose instances can be "thrown" if a problem is found when executing a member function of class `Number`. This `NumProblem` class could be defined as follows:

```
class NumProblem {
public:
    enum    NProblemType
            { eOVERFLOW, eBADINPUT, eZERODIVIDE };
    NumProblem(NProblemType whatsbad);
    void PrintOn(ostream& out);
private:
    NProblemType    fWhy;
};
```

Its implementation would be simply:

```
NumProblem::NumProblem(NProblemType whatsbad)
{ fWhy = whatsbad; }

void NumProblem::PrintOn(ostream& out)
{
    switch(fWhy) {
    case eOVERFLOW:  cout << "Overflow" << endl; break;
    case eBADINPUT:  cout << "Bad input" << endl; break;
    case eZERODIVIDE: cout << "Divide by zero" << endl; break;
    }
}
```

Some of the previously defined member functions of class `Number` would need to be modified. For example, the constructor that creates a number from a string would now become:

```
Number::Number(char numstr[])
{
```

```

    for(int i=0;i<=kMAXDIGITS;i++) fDigits[i] = 0;
    fPosNeg = 0; /* Positive */
    ...
    while(numstr[i] != '\0') {
        if(!isdigit(numstr[i]))
            throw(NumProblem(NumProblem::eBADINPUT));
        fDigits[pos] = numstr[i] - '0';
        i++;
        pos--;
    }
    ...
}

```

(The version in Section 19.3 had a call to `exit()` where this code throws the "bad input" `NumProblem` exception.)

Similarly, the `Number::DoAdd()` function would now throw an exception if the sum of two numbers became too large:

```

void Number::DoAdd(const Number& other)
{
    int    lim;
    int    Carry = 0;
    lim = (fLength >= other.fLength) ? fLength :
        other.fLength;
    for(int i=0;i<lim;i++) {
        ...
    }
    fLength = lim;
    if(Carry) {
        if(fLength == kMAXDIGITS)
            throw(NumProblem(NumProblem::eOVERFLOW));
        fDigits[fLength] = 1;
        fLength++;
    }
}

```

and the `Number::Divide()` function could throw a "zero divide" exception:

```

Number Number::Divide(const Number& other) const
{
    if(other.Zero_p())
        throw(NumProblem(NumProblem::eZERODIVIDE));
    Number Result; // Initialized to zero

    ...

    return Result;
}

```

Code using instances of class `Number` can take advantage of these extensions. If the programmer expects that problems might occur, and knows what to do when things do go wrong, then the code can be written using `try { ... }` and `catch (...) { ... }`. So, for example, one can have an input routine that handles cases where a `Number` can't be successfully constructed from a given string:

```
Number GetInput()
{
    int    OK = 0;
    Number n1;
    while(!OK) {
        try {
            char buff[120];
            cout << "Enter number : ";
            cin.getline(buff,119,'\n');
            Number n2(buff);
            OK = 1;
            n1 = n2;
        }
        catch (NumProblem p) {
            cout << "Exception caught" << endl;
            p.PrintOn(cout);
        }
    }
    return n1;
}
```

The following test program exercises this exception handling code:

```
int main()
{
    Number n1;
    Number n2;
    n1 = GetInput();
    n2 = GetInput();

    Number n3;
    n3 = n1.Divide(n2);

    cout << "n1 "; n1.PrintOn(cout); cout << endl;

    cout << "n2 "; n2.PrintOn(cout); cout << endl;

    cout << "n3 "; n3.PrintOn(cout); out << endl;

    return 0;
}
```

The following recordings illustrate test runs:


```

$ Numbers
Enter number : 710
Exception caught
Bad input
Enter number : 710
Exception caught
Bad input
Enter number : 710
Enter number : 5
n1 710
n2 5
n3 142

```

The erroneous data, characters 'l' (ell) and 'O' (oh), cause exceptions to be thrown. These are handled in the code in `GetNumber()`.

The main code doesn't have any `try {} catch { }` constructs around the calculations. If a calculation error results in an exception, this will cause the program to terminate:

```

$ Numbers
Enter number : 69
Enter number : 0
Run-time exception error; current exception: NumProblem
    No handler for exception.
Abort - core dumped

```

The input values of course cause a "zero divide" `NumProblem` exception to be thrown. (These examples were run on a Unix system. The available versions of the Symantec compiler did not support exceptions.)

26.3 IDENTIFYING THE EXCEPTIONS THAT YOU INTEND TO THROW

If you intend that your function or class member function should throw exceptions, then you had better warn those who will use your code.

A function (member function) declaration can include "exception specifications" as part of its prototype. These specifications should be repeated in the function definition. An exception specification lists the types of exception that the function will throw.

Thus, for class `Number`, we could warn potential users that some member functions could result in `NumProblem` exceptions being thrown:

*Exception
specification*

```

class Number {
public:
    Number();
    Number(long);

```

Specification in a function declaration

```

Number(char numstr[]) throw(NumProblem);

...

Number Add(const Number& other) const throw(NumProblem);
Number Subtract(const Number& other) const
    throw(NumProblem);
Number Multiply(const Number& other) const
    throw(NumProblem);
Number Divide(const Number& other) const
    throw(NumProblem);

int    Zero_p() const;
void   MakeZero();

...
};

```

The constructor that uses the string throws exceptions if it gets bad input. Additions, subtraction, and multiplication operations throw exceptions if they get overflow. Division throws an exception if the divisor is zero.

The function definitions repeat the specifications:

Specification in a function definition

```

Number::Number(char numstr[]) throw(NumProblem)
{
    for(int i=0;i<=kMAXDIGITS;i++) fDigits[i] = 0;
    ...
}

```

Exception specifications are appropriate for functions that don't involve calls to other parts of the system. All the functions of class `Number` are "closed"; there are no calls to functions from other classes. Consequently, the author of class `Number` knows exactly what exceptions may get thrown and can provide the specifications.

Don't always try to specify

You can not always provide a complete specification. For example, consider class `BTree` that stores instances of any concrete class derived from some abstract `KeyedStorable` class. The author of class `BTree` may foresee some difficulties (e.g. disk transfer failures) and so may define some `DiskProblem` exceptions and specify these in the class declaration:

```

void BTree::Insert(const KeyedStorable& data)
    throw(DiskProblem);

```

However, problems with the disk are not the only ones that may occur. The code for `Insert()` will ask the `KeyedStorable` data item to return its key or maybe compare its key with that of some other `KeyedStorable` item. Now the specific concrete subclass derived from `KeyedStorable` might throw a `BadKey` exception if there is something wrong with the key of a record.

This `BadKey` exception will come rumbling back up the call chain until it tries to terminate execution of `BTree::Insert`. The implementation of exceptions is supposed to check what is going on. It knows that `BTree::Insert()` promises to throw only `DiskProblem` exceptions. Now it finds itself confronted with something that claims to be a `BadKey` exception.

In this case, the exception handling mechanism calls `unexpected()` (a global run-time function that terminates the program). *unexpected()*

If you don't specify the exceptions thrown by a function, any exception can pass back through that function to be handled at some higher level. An exception specification is a filter. Only exceptions of the specified types are allowed to be passed on; others result in program termination.

In general, if you are implementing something concrete and closed like class `Number` then you should use exception specification. If you are implementing something like class `BTree`, which depends indirectly on other code, then you should not use exception specifications.

Of course, the documentation that you provide for your function or class should describe any related exceptions irrespective of whether the (member) function declarations contain explicit exception specifications. *Remember the documentation*

27

27 Example: Supermarket

This chapter presents a single example program. The program is a simulation, slightly more elaborate than that in the AirController/Aircraft example in Chapter 20. The program makes limited use of inheritance. The use of inheritance here is largely to simplify some aspects of the design; there isn't much sharing of code among interrelated classes.

The number of objects present at run time is somewhat larger in this example than earlier examples. At any one stage of the simulation there may be as many as two hundred to three hundred objects. The number of objects created and destroyed during a run will be something in the range one thousand to three thousand. Naturally, since the program will be creating and working with large numbers of objects, some of the standard collection classes appear. Although there are hundreds of objects most are instances of the same class; the program uses fewer than ten different concrete classes.

The program makes use of a class that came with the very first class library released with C++. This "histogram" class (based on the version in the original "tasks" library) collects data that are to be displayed as a histogram and, when data collection is completed, produces a printout.

27.1 BACKGROUND AND PROGRAM SPECIFICATION

Helping the supermarket's manager

The manager of a large supermarket wants to explore possible policies regarding the number of checkout lanes that are necessary to provide a satisfactory service to customers.

The number of customers, and the volume of their purchases, varies quite markedly at different times of day and the number of open checkout lines should be adjusted to match the demand. Customers will switch to rival stores if they find that they have to queue too long in this supermarket so, at times, it may be necessary to have a large number of checkouts open. However, the manager must also try to minimize idle time at checkouts; there is no point keeping checkouts open if there are no customers to serve.

The manager is proposing a policy in which supermarket staff will be assigned to "general duties". Staff may be restocking shelves, cleaning, recovering trolleys from parking lots, or operating checkouts. Staff can be switched "immediately" between different roles. The manager (or some deputy) will make regular checks on activity in the shop and, if appropriate, may assign more staff to open extra checkout lanes or may close idle checkouts so freeing the operators to perform other duties.

Rules for running the supermarket

The manager is proposing some "rules of thumb" (heuristics) for choosing when to open or close checkouts and for the frequency of scheduling such decisions. However, before trying these rules in the shop, the manager wishes to try them out in simulations.

Rule parameters

The rules are "parameterized". For example, one rule limits use of "fast checkout lanes" to customers who are purchasing less than some specified maximum number of items. Naturally, this "fast checkout limit" is a parameter that can be adjusted. The manager wants to run many simulations with different settings for the various parameters. In this way, it is hoped that it will be possible to identify which parameters are more critical and to establish some practical operating rules.

Program to simulate working for different sets of parameters

The program, that is to be developed for the manager, is to start by accepting a set of inputs that define a particular set of control parameters for the manager's rules. It is then to simulate one day of operation of the supermarket subject to these rules. The simulation will involve some "randomized" customer traffic. During the simulation, various statistics will be gathered that characterize the distribution of queuing times for the customers and the total amount of open-time and idle time for the checkouts. At the end of a simulation run, these data are to be displayed. The manager may then wish to initiate another "randomized" run, or may wish to repeat the run using the same "randomized" pattern for customer traffic but with a changed set of parameters for the rules.

Some details:

Time period simulated

The supermarket opens its doors at 8 a.m.. Once opened, the doors can be pictured as delivering another batch of frantic shoppers every minute. This continues until the manager closes the doors, which will happen at the first check time after the supermarket's nominal closing time of 7 p.m.. All customers then in the supermarket are allowed to complete their shopping, and must queue to pay for their purchases before leaving. During this closing period, the manager checks the state of the shop every 5 minutes and closes any checkouts that have become idle. The supermarket finally closes when all customers have been served. The closing time should be displayed along with the statistics that have been gathered to characterize the simulation run.

Checkout numbers

The supermarket has a maximum of 40 checkout lanes, but the maximum number to be used in any run of the simulation is one of the parameters that the manager wishes to vary. The checkouts may be "fast" or "standard" (as noted earlier, another of the parameters for a simulation run defines the limit on purchases permitted to users of fast checkout lanes). The supermarket always has at least one

standard checkout open; the manager wants the option of specifying a minimum of 1..3. There is no requirement that a "fast" checkout be always open, though again the manager wants to be able to specify a minimum number of fast checkouts (in the range 0..3). One of the other rules proposed by the manager specifies when to open fast checkouts if there are none already open.

Checkouts process a current customer, and have a "first-come-first-served" queue of customers attached. A checkout processes ten items per minute (this is not one of the parameters that the manager normally wishes to vary, but obviously the processing rate, as defined in the program, should be easy to change.) The minimum processing time at a checkout is one minute (customers have to find their change and dispute the bill even if they buy only one item). Once a checkout finishes processing a customer, that customer leaves the shop (and disappears from the simulation). Checkout operators record any time period that they are idle and, when reassigned to other duties, report their idle time to a supervisor who accumulates the total idle time.

Customers entering the supermarket have some planned number of purchases (see below for how this is determined). Customers have to wander the aisles finding the items that they require before they can join a queue at a checkout. It is a big store and the minimum time between entry and joining a checkout queue would be two minutes for a customer who doesn't purchase anything. Most customers spend a reasonable amount of time doing their shopping before they get to join a queue at a checkout. This shopping time is determined by the number of items that must be purchased and the customer's "shopping rate". Shopping rates vary; for these simulations the shopping rates should be distributed in the range 1..5 items per minute (again, this range is not a parameter that the manager wishes to change on different runs of the simulation, but the program should define the range in a way that it is easy to change if necessary).

Once they have completed their shopping, customers must choose the checkout where they wish to queue. You can divide customers into "fast" and "standard" categories. "Fast" customers are those who are purchasing a number of items less than or equal to the supermarket's "fast checkout limit" (together with that 1% of other customers who don't wish to obey such constraints). When choosing a checkout, a "fast customer" will, in order of preference, pick: a) an idle "fast" checkout, b) an idle "standard" checkout, or c) the checkout ("fast" or "standard") with the minimum current workload. Similarly, "standard customers" will pick a) an idle "standard" checkout, or b) the "standard" checkout with the minimum current workload.

The workload of a checkout can be taken as the sum of all the items in the trolleys of customers already in the queue at that checkout plus the number of items still remaining to be scanned for the current customer.

The manager is proposing a scheme whereby checks are made at regular intervals. This interval is one of the parameters for a simulation run; it should be something in the range 5..60 minutes.

When running a check on the state of the supermarket, the manager may need to close idle checkouts, or to open checkouts to meet the minimum requirements for fast and standard checkouts, or to open extra checkouts to avoid excessively long queues at checkouts. The suggested rules are:

Queues at checkouts

Customers

Choosing where to queue

Workloads at checkouts

Scheduling changes to checkouts

Rules for opening/closing checkouts

- Checkout closing rule:
If the number of shoppers in the aisles has decreased since the last check, then all idle checkouts should be closed (apart from those that need to be left open to meet minimum requirements).
- Minimum checkouts rules:
Open fast and standard checkouts as needed to make up the specified minimum requirements.
If the total number of customers in the supermarket (shoppers and queuers) exceeds 20, there must be at least one fast checkout open.

- Extra checkouts rules:

If either the number of customers queuing, or the number of customers shopping has increased since the last check, then the following rules for opening extra checkouts should be applied:

- a) If the average queue length at fast checkouts exceeds a minimum (specified as an input for the simulation run), then one additional fast checkout should be opened (but not if this would cause the total number of checkouts currently open to exceed the overall limit).
- b) A number of additional standard checkouts may have to be opened (subject to limits on the overall number of checkouts allowed to be open). Checkouts should be opened until the average queuing time at standard checkouts falls below a maximum limit entered as an input parameter for the simulation. (The queuing times can be estimated from the workloads at the checkouts and the known processing rate of checkouts.)

In the real world, when a new checkout opens, customers at the tails of existing queues move to the queue forming at the newly opened checkout if this would result in their being served more quickly. For simplicity, the simulation will omit this detail.

Customer traffic

The rate of arrival of customers is far from uniform and cannot be simulated by random drawing from a uniform distribution. Instead, it must be "scripted". The pattern of arrivals averaged for a number of days serves as the basis for this script. Typically, there is fairly brisk traffic just after opening with people purchasing items while on their way to work. After a lull traffic builds to a peak around 10.30 a.m. and then slackens off. There is another brisk period around lunch time, a quiet afternoon, and a final busy period between 5.30 p.m. and 6.30 p.m.. The averaging of several days records has already been done; the results are in a file in the form of a definition of an initialized array Arrivals[]:

```
static int Arrivals[] = {  
    // Arrivals at each 1 minute interval starting 8am  
    12, 6, 2, 0, 0, 1, 2, 2, 3, 4, 3, 0, 2, 0, 0,  
    ...  
};
```

This has entries giving the average number of people entering in each one minute time interval. Thus, in the example, 12 customers entered between 8.00 and 8.01 am etc. These values are to be used to provide "randomized" rates of customer arrival. If for time period i , `Arrivals[i]` is n , then make the number of customers entering in that period a random number in the range $0..2n-1$.

Similarly, the number of purchases made by customers is far from uniform. In fact, the distribution is pretty close to exponential. Most customers buy relatively few items, but a few individuals do end up filling several trolleys with three hundred or more items. The number of items purchased by customers can be approximated by taking a random number from an exponential distribution with a given mean value.

It has been noted that the mean values for these distributions are time dependent. Early customers, and those buying items for their lunch, require relatively few items (<10); so at these times the means for the exponential distributions are low. The customers shopping around 10.30 a.m., and those shopping between 5.30 and 6.30 p.m., are typically purchasing the family groceries for a week and so the mean numbers of items at these times are high (100+).

Again, these patterns are accommodated through scripting. The file with the array `Arrivals[]` has a commensurate array `Purchases[]`:

```
static int Purchases[] = {
// Average number purchases of customer
4, 3, 4, 5, 3, 5, 7, 6, 5, 4, 3, 5, 2, 5, 4,
...
};
```

The entries in this array are the mean values for the number of purchases made by those customers entering in a particular one minute period. The number of purchases made by an individual customer should be generated as a random number taken from an exponential distribution with the given mean.

The manager wants the program to provide an informative statistical summary at the end of each run. This summary should include:

- the shop's closing time;
- the number of customers served;
- a histogram showing the number of items purchased by customers;
- histograms summarizing the shopping, queuing and total times spent by customers;
- details of checkout operations such as total operating and idle times of checkouts

There should also be a mechanism for displaying the state of the supermarket at each of the check times.

Specification

Implement the Supermarket program:

Purchase amounts

Reports for the boss

- 1 The program will start by prompting for the input parameters:
 - frequency of floor manager checks;
 - maximum number of checkouts;
 - minimum number of fast and standard checkouts;
 - purchase limit for use of fast checkout;
 - length of queues at fast checkout necessary that, if exceeded, will cause the manager to open an extra fast checkout;
 - queuing time at standard checkout that, if exceeded, will cause the manager to open an extra fast checkout;
- 2 The program is then to simulate activities in the shop from 8.00 a.m. until final closing time.
At intervals corresponding to the floor manager's checks, the program is to print a display of the state of the shop. This should include summaries of the total number of customers currently in the shop, those in queues, and details of the checkouts. Active checkouts should indicate their queue lengths. The number of idle checkouts should be stated.
Details of any changes to the number of open checkouts should also be printed.
- 3 When all customers have left, the final closing time should be printed along with the histograms of the statistics acquired. The histograms should include those showing number of purchases, total time spent in the shop, and queuing times. Details of total and idle time of checkouts should also be printed.

27.2 DESIGN

27.2.1 Design preliminaries

The simulation mechanism

Loop representing passage of time

The simulation mechanism is similar but not identical to that used in the `AirController/Aircraft` example. The core of the simulation is again going to be a loop. Each cycle of this loop will represent one (or more) minute(s) of simulated time. Things happen almost every minute (remember, another bunch of frantic customers comes through the door). Most activities will run for multiple minutes (two minute minimum shopping time, one minute minimum checkout time etc).

On each cycle do ...

Each cycle of the simulation loop should allow any object that needs to perform some processing to "run". This is again similar to the `AirController/Aircraft` example where all the `Aircraft` got a chance to `Move()` and update their positions. The Supermarket program could arrange to tell all objects to "run" for one minute (shoppers complete more of their purchasing, checkouts scan a few more items). However, as there are now going to be hundreds of objects it is better to use a slightly more efficient mechanism whereby objects suspend themselves for a period of time and only those that really need to run do so in any one cycle.

Priority queue used in simulation

There is a standard approach for simulations that makes use of a priority queue. Objects get put in this queue using "priorities" that represent the time at which they are going to be ready to switch to a new task (all times can be defined as integer values – minutes after opening time). For example, a customer who starts doing 15

minutes worth of shopping at 8.30 a.m. can be inserted into this queue with priority 45 (i.e. ready at 8.45 a.m.), a customer starting at the same time but with only 3 minutes worth of shopping will get entered with priority 33 (i.e. ready at 8.33).

The simulation loop doesn't have to advance time by single units. Instead, it pulls an item from the front of the priority queue. Simulated time can then be advanced to the time at which this next event is supposed to happen. So, if the front item in the queue is supposed to occur at 8.33 a.m., the simulated time is advanced to 8.33 a.m.. In this example, there will tend to be activities scheduled for every minute; but you often have examples where there are quiet times where nothing happens for a period. The priority queue mechanism lets a simulation jump these quiet periods.

When items are taken from the queue, they are given a chance to "run". Now "run" will typically mean finishing one operation and starting another, though for some objects it will just mean doing the same thing another time. This mechanism for running a simulation is efficient because "run" functions are only called when it is time for something important to happen. Thus, there no need to disturb every one of the hundred customers still actively shopping to ask if they are ready go to a checkout; instead, the customers are scheduled to be at the front of the priority queue when they are ready to move to checkouts.

Usually, the result of an object's "run" function will be some indication that the it wants to be put back in the priority queue at some later time (larger priority number), or that it needs to be transferred to some other queue, or that it is finished and can be removed from the simulation.

There will be something in the program that handles this main loop. But before we get too deep into those details we need to identify the objects. What are these things that get to run, and what do they want to do when they have a chance to run?

The objects

So, what are the objects? What they own? What they do?

Some are obvious. There are going to be "Customer" objects and "Checkout" objects. There will also be "Queue" objects, and "Histogram" objects. Now while all these will be important it should be clear that they aren't the ones that really define the overall working of the simulation.

Customer objects are going to be pretty passive. They will just hang around "shopping" until they eventually chose to move to a checkout queue. They will then hang around in a queue until they get "processed" by a Checkout after which they will report their shopping time, queuing time, etc and disappear. But something has got to create them. Something has to ask them for their statistics before they leave. Something has to keep count of them so that these data are available when the rules for opening/closing checkouts are used.

Checkout objects might be a little more active. They will add customers to their queues (if customers were to be permitted to switch queues, the checkouts would have to allow customers to be removed from the tails of their queues). They pull customers off their queues for processing. They report how busy they are. But

Obvious objects

Customers

Checkouts

they still don't organize much. There has to be some other thing that keeps track of the currently active checkouts.

Finding the objects The other objects, in this example the more important control objects, have still to be identified.

Underline the nouns? An approach sometimes suggested is to proceed by underlining the nouns in the problem description and program specification. After all, nouns represent objects. You will get a lengthy and rather strange list: supermarket, door, minute, time, shopping, purchases, manger, employee, statistics, aisles, You drop those that you feel confident don't add much; so minute, shopping, aisles can all go immediately. The remaining nouns are considered in more detail.

Purchases? How about purchases? Not an object. The only thing we need to know about a customer's purchases is the number. Purchases can be a data member of the Customer class. The class had better provide an access function to let other objects get this number when needed.

Time? Time? The simulation has to represent time. But basically it is just an integer counter (minutes after opening time or maybe minutes after midnight). Something owns the system's timer; something updates it (by advancing this timer when items are taken from the priority queue). Many objects will need access to the time value. But the time is not an object.

Manager? Manager? This seems a better candidate. Something needs to hold the parameters used in the checkout opening/closing rules. Something needs to execute the code embodying those rules. There would only be one instance of this class. Despite that, it seems plausible. It offers a place to group some related data and behaviours.

Employee? Employees? No. The simulation doesn't really need to represent them. Their activities when they are not operating checkouts are irrelevant to the simulation. There is no need to simulate both employee and checkout. In the simulation the checkouts embody all the intelligence needed to perform their work.

Avoid over faithful models of real world You can do design by underlining nouns, and then choosing behaviours for the objects that these nouns represent. It doesn't always work well. Often it results in over faithfully modelling the real world. You tend to end up with "Employee" objects and Checkout objects, and a scheme for assigning "Employee" objects to run Checkout objects. This just adds unnecessary complication to the program.

Alternative approach for finding objects: Scenarios Use of scenarios is an alternative way to find objects. Scenarios, and the patterns of object interactions that they reveal, were used in Chapter 22. There we already had a good idea as to the classes and needed just to flesh out our understanding of their behaviours and interactions. However, we can use scenarios at an earlier stage where we are still trying to identify the classes. Sometimes, a scenario will have a "something" object that asks a Customer object to provide some data. We can try and identify these "somethings" as we go along.

Starting points for scenarios The example programs in Chapter 22 were driven by user entered commands. So, we could proceed by working out "What happens when the user requests action X?" and following the interactions between the objects that were involved in satisfying the users request.

This program isn't command driven. It grabs some initial input, then runs itself. So we can't start by following the effect of each command.

Where else might we start?

The creation and destruction of objects are important events in any program. We have already identified the need for `Customer` objects and `Checkout` objects. So, a possible starting point is looking at how these get created and destroyed.

Something has to create `Customers`, tell them how much they want to buy, and then pass them to some other something that looks after them until they leave the shop. Note, we have already found the need for two (different kinds of) somethings. The first something knows about those tables of arrivals and purchase amounts. The second something owns data like a list of current customers.

When `Customer` objects leave, they have to report their statistics. How? To what? A `Customer` object has several pieces of data to report – its queuing time, its number of purchases, its shopping time, etc. These data are used to update different `Histogram` objects. It would be very inconvenient if every `Customer` needed to know about each of the different `Histogram` objects (too many pointers). Instead, the gathering of statistics would be better handled by some object that knew about `Customer` objects and also knew about (possibly owned?) the different `Histogram` objects.

`Checkout` objects also get created and destroyed. The "Manager" chooses when creation and destruction occurs and may do the operations or may ask some other object to do the actual creation (destruction). Something has to keep track of the checkout objects. When a checkout is destroyed, it has to report its idle time to something.

Scenarios relating to these creation and destruction steps of known objects will help identify other objects that are needed. Subsequently, we can examine scenarios related to the main simulation loop. "Active objects" are going to get taken from the front of the priority queue and told to "run". We will have to see what happens when objects of different types "run".

27.2.2 Scenarios: identifying objects, their classes, their responsibilities, their data

Creation and destruction scenarios

Creating customers

A supermarket without customers is uninteresting, so we might as well start by looking at how `Customer` objects come into the system.

Figure 27.1 provides a first idea. We could have a "Door" object that creates the `Customers`, providing them with the information they need (like their number of purchases). The `Customer` objects will have to complete any initialization (e.g. choose a rate at which to shop), work out how long their shopping will take, and then they will have to add themselves to some collection maintained by a "Shop" object. To do that, they will have to be given a pointer to the shop object (customers need to know which shop they are in). The `Door` object can provide this pointer (so long as it knows which shop it is attached to).

Scenarios for the creation and destruction of objects

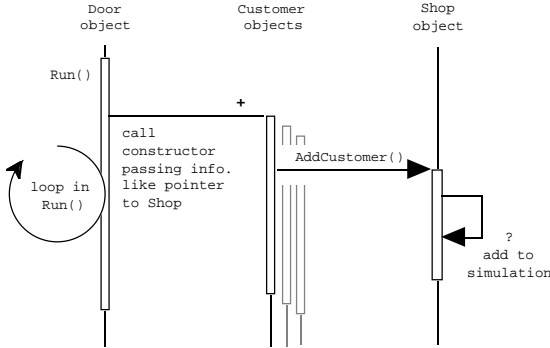


Figure 27.1 Scenario for creation of Customer objects.

Customers are supposed to come into the shop every minute. The main simulation loop is going to get "active objects" (like the `Door`) to "run" each minute (simulated time). Consequently, the activities shown in Figure 27.1 represent the "run" behaviour of class `Door`. It can use the table of arrival rates and average purchases to choose how many `Customer` objects to create on each call to its `Run()` function.

This initial rough scenario for creating a `Customer` object cannot clarify exactly what the `Shop` object must do when it "adds a customer". Obviously it could have a list of some form that holds all customers; but it might keep separate lists of those shopping and those queuing. Some activity by the customer (switch from shopping to queuing) has also got to be scheduled into the simulation. The `Shop` object might be able to organize getting the customer into the simulation's priority queue.

Results:

We appear to need:

- a `Door` object.
It uses (owns?) those arrays (with details of when customers arrive and how much they want to purchase), and a pointer to a `Shop` object.
In its `Run()` member function, called from the main simulation loop, it creates customers.
- Constructor
The constructor function gets given a pointer to a `Shop`, and a value for the number of purchases. The function is to pick an actual number of purchases (exponentially distributed with given number as mean, minimum of 1 item), and select a shopping rate. The `Customer` better record starting time so that later it is possible to calculate things like total time spent in the shop. (How does it get the current time? Unknown, maybe the time is a global, or maybe

the Customer object could ask the Shop. Decide later.) These data get stored in private data members.

Constructor should invoke "AddCustomer" member function of the Shop object.

- a Shop object.
This keeps track of all customers, and gets them involved in the main simulation.

Removing customers from simulation

Customers leave the simulation when they finish being processed by a Checkout. The Checkout object would be executing its Run() member function when it gets to finish with a current customer; it can probably just delete the Customer object. We can arrange that the destructor function for class Customer notify the Shop object. The Shop object can update its count of customers present. Somehow, the Customer has to report the total time it spent in the shop. Probably the report should be made to the Shop object; it can log these times and use the data to update the Histogram objects that will be used to display the data.

Figure 27.2 illustrates the interactions that appear to be needed.

Results:

Responsibilities of Shop object becoming clearer. It is going to gather the statistics needed by the various Histogram s. It probably owns these Histogram objects.

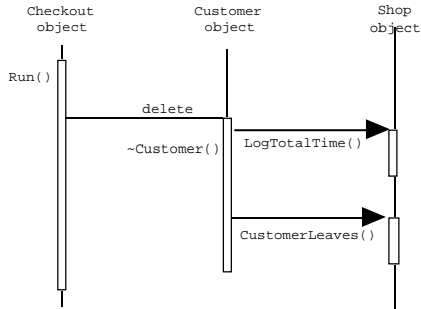


Figure 27.2 Scenario for deletion of Customer objects.

Creating and Destroying Checkout objects

Customer objects will be getting created and deleted every minute. Changes to the Checkouts are less common. Checkout objects are only added or removed when the floor manager is performing one of his/her regular checks (the problem specification suggested that these checks would be at intervals of between 5 and 60 minutes). Checkouts are added/removed in accord with the rules given earlier.

There seems to be a definite roll for a "Manager" object. It will get to "run" at regular intervals. Its "run" function will apply the rules for adding and removing checkouts.

The Manager object will need to be able to get at various data owned by the Shop object, like the number of customers present. Maybe class Manager should be a friend of class Shop, otherwise class Shop is going to have to provide a large set of access functions for use by the Manager object.

Figure 27.3 illustrates ideas for scenarios involving the creation and deletion of checkout objects. These operations will occur in (some function called) from Manager::Run(). The overall processing in Manager::Run() will start with some interactions between the Manager object and the Shop object. These will give the Manager the information needed to run the rules for opening/closing checkouts.

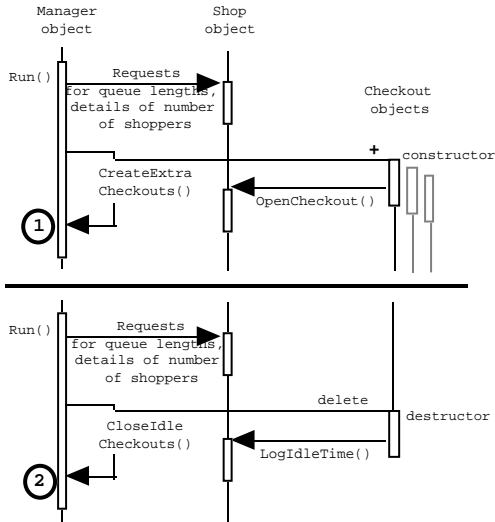


Figure 27.3 Scenarios for creation and deletion of Checkout objects.

The Shop might be involved in further interactions with existing checkouts to find their queue lengths etc; we can ignore these secondary interactions until later.

Pane 1 of Figure 27.3 illustrates an idea as to events when the shop is getting busier. The `Manager` will create additional checkouts. As part of their "constructor" operations, these checkout objects can register with the `Shop` (it will need to put them into lists and may have to perform other operations).

Pane 2 of Figure 27.3 shows the other case where there are idle checkouts that should be deleted. The `Manager` object will have to get access to a list of idle checkouts that would presumably be kept by the `Shop` object. It could then remove one or more of these, deleting the `Checkout` objects once they had been removed.

The destructor of the `Checkout` object would have to pass information to the `Shop` object so that it could maintain details of idle times and so forth.

Note the use of destructors in this example, and the preceding case involving `Customer` objects. Normally, we've used destructors only for tidying up operations. But sometimes, as here, a destructor should involve sending a notification to another object.

Expanded role for destructors

Other acts of creation and destruction

The other objects appear all to be long lived. The simulation could probably start with the `main()` function creating the principal object which appears to be the `Shop`:

```
int main()
{
    // Some stuff to initialize random number generator

    Shop  aSuperMart;
    aSuperMart.Setup();
    aSuperMart.Run();
    return 0;
}
```

The `Shop` object could create the `Door`, and the `Manager` objects, either as part of the work of its constructor or as some separate `Setup()` step. Things like the `PriorityQueue` and `Histogram` objects could be ordinary data members of class `Shop` and so would not need separate creation steps.

The `Shop`, `Door`, and `Manager` objects can remain in existence for the duration of the program.

Scenarios related to the main `Run()` loop in the simulation

The main `Shop::Run()` loop working with the priority queue will have to be something like the following:

```
while Priority Queue is not empty
    remove first thing from priority queue
    move time forward to time at which this thing is
        supposed to run
```

```
let the thing run

check status of thing
if terminated
    delete it
if idle
    ignore it
if running
    reenter into priority queue

report final statistics
```

Need for a class hierarchy

The priority queue contains anything that wants a chance to "run", and so it includes the `Door` object, a `Manager` object, some `Checkout` objects and some `Customer` objects. But as far as this part of the simulation is concerned, these are all just objects that can "run", can specify their "ready time", and can be asked their status (terminated, idle, running).

A simulation using the priority queue mechanism depends on our being able to treat the different objects in the priority queue as if they were similar. Thus here we are required to employ a class hierarchy. We need a general abstraction: class "Activity".

class Activity

Class `Activity` is an abstraction that describes an `Activity` object as something that can:

- `Run()`
An `Activity`'s `Run()` function will complete work on a current task and select the next task. This will be a pure virtual function. Specialized subclasses of class `Activity` define their own task agendas.
- `Status()`
An `Activity` can report its status (value will be an enumerated type). Its status is either "running" (the `Activity` is in, or should be added to, the main priority queue used by the simulation), or "idle" (the `Activity` is on some "idle" list, some other object may invoke a specialized member function that change the `Activity`'s status), or "terminated" (the main simulation loop should get rid of those `Activity` objects that report they have terminated).
- `Ready_At()`
An `Activity` can report when it is next going to be ready; the result will be in simulation time units (in this example, these units will be minute times during the day).

The current hierarchy of `Activity` classes is illustrated in Figure 27.4.

The scenarios shown in Figures 27.1 and 27.3 related to `Run()` member functions of two of the classes. The other activities triggered from this main loop will involve actions by `Customer` objects and `Checkout` objects.

A `Customer` object should be initially scheduled so that it "runs" when it completes its shopping phase. At this time, it should choose the `Checkout` where it wants to queue.

Probably, a `Customer` object should ask the `Shop` to place it on the shortest suitable queue; see Figure 27.5. This would avoid having the `Customer` objects

interacting directly with the Checkout objects. Once it is in a Checkout queue, the Customer object can remove itself from the main simulation (it has nothing more to do); to achieve this, it just has to set its state to "idle".

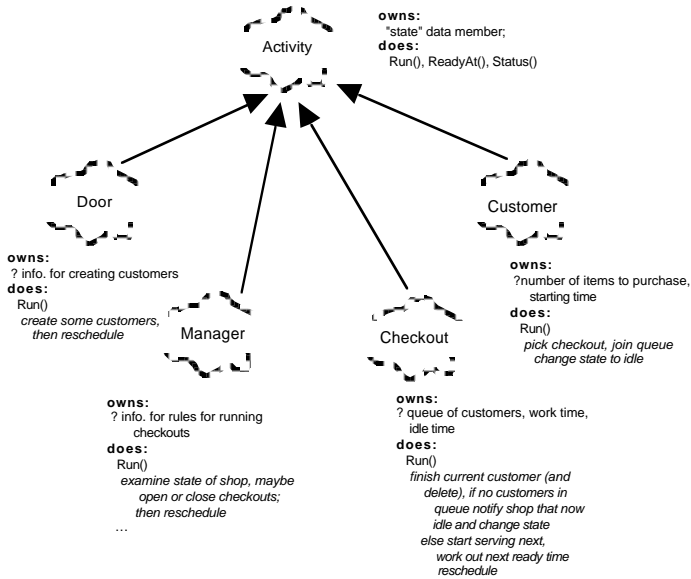


Figure 27.4 A hierarchy of "Activity" classes.

Checkout objects are scheduled to "run" at the time they finish processing a current customer. At this time they can delete the customer (scenario in Figure 27.2). They then have to check their queues. If a Checkout's queue is empty, it should set its state to "idle" and inform the Shop (this needs to keep track of idle checkouts); once it is idle, the Checkout drops out of the simulation but can be reinserted if the Shop object gives the Checkout more work and gets it to reschedule itself. A Checkout should note the time that it becomes idle so that it can report its idle time. Of course, Checkouts will usually find other Customers queuing; the first Customer should be removed from the queue for processing. The Checkout can calculate its next "ready at" time from the amount of the Customer's purchases and will continue in the "running" state.

Checkout objects and Customer objects drop out of the simulation loop by becoming "idle"; some other interaction causes them to be deleted. The simulation continues until the queue is empty. The Customer and Checkout objects get removed as they become idle, but what about the Door and the Manager.

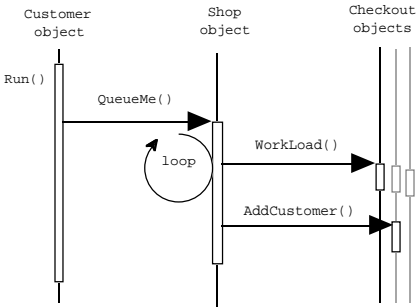


Figure 27.5 Interactions with Customer::Run().

These two normally reschedule themselves as running; the Door object setting its next "ready at" time for one minute later, the Manager selecting a time based on the frequency of checks. The Manager is responsible for getting the Shop to close its Door sometime after 7 p.m.. This is probably the basis for getting the Door removed from the simulation. If it has been "closed", instead of rescheduling itself the Door should report that it has "terminated". The simulation loop will then get rid of it.

Similarly, the Manager object should "terminate" once all its work has been done. This will be when the door has been closed and the last customer has been finished. The termination of the Manager will leave the priority queue empty and the simulation will stop.

When the main simulation loop ends, the Shop can printout the statistics that it has gathered.

27.2.3 Filling out the definitions of the classes for a partial implementation

The analysis of the problem is still incomplete. We haven't yet really covered how the statistics are gathered or how the histograms are displayed; nor have the rules for opening and closing checkouts been considered in any detail.

Despite that, it would be reasonable to implement a simplified version of the overall simulation at this stage. Such a partial implementation makes it possible to fully test some of the classes, e.g. class Door, and clarify other aspects such as the use of the priority queue in the simulation.

Of course, some of the code created for a partial implementation is usually irrelevant to the final program. You may have to have extra functions that do, in a simplified way, some work that will later be the responsibility of a different object. You may assign responsibility for some data to one class, only to find later that you have to move those data elsewhere. Some code is thrown away. Some time is wasted.

Disadvantages of partial, prototype implementations

Although there are disadvantages associated with partial "prototype" implementations they do have benefits. If you "analyse a little, design a little, and implement a little", you get a better understanding of the problem. Besides, you may find it boring to have to work out everything on paper in advance; sometimes it is "fun" to dive in and hack a little.

The simplified version of the program could omit the `Checkouts`. Customer objects would just change their state to "terminated" when they finish shopping. The `Manager`'s role can be restricted to arranging for regular reports to be produced; these would show the number of customers present. Such changes eliminate most of the complexities but still allow leave enough to allow testing of the basic simulation mechanisms.

The classes needed in the reduced version are: `Door`, `Manager`, `Customer` (all of which are specialized subclasses of class `Activity`) and `Shop`.

In addition, the `Shop` will use an instance of class `PriorityQ` and possibly some instances of class `List`. These will simply be the standard classes from Chapter 24. The priority queue needs one change from the example given there; the maximum size of the queue should now be 1000 rather than 50 (change the constant `k_PQ_SIZE` in the class header). The `List` class should not need any changes.

The abstract class, class `Activity`, should be defined first:

```
class Activity {
public:
    enum State { eTERMINATED, eIDLE, eRUNNING };
    Activity(State s = eRUNNING) : fState(s) { }
    virtual ~Activity() {}
    virtual void Run() = 0;
    virtual long Ready_At() = 0;
    State Status() { return fState; }
protected:
    State fState;
};
```

It is simply an interface. `Activity` objects are things that "run" and report when they will next be ready; how they do this is left for definition in subclasses. Functions `Run()` and `Ready_At()` are defined as pure virtual functions; the subclasses must provide the implementations.

The abstract class can however define how the `Status()` function works as this function is simply meant to give read access to the `fState` variable. The constructor sets the `fState` data member; the default is that an `Activity` is "running". Data member `fState` is protected; this allows it to be accessed and changed within the functions defined for subclasses.

Note the virtual destructor. Objects are going to be accessed via `Activity*` pointers; so we will get code like:

```
Activity* a = (Activity*) fSim.First();
...
switch (a->Status()) {
case Activity::eTERMINATED:
    delete a;
    break;
```

Advantages of partial, prototype implementations

Classes to be defined

Reused classes

Class Activity

Pure virtual functions to be defined in subclasses

Definitions for other member functions

virtual destructor

The `delete` operation has to cause the appropriate destructor to be executed. If you did not specify `virtual ~Activity()` (e.g. you either didn't include a destructor or you declared it simply as `~Activity()`) the compiler would assume that it was sufficient either to do nothing special when `Activity` objects were deleted, or to generate code that included a call to `Activity::~~Activity()`. However, because `virtual` has been specified correctly, the compiler know to put in the extra code that uses the table lookup mechanism (explained in Chapter 26) and so at run-time the program determines whether a call should be made to `Customer::~~Customer()`, or `Manager::~~Manager()`, or `Door::~~Door()`.

If a subclass has no work to be done when its instances are deleted, it need not define a destructor of its own. (A definition has to be given for `Activity::~~Activity()`; it is just an empty function, `{ }`, because an `Activity` has nothing of its own to do.)

Class `Door` must provide implementations for `Run()` and `Ready_At()`. It will obviously have its own constructor but it isn't obvious that a `Door` has to do anything when it is deleted so there may not be a `Door::~~Door()` destructor (the "empty" `Activity::~~Activity()` destructor function will get used instead). Apart from `Run()`, `Status()`, and `Ready_At()`, the only other thing that a `Door` might be asked to do is `Close()`.

The specification implies that a file already exists with definitions of the arrays of arrival times and purchase amounts. These are already defined as "filescope" variables. If you had the choice, it would be better to have these arrays as static data members of class `Door`. However, given the specification requirements, the file with the arrays should just be `#included` into the file with the code file `Door.cp`. The `Door` functions can just use the arrays even though they can't strictly be "owned" by class `Door`.

The other data that a `Door` needs include: a pointer to the `Shop`, a flag indicating whether the `Door` is "open", a counter that lets it step through the successive elements in the `Arrivals[]` and `Purchases[]` arrays, and a long integer that holds the "time" at which the `Door` is next ready to run.

The class declaration should be along the following lines:

```
class Door : public Activity {
public:
    Door(Shop* s);
    void Close();
    virtual void Run();
    virtual long Ready_At() { return fready; }
private:
    Shop *fs; // Link to Shop object
    long fready; // Ready time
    short fopen; // Open/closed status
    short fndx; // Next entries to use from Arrivals[], Purchases[]
};
```

The initial part of the declaration:

```
class Door : public Activity {
```

essentially states that "A Door is a kind of Activity". Having seen this, the C++ compiler knows that it should accept a programmer using a Door wherever an Activity has been specified.

The behaviours for the functions are:

```
Constructor
Set pointer to Shop; initialize fready with time from Shop object;
set fopen to true; and fndx to 0.
(The Shop object will create the Door and can therefore arrange
to insert it into the priority queue used by the simulation.)

Run()
If fopen is false, set fState (inherited from Activity) to
"terminated".
Otherwise, pick number of arriving customers (use entry in
Arrivals[], get random number based on that value as default).
Loop creating Customer objects.

Close()
Set fopen to false.
```

This version of class Door should not need any further elaboration for the final program.

Class Customer is again a specialized Activity that provides implementations for Run() and Ready_At(). The constructor will involve a Customer object interacting with the Shop so that the counts of Customers can be kept and Customer objects can be incorporated into the simulation mechanism.

The destructor will also involve interactions with the Shop; as they leave, Customers are supposed to log details of their total service times etc.

The only other thing that a Customer object might be asked is to report the number of purchases it has made. This information will be needed by the Checkout objects once that class has been implemented.

A Customer object needs a pointer to the Shop, a long to record the time that it is next ready to run, a long for the number of items, and two additional long integers to record the time of entry and time that it started queuing at a checkout.

A declaration for the class is:

```
class Customer : public Activity{
public:
    Customer(Shop* s, short mean);
    ~Customer();
    virtual void Run();
    virtual long Ready_At() { return fready; }
    long ItemsPurchased() { return fitems; }
private:
    Shop *fs; // Link to shop
    long fready; // Ready time
    long fstarttime; // Other time data
```

class Customer

```
long fstartqueue;
long fitems; // # purchases
};
```

The behaviours for the functions are:

```
Constructor
Set pointer to Shop. The second argument to the constructor is to
be used to pick the number of items to purchase; the value is to be
an integer from an exponential distribution with the given mean
value. Pick fitems accordingly.
Record the start time (getting the current time from the Shop
object).
Choose a shopping rate (random in range 1...5) and use this and
the value of fitems to calculate the time the Customer will be
ready to queue.
Tell Shop to "Add Customer".
```

```
Run()
In this limited version, just set fState to terminated.
```

```
Destructor
Get Shop to log total time; then tell shop that this Customer is
leaving.
```

The final program will require changes to Run(). The Customer object should get the Shop to transfer it to the queue at one of the existing Checkouts and set its own state to "idle" rather than "terminated".

class Manager

Class Manager is the third of the specialized subclasses of class Activity. Once again, it has to provide implementations for Run() and Ready_At(). Its constructor might be a good place to locate the code that interacts with the user to get the parameters that control the simulation. It does not appear to need to take any special actions on deletion so may not need a specialized destructor.

So far, it seems that the Manager will only be asked to Run() (and say when it is ready) so it may not need any additional functions in its public interface. Its Run() behaviour will eventually become quite complex (it has to deal with the rules for opening and closing checkouts). Consequently, the final version may have several private member functions.

The Manager object seems a good place to store most of the control parameters like the frequency of floor checks, the minimum numbers of fast and standard checkouts and the control values that trigger the opening of extra checkouts. Other information required would include the number of customers shopping and queuing at the last check time (the Shop object can be asked for the current numbers). The only parameter that doesn't seem to belong solely to the Manager is the constraint on the number of purchases allowed to users of fast checkouts. This gets used when picking queues for Customers and so might instead belong to Shop.

A declaration for the class is:

```
class Manager : public Activity{
public:
    Manager(Shop* s);
    virtual void Run();
    virtual long Ready_At() { return fready; }
private:
    // Will eventually need some extra functions that
    // select when to open/close checkouts

    Shop      *fs;           // Link to shop
    long       ft;           // time between checks
    long       fready;       // Ready time
    short      fmaxcheckouts;// Control parameters
    short      fminfast;
    short      fminstandard;
    short      fqlen;
    short      fqtime;
    short      fQueuingLast;
    short      fShoppingLast;
};
```

(All three classes uses the same style of implementation of the function Ready_At(); this suggest that it could be defined as a default in the Activity class itself.)

The behaviours for the functions are:

- Constructor
Set pointer to Shop; initialize fready with time from Shop object; set fopen to true; and records from "last check" to -1.
(The Shop object will create the Manager and can therefore arrange to insert it into the priority queue used by the simulation.)
- Prompt the user to input the values of the control parameters for the simulation.
- Run()
If the time is after the shop's closing time, get the shop to make certain that the door is closed, check the number of customers still present and if zero change own status to "terminated".
Otherwise get the shop to print a status display and reschedule the Manager to run again after another ft minutes.

The final program will require changes to Run(). The Manger object will have to interact with the Shop to get Checkouts opened or closed.

Class Shop is shaping up to be the most elaborate component in the simulation. *class Shop*
A declaration with the members identified so far is:

```
class Shop {
public:
    Shop();
```

```
// Organizing simulation
void Setup();
void Run();
// Changing customers (and checkouts)
void AddCustomer(Customer* c);
void CustomerLeaves();
// Display of state
void DisplayState();

// keeping statistics
void LogShopTime(int); // Customer times
void LogQueueTime(int);
void LogTotalTime(int);
void LogNumberPurchases(int);

void LogOpenTime(int); // Checkout times
void LogIdleTime(int);

// Time details
long Time();

// closing
void CloseDoor();

// To save having lots of access functions
friend class Manager;
private:
    PriorityQ fSim;

    Manager *fManager;
    Door *fDoor;

    long fTime;

    short fCustomersPresent;
    short fCustomersQueuing;
    short ffastlanemax;

    long fidle; // Idle time of checkouts
    long fworktime; // Total open time of checkouts

    ...
};
```

Use of a friend relation

The final program will have several additional data members (lists to keep records of Checkout objects etc) and some additional member functions.

It seems worthwhile making class Manger a friend of class Shop. The Manager needs access to data such as the number of customers present. Rather than provide a series of access functions for all the various data elements that might be needed, we can use a friend relation. (It isn't just a program "hack"; it is appropriate that a Manager know all details of the Shop.)

The functions used to record statistical data can start to be defined, even though at this stage they may have empty implementations. Similarly, we can start to have

data members that will be used to store the statistics, e.g. an integer `fidle` to store the total time that there were checkouts that were open but idle.
The behaviours for the functions are:

- Constructor
Initialize all data members (`fidle = 0`; `fTime = kSTARTTIME`; etc).
- Setup()
Create `Manager` and `Door` objects, insert them into the `PriorityQueue fSim`.
- Run()
Implementation of loop shown earlier in which `Activity` objects get removed from the priority queue and given a chance to run.
- AddCustomer()
Update count of customers present, and insert customer into priority queue.
- CustomerLeaves()
Decrement counter.
- DisplayState()
Show time and number of customers (might need an auxiliary private member function to "pretty print" time).
Full implementation will need display details of active and idle checkouts as well.
- Log functions
All empty "stubs" in this version.
- Time()
Access function allowing instances of other classes to have read access to `fTime`.
- CloseDoor()
If `fDoor` pointer not `NULL`, tell door to close then set `fDoor` to `NULL`.

Member functions of
class Shop

There doesn't appear to be any need for a destructor. Output of the time in a "pretty" format (e.g. 9.07 a.m., 3.56 p.m.) might be handled in some extra private `PrintTime()` member function.

27.3 A PARTIAL IMPLEMENTATION

main() and auxiliary functions

Most of the code for `main()` was given earlier (just before introduction of the "activity" class hierarchy). There is one extra feature. The manager using the program wants to be able to run simulations of slightly different patterns of customer arrivals, and simulations using an identical pattern of arrivals but different parameters.

Seeding the random number generator This can be achieved by allowing the user to "seed" the pseudo random number generator. Since "pseudo random numbers" are generated algorithmically, different runs using the same seed will get identical sequences of numbers. Use of a different seed results in a different number sequence.

```
int main(int,char**)
{

    long   aseed;
    cout << "Enter a positive integer to seed the random "
           "number generator\n";
    cin >> aseed;

    srand(aseed);

    Shop   aSuperMart;
    ...    // as shown above
```

Random numbers
from an exponential
distribution

The random number generator in the standard maths library produces numbers that are uniformly distributed. This program also requires some numbers that are taken from an exponential distribution with a defined mean (such a distribution has large numbers of small values, and a tail with large values). Most versions of the maths library don't include this version of the number generator. The required function, `erand()`, uses the normal random number generator `rand()` and a few mathematical conversions to produce random numbers with the required characteristics:

```
erand()    int erand(int mean)
           {
               return int(-mean *log(
                           double(SHRT_MAX-rand() + 1)/SHRT_MAX )
                           + 0.5);
           }
```

This function can be defined in the file with the code for class `Customer`; the header files `math.h`, `stdlib.h`, and `limits.h` must be `#included`. (This implementation of `erand()` assumes that `rand()` generates values in the range `0...SHRT_MAX`. Your implementation may use a different range in its random number generator, so you may need to modify this definition of `erand()`. The range used by `rand()` is supposed to be defined by constants in the `limits.h` header file but many systems do not comply.)

Module structure

Like the examples in Chapter 22, this program should be built from many separate files (a header file and an implementation file for essentially every class). Consequently, you have to sort out "header dependencies".

A possible arrangement of files, and most of the header dependencies, is shown in Figure 27.6. (Standard header files provided by the IDE are not shown.)

The main program defines an instance of class `Shop` and therefore must `#include` the `Shop.h` header. The class definition in `Shop.h` will specify that a `Shop` object contains a `PriorityQ` as a data member. Consequently, the header `Shop.h` can only be handled if the header declaring the `PriorityQ` has been read; hence the dependency from `Shop.h` to `pq.h`.

The declaration of class `Shop` also mentions `Manager`, `Customer`, and `Door` but these only appear as pointer types. It isn't essential to read the declarations of these classes to compile `Shop.h`; but these names must be defined as class names. So file `Shop.h` starts with simple type declarations:

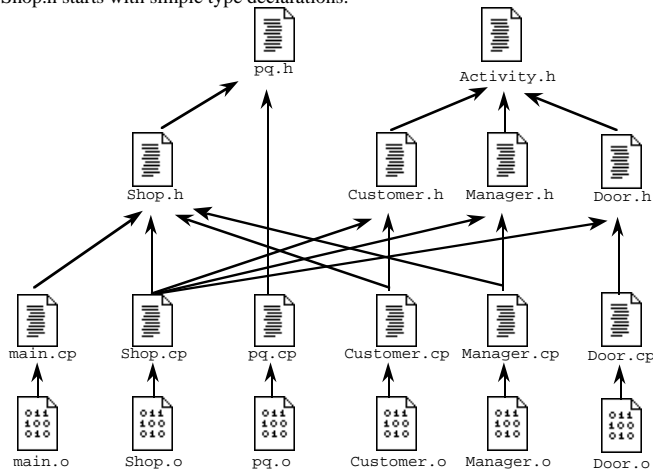


Figure 27.6 Module structure and header dependencies for partial implementation of "Supermarket" example.

```
#ifndef __SHOP__
#define __SHOP__

#include "pq.h"

#endif
```

Shop.h header file

Header file dependencies

```
const int kSTARTTIME = 480; // 8am, in minutes
const int kCLOSETIME = 540; // 9am, in minutes

class Door;
class Manager;
class Customer;

class Shop {
public:
    Shop();
    ...
    void AddCustomer(Customer* c);
    ...
    friend class Manager;
private:
    PriorityQ fSim;
    ...

    Manager *fManager;
    Door *fDoor;
    ...
};

#endif
```

Protect against multiple #includes

All header files must be bracketed with `#ifndef __XX__ ... #endif` conditional compilation directives to avoid problems from multiple inclusion. (You can see from Figure 27.6, that the file "Activity.h" will in effect be `#included` three times by `Shop.cp`.) The italicised lines in the file listing above illustrate these directives.

Reducing compile times

By now you must have noticed that the majority of time used by your compiler is devoted to reading header files (both Symantec and Borland IDE's have compilation-time displays that show what the compiler is working on, just watch). Having `#ifdef ... #endif` compiler directives in the files eliminates errors due to multiple `#includes` but the compiler may still have to read the same file many times. The example above illustrates a technique that can slightly reduce compile times; you will note that the file "pq.h" will only get opened and read if it has not already been read.

The constants kSTARTTIME ...

The two constants define the start and end times for the partial simulation; these values are need in `Manger.cp` and `Shop.cp`. The values will be changed for the full implementation.

The implementation file `Shop.cp` contains calls to member functions of the various "activity classes"; consequently, `Shop.cp` must `#include` their headers (so that the compiler can check the correctness of the calls). These dependencies are shown in Figure 27.6 by the links from `Shop.cp` to `Customer.h` etc.

The three "activity classes" all need to `#include` `Activity.h` into their own header files. As they all have `Shop*` data members, their headers will need a declaration of the form `class Shop;`. Classes `Customer` and `Manager` both use features of class `Shop`, so their implementation files have to `#include` the `Shop.h` header.

The final implementation will add classes `List`, `Histogram`, and `Checkout`. Their files also have to be incorporated into the header dependency scheme. Class

List and Histogram have to be handled in the same way as class PriorityQ; class Checkout will be the same as class Manager.

class Shop

The constructor for class Shop is trivial, just a few statements to zero out counters and set fTime to kSTARTTIME. The Setup() function creates the collaborating objects. Both need to have a Shop* pointer argument for their constructors that is supposed to identify the Shop object with which they work; hence the this arguments.

```
void Shop::Setup()
{
    fDoor = new Door(this);
    fManager = new Manager(this);
    fSim.Insert(fManager, fManager->Ready_At());
    fSim.Insert(fDoor, fDoor->Ready_At());
}
```

Once created, the Manager and Door objects get inserted into the PriorityQ fSim using their "ready at" times as their priorities.

The main simulation loop is defined by Shop::Run():

```
void Shop::Run()
{
    while(!fSim.Empty()) {
        Activity* a = (Activity*) fSim.First();
        fTime = a->Ready_At();
        a->Run();
        switch (a->Status()) {
        case Activity::eTERMINATED:
            delete a;
            break;
        case Activity::eIDLE:
            break;
        case Activity::eRUNNING:
            fSim.Insert(a, a->Ready_At());
        }
    }
}
```

Adding a Customer object to the simulation is easy:

```
void Shop::AddCustomer(Customer* c)
{
    fSim.Insert(c, c->Ready_At());
    fCustomersPresent++;
}
```

The DisplayState() and auxiliary PrintTime() functions provide a limited view of what is going on at a particular time:

```
void Shop::DisplayState()
{
    cout << "-----" << endl;
    cout << "Time ";
    PrintTime();
    cout << endl;
    cout << "Number of customers " << fCustomersPresent << endl;
}

void Shop::PrintTime()
{
    long hours = fTime/60;
    long minutes = fTime % 60;
    if(hours < 13) {
        cout << hours << ":" << setw(2) <<
            setfill('0') << minutes;
        if(hours < 12) cout << "a.m.";
        else cout << "p.m.";
    }
    else cout << (hours - 12) << ":" << setw(2) <<
        setfill('0') << minutes << "p.m.";
}
```

File Shop.cp has to #include the iomanip.h header file in order to use facilities like setw() and setfill() (these make it possible to print a time like seven minutes past eight as 8.07).

Most of the "logging" functions should have empty bodies, but for this test it would be worthwhile making Shop::LogNumberPurchases(int num) print the argument value. This would allow checks on whether the numbers were suitably "exponentially distributed".

class Door

The constructor for class Door initialises its various data members, getting the current time from the Shop object with which it is linked.

The only member function with any complexity is Run():

```
void Door::Run()
{
    if(fopen) {
        int count = Arrivals[fndx];
        if(count>0) {
            count = 1 + (rand() % (count + count));
            for(int i=0;i<count;i++)
                Customer* c = new
                    Customer(fs,Purchases[fndx]);
        }
        fndx++;
        fready = fs->Time() + 1;
    }
    else fState = eTERMINATED;
}
```

On successive calls, `Run()` takes data from successive locations in the file scope `Arrivals[]` and `Purchases[]` arrays. These data are used to determine the number of `Customer` objects to create. The first time that `Run()` gets executed after the `Close()` function, it changes the state to "terminated".

class Manager

The constructor for class `Manager` can be used to get the control parameters for the current run of the program. Most input data are used to set data members of the `Manager` object; the "maximum number of items for fast checkout" is used to set the appropriate data member of the `Shop` object. (The `Manager` is a friend so it can directly change the `Shop`'s data member.)

```
Manager::Manager(Shop* s)
{
    fs = s; // Set link to shop
    cout << "Enter time interval for managers checks on "
           "queues (min 5 max 60)\n";

    short t;
    cin >> t;

    if((t<5) || (t>60)) {
        t = 10;
        cout << "Defaulting to checks at 10 minute "
              "intervals\n";
    }

    ft = t;

    cout << "Enter maximum number of checkouts (15-40) ";
    ...
    fmaxcheckouts = t;

    cout << "Enter minimum number of fast checkouts (0-3) ";
    ...
    ...

    cout << "Enter maximum purchases for fast lane "
           "customers (5-15)\n";
    cin >> t;
    // Code to check value entered
    ...
    // then copy into Shop object's data member
    fs->ffastlanemax = t;

    ...

    fready = fs->Time();
    fQueuingLast = fShoppingLast = -1;
}
```

*Exploit "friendship"
with Shop*

The `Run()` function will have to be elaborated considerably in the final version of the program. In this partial implementation it has merely to get the shop to print its state, then if the time is before the closing time, it reschedules the `Manager` to run again after the specified period. If it is later than the closing time, the `Shop` should be reminded to close the door (if it isn't already closed).

```
void Manager::Run()
{
    fs->DisplayState();
    if(fs->Time() < kCLOSETIME) {
        fready = fs->Time() + ft;
    }
    else {
        fs->CloseDoor();
        fready = fs->Time() + 5;
        if(fs->fCustomersPresent == 0) fState = eTERMINATED;
    }
}
```

The `Manager` object can "terminate" if it is after closing time and there are no `Customer` objects left in the store.

class Customer

The constructor is probably the most elaborate function for this class. It has to pick the number of items to purchase and a shopping rate (change that `SHRT_MAX` to `RAND_MAX` if this is defined in your `limits.h` or other system header file). The shopping time is at least 2 minutes (defined as the constant `kACCESSTIME`) plus the time needed to buy items at the specified rate. This shopping time determines when the `Customer` will become ready.

The `Customer` object can immediately log some data with the `Shop`.

```
Customer::Customer(Shop* s, short t)
{
    fs = s;
    fitems = 1 + erand(t);
    while (fitems>250)
        fitems = 1 + erand(t);
    fstarttime = fs->Time();
    double ItemRate = 1.0 + 4.0*rand()/SHRT_MAX;
    int shoptime = kACCESSTIME + int(0.5 + fitems/ItemRate);
    fready = fstarttime + shoptime;
    fs->AddCustomer(this);
    fs->LogShopTime(shoptime);
    fs->LogNumberPurchases(fitems);
}
```

The destructor arranges for the `Customer` object to "check out" of the `Shop`:

```
Customer::~~Customer()
{
}
```

```
fs->LogTotalTime(fs->Time()-fstarttime);
fs->CustomerLeaves();
}
```

Testing

The other functions not given explicitly are all simple and you should find it easy to complete the implementation of this partial version of the program.

Traces from `LogNumberPurchases()` should gives number sequences like: 2, 2, 11, 2, 6, 3, 6, 12, 2, 6, 1, 15, 3, 6, 3, 5, 16, 6, 11, 3, 3, 19, ...; pretty much the sort of exponential distribution expected. A test run produced the following output showing the state of the shop at different times:

```
Time 8:00a.m.
Number of customers 8
-----
Time 8:10a.m.
Number of customers 8
-----
Time 8:20a.m.
Number of customers 5
-----
Time 8:30a.m.
Number of customers 19
-----
...
Time 9:10a.m.
Number of customers 2
-----
Time 9:15a.m.
Number of customers 0
```

27.4 FINALISING THE DESIGN

There are two main features not yet implemented – operation of checkouts with customers queuing at checkouts, and the histograms. The histograms are easier so they will be dealt with first.

27.4.1 Histograms

The `Histogram` class comes from the "tasks" class library.

The "tasks" library is often included with C++ systems. Since it is about fifteen years old its code is old fashioned. (Also, it contains some "coroutine" components that depend on assembly language routines to modify a program's stack. If the tasks library is not included with your compiler, it is probably because no one converted these highly specialized assembly language routines. A "Coroutine" is a specialized kind of control structure used mainly in sophisticated forms of simulation.)

class myhistogram

The histogram class keeps counts of the number of items (integer values) that belong in each of a set of "bins". It automatically adjusts the integer range represented by these bins. When all data have been accumulated, the contents of the bins can be displayed. Other statistics (number of items, minimum and maximum values, mean and standard deviation) are also output.

The interface for the class is simple. Its constructor takes two character strings for labels in the final printout, and a number of integers that define things like the number of bins to be used. The `Add()` member function inserts another data item while `Print()` produces the output summary. There are several data members; these include `char*` pointers to the label strings, records of minimum and maximum values observed, sum and sum of squares (needed to calculate mean and standard deviation) and an array of "bins".

```
// Based on code in ATT C++ "tasks" library.
class myhistogram {
public:
    myhistogram( char* title = "", char* units = "",
                int numbins = 16,int low = 0,int high = 16);
    void Add(int);
    void Print();
private:
    int l,r;           // total range covered
    int binsize;       // range for each "bin"
    int nbins;         // number of "bins"
    int *h;            // the array of "bins" with counts
    long sum;          // data for calculating average
    long sqsum;        // standard deviation etc
    short count;
    long max;          // max and min values recorded
    long min;
    char *atitle;      // pointers to labels
    char *aunits;
};
```

The constructor works out the number of bins needed, allocates the array, and performs related initialization tasks. (It simply stores pointers to the label strings, rather than duplicating them; consequently the labels should be constants or global variables.)

```
myhistogram::myhistogram(char* title, char* units,
                          int numbins, int low, int high)
{
    atitle = title;
    aunits = units;

    int i;
    if (high<=low || numbins<1) {
        cerr << "Illegal arguments for histogram\n";
        exit(1);
    }
    if (numbins % 2) numbins++;

    while ((high-low) % numbins) high++;
```

*Allocate the bins
array*

```

    binsize = (high-low)/numbins;
    h = new int[numbins];

    for (i=0; i<numbins; i++) h[i] = 0;
    l = low;
    r = high;
    nbin = numbins;
    sum = 0;
    sqsum = 0;
    count = 0;
    max = LONG_MIN;
    min = LONG_MAX;
}

```

The `Add()` member function does quite a lot more than just increment a counter. Most of the code concerns recalculation of the total range represented and size of each bin in accord with the data values entered. The various counters and sums etc are also updated.

```

void myhistogram::Add(int a)
{
    count++;

    max = (max > a) ? max : a;
    min = (min < a) ? min : a;

    /* add a to one of the bins, adjusting histogram,
    if necessary */
    int i, j;
    /* make l <= a < r, */
    /* possibly expanding histogram by doubling binsize
    and range */
    while (a<l) {
        l -= r - l;
        for (i=nbin-1, j=nbin-2; 0<=j; i--, j-=2)
            h[i] = h[j] + h[j+1];
        while(i >= 0) h[i--] = 0;
        binsize += binsize;
    }
    while (r<=a) {
        r += r - l;
        for (i=0, j=0; i<nbin/2 ; i++, j+=2)
            h[i] = h[j] + h[j+1];
        while (i < nbin) h[i++] = 0;
        binsize += binsize;
    }
    sum += a;
    sqsum += a * a;
    h[(a-l)/binsize]++;
}

```

The `Print()` function outputs the summary statistics and then loops through the array of bins outputting their values as numbers.

```

void myhistogram::Print()
{
    if(count <= 1) {
        cout << "Too little data for histogram!\n";
        return;
    }

    cout << "\n\n" << atitle << "\n\n";
    cout << "Number of samples " << count << "\n";

    double average = ((double)sum)/count;
    double stdev =
        sqrt(((double)sqsum - sum*average)/(count-1));

    cout << "Average = " << average << aunits << "\n";
    cout << "Standard deviation = " << stdev << aunits <<
    "\n";

    cout << "Minimum value = " << min << aunits << "\n";
    cout << "Maximum value = " << max << aunits << "\n";

    int i;
    int x;
    int d = binsize;

    for (i=0; i<nbin; i++) {
        x = h[i];
        if (x != 0) {
            int ll = l+d*i;
            cout << "[";
            cout << setw(4) << setfill(' ');
            cout << ll << " : " << ll + d << "]" << "\t: ";
            cout << setw(6) << x << "\n";
        }
    }
}

```

The function uses features from the `iomanip` library so the `iomanip.h` header file must be `#included`.

Use of histograms

The program is supposed to produce histograms showing the variations in numbers of items purchased, queuing times, shopping times, and total times spent by customers. Consequently, class `Shop` had better have `myhistogram` data members for each of these profiles. (The statistics needed for idle times of checkouts etc just require totals rather than histograms).

```

class Shop {
public:
    ...
private:
    ...
    // As before, plus
    myhistogram fPurchases;
    myhistogram fQTimes;
    myhistogram fShopTimes;
    myhistogram fTotalTimes;
}

```

```
};

The constructor for class Shop has to arrange for the initialization of its myHistogram data members. The constructor for the myHistogram class has defaults for its arguments, but we would want the histograms to have titles so explicit initialization is required:
```

```
Shop::Shop() : fPurchases("Number items purchased"),
               fQTimes("Queuing times"), fShopTimes("Shopping times"),
               fTotalTimes("Total time customer in shop")
{
    fTime = kSTARTTIME;
    fIdle = fworktime = 0;
    fCustomersPresent = fCustomersQueuing = 0;
}
```

Constructors for data members have to be invoked prior to entry to the body of the constructor for the class. As previously illustrated, Chapter 25, the calls to the constructors for data members come after the parameter list for the class's constructor and before the { begin bracket of the function body. A colon separates the list of data member constructors from the parameter list.

The "log" functions simply request that the corresponding myHistogram object "add" an item to its record:

```
void Shop::LogNumberPurchases(int num)
{
    fPurchases.Add(num);
}
```

The statistics gathered in a run are to be printed when the main loop in Shop::Run() completes. Class Shop should define an additional private member function, ReportStatistics(), that gets called from Run() to print the final information. A preliminary implementation would be:

```
void Shop::ReportStatistics()
{
    fPurchases.Print();
    fShopTimes.Print();
}
```

The existing partial implementation can be extended to test the histogram extensions. It should produce outputs like the following:

```
Number items purchased

Number of samples 234
Average = 6.371795
Standard deviation = 6.753366
Minimum value = 1
Maximum value = 47
[ 0 :4]      :    100
[ 4 :8]      :     72
```

*Changed constructor
for class Shop*

*Effective
implementation for
the "log" functions*

*Extra Shop::
ReportStatistics()
private member
function*

Retesting

```
[ 8 :12]      :     29
...
[ 44 :48]     :      1

Shopping times

Number of samples 234
Average = 4.602564
Standard deviation = 3.583485
Minimum value = 2
Maximum value = 39
[ 0 :4]      :    112
[ 4 :8]      :     96
...
[ 36 :40]     :      1
```

27.4.2 Simulating the checkouts and their queues

All that remains is the checkouts. The checkouts are relatively simple in themselves; the complexities lie in the operations of Shop and Manager that involve Checkout objects.

What do Checkout objects get asked to do?

Checkout objects will get created at the behest of the Manager (the Manager might create the Checkout and pass it to the Shop, or the Manager might direct the Shop object to handle creation). They are created as either "fast" or "standard" and don't subsequently change their type. When initially created, they are "idle".

Checkout objects will be asked to "add a customer" to a queue. The queue can be represented by a List data member; customers get appended to the end of the list and get removed from the front. The Shop object will be responsible for giving Customer objects to the Checkout objects. The Shop can take responsibility for dealing with cases where a Customer gets added to an idle Checkout; after it has dealt with the addition operation, the Checkout should get put into the Shop's PriorityQ fSim. The Checkout object however will have to do things like noting the time for which it has been idle.

Since a Checkout is a kind of Activity, it must provide implementations of Run() and Ready_At(). The Ready_At() function can be handled in the same way as was done for the other specialized Activity classes. The Run() function will get called when a Checkout is due to have finished with a Customer; that Customer can be deleted. If there are other Customer objects queued, the Checkout can remove the first from its list. The Customer should be told to report the length of time that it has been queuing. If the queue is empty, the Checkout should mark its state as "idle" and should notify the Shop.

The Manager object's rules for creating and destroying Checkout objects depend on details like average queue lengths and workloads. The Shop object is supposed to pick the best Checkout for a Customer; choosing the best depends on details of

Adding customers

*Ready_At() and
Run()*

*Queue length and
workload*

workload and type ("fast" or "standard"). Consequently, a Checkout will have to be able to supply such information.

Checkout objects that are not idle should display their queues as part of the *DisplayState* Shop object's `DisplayState()` process.

When a Checkout object gets deleted, it should report details of its total time of operation and its idle time. These reports get made to the Shop object so that relevant overall statistics can be updated. *Destructor*

What do Checkout objects own?

Checkout objects will need:

- a pointer to the Shop object;
- long integers representing time current task completed, time current task started, time of creation, total of idle periods so far;
- a list to hold the queuing Customer objects;
- a pointer to the Customer currently being served;
- a short to hold the "fast" or "standard" type designation.

Design diagram for class Checkout

Figure 27.6 is a class "design diagram" that summarizes the features of class Checkout. Such diagrams are often used as a supplement to (or alternative to) textual descriptions like those given earlier for the other Activity classes like class Manager.

The diagram in Figure 27.7 is simpler than the styles that you will be using later when you have been taught the current standard documenting styles, but it still provides an adequate summary. The header should give details of inheritance. There should be a means of distinguishing the public interface and private implementation. Public functions should have a brief comment explaining their role (often details of return types and argument lists are omitted because these design diagrams can be sketched out before such fine details have been resolved).

It is often useful to distinguish between "own data" and "links to collaborators". Both are implemented as "data members". It is just a matter of a different role, but it is worth highlighting such differences in documentation.

Implementation of class Checkout

The implementation of class Checkout will identify a few extra member functions that will be needed in class Customer and class Shop.

For example, a Checkout has to notify a Customer when processing starts; this is needed to allow the Customer object to work out its queuing time and report this to the Shop. Consequently, class Customer will need an additional function, "finish queuing", in its public interface.

Such extra functions would be noted while the implementation of class Checkout was sketched out. Subsequently, the extra functions would be defined in the other classes.

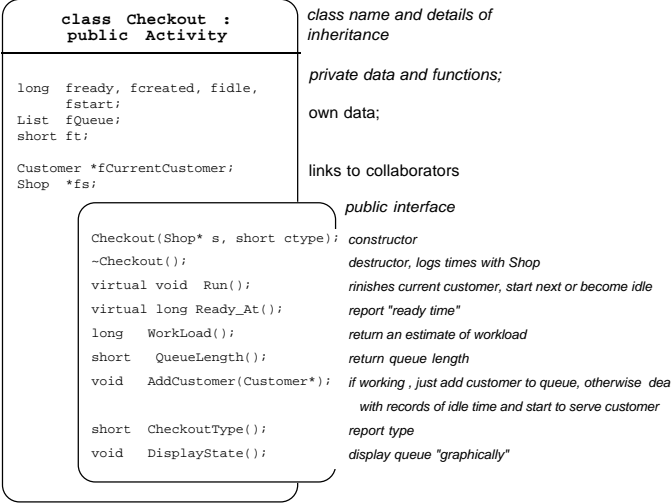


Figure 27.7 Design diagram for class Checkout.

Checkout's constructor

Previous examples have shown cases where data members that were instances of classes had to have their constructors invoked prior to entry to the main body of a constructor. This time, we need to invoke the base class's constructor (`Activity::Activity()`). The other specialized subclasses of `Activity` relied on the default parameters to initialize the "activity" aspect of their objects. But, by default, the initializer for class `Activity` creates objects whose state is "running". Here we want to create an object that is initially "idle". Consequently, we have to explicitly invoke the constructor with the appropriate `eIDLE` argument; while we are doing that we might as well initialize some of the other data members as well:

```
Checkout::Checkout(Shop* s, short t) : Activity(eIDLE),
    fs(s) , ft(t)
{
    fCurrentCustomer = NULL;
    fidle = 0;
    fcreated = fready = fs->Time();
}
```

The "type" of a Checkout (argument `t` for constructor, data member `ft`) could use some integer coding (e.g. 1 => fast, 0 => standard).

The destructor should finalize the estimate of idle time and then log the Checkout's idle and open times with the Shop:

Destructor

```
Checkout::~Checkout()
{
    fidle += fs->Time() - fready;
    fs->LogIdleTime(fidle);
    fs->LogOpenTime(fs->Time() - fcreated);
}
```

The Run() function starts by getting rid of the current customer (if any). Then, if there is another Customer in the queue, this object is removed from the queue to become the current customer, is notified that it has finished queuing, and a new completion time calculated based on the processing rate (kSCANRATE) and the number of items purchased.

```
void Checkout::Run()
{
    if(fCurrentCustomer != NULL) {
        delete fCurrentCustomer;
        fCurrentCustomer = NULL;
    }

    if(fQueue.Length() > 0) {
        fCurrentCustomer = (Customer*) fQueue.Remove(1);
        fCurrentCustomer->FinishQueuing();
        fstart = fs->Time();
        short t = fCurrentCustomer->ItemsPurchased();
        t /= kSCANRATE;
        t = (t < 1) ? 1 : t;
        fready = fstart + t;
    }
    else {
        fState = eIDLE;
        fs->NoteCheckoutStopping(this);
    }
}
```

Alternatively, if its queue is empty, the Checkout becomes idle and notifies the Shop of this change.

The version of List used for this example could be that given in Chapter 26 with the extra features like a ListIterator. The Checkout::WorkLoad() function can use a ListIterator to run down the list of queuing customers getting each to state the number of items purchased. Unscanned items from the current customer should also be factored into this work load estimate.

Using a ListIterator

```
long Checkout::WorkLoad()
{
    long res = 0;
    ListIterator LL(&fQueue);
    LL.First();
```

ListIterator

```
while(!LL.IsDone()) {
    Customer *c = (Customer*) LL.CurrentItem();
    res += c->ItemsPurchased();
    LL.Next();
}

if(fCurrentCustomer != NULL) {
    long items = fCurrentCustomer->ItemsPurchased();
    long dlta = (items - kSCANRATE*(fs->Time()-fstart));
    dlta = (dlta < 0) ? 0 : dlta;
    res += dlta;
}

return res;
}
```

The work involved in adding a customer depends on whether the Checkout is currently idle or busy. Things are simple if the Checkout is busy; the new Customer is simply appended to the list associated with the Checkout. If the Checkout is idle, it can immediately start to serve the customer; it should also notify the Shop so that the records of "idle" and "busy" checkouts can be updated.

```
void Checkout::AddCustomer(Customer* c)
{
    if(fState == eIDLE) {
        fidle += fs->Time() - fready;
        fCurrentCustomer = c;
        fCurrentCustomer->FinishQueuing();
        fstart = fs->Time();
        short t = fCurrentCustomer->ItemsPurchased();
        t /= kSCANRATE;
        t = (t < 1) ? 1 : t;
        fready = fstart + t;
        fState = eRUNNING;
        fs->NoteCheckoutStarting(this);
    }
    else fQueue.Append(c);
}
```

The final DisplayState() function just has to provide some visual indication of the number of customers queuing, e.g. a line of '*s'.

27.4.3 Organizing the checkouts

The addition of Checkout objects in the program requires only a minor change in class Customer. Its Customer::Run() function no longer sets its state to terminated, instead a Customer should set its state to eIDLE and ask the Shop object to find it a Checkout where it can queue.

However, classes Shop and Manager must have major extensions to allow for Checkouts.

The Manager object will be telling the Shop to add or remove Checkouts; the type (fast or standard) will be specified in these calls:

Additional responsibilities for class Shop


```
void      Shop::AddCheckout(short ctype);
void      Shop::RemoveIdleCheckout(short ctype);
```

Checkout objects notify the Shop when the start or stop work:

```
void      Shop::NoteCheckoutStarting(Checkout* c);
void      Shop::NoteCheckoutStopping(Checkout* c);
```

and, as just noted, Customer objects will be asking the Shop to move them onto queues at Checkouts:

```
void      Shop::QueueMe(Customer* cc);
```

The Shop has to keep track of the various Checkouts, keeping idle and busy checkouts separately. It would probably be easiest if the Shop had four List objects in which it stored Checkouts:

```
List      fFastIdle;
List      fStandardIdle;
List      fFastWorking;
List      fStandardWorking;
```

The AddCheckout(), RemoveIdleCheckout() will both involve "idle" lists; the ctype argument can identify which list is involved.

When a Checkout notifies the Shop that it has stopped working, the Shop can move that Checkout from a "working" list to an "idle" list (the Shop can ask the Checkout its type and so determine which lists to use). Similarly, when a Checkout starts, the Shop can move it from an "idle" list to a "working" list (and also get it involved in the simulation by inserting it into the priority queue as well).

The only one of these additional member functions that is at all complex is the Checkout::QueueMe() function which is outlined below.

The Manager object will need information on the numbers of idle and busy checkouts etc. Since the Manager is a friend of class Shop, it can simply make calls like:

```
long num_working = fs->fFastWorking.Length() +
                  fs->fStandardWorking.Length();
long num_idle = fs->fFastIdle.Length() +
                fs->fStandardIdle.Length();
```

Here, the Manager uses its Shop* pointer fs to access the Shop object, exploiting its friend relation to directly use a private data member like fFastWorking; the List::Length() function is then invoked for the chosen List object.

The additional responsibilities for class Manager all relate to the application of those rules for choosing when to open and close checkouts. The Manager::Run() function needs to get these rules applied each time a check on the shop is made; this function now becomes something like:

```
void Manager::Run()
{
```

Extra data members

Manager exploits friend relation

```
fs->DisplayState();
if(fs->Time() < kCLOSETIME) {
    Sortoutcheckouts();
    fready = fs->Time() + ft;
}
else {
    fs->CloseDoor();
    Closingcheckouts();
    fready = fs->Time() + 5;
    if(fs->fCustomersPresent == 0)
        fState = eTERMINATED;
}
}
```

There are two additional calls to new functions that will become extra private member functions of class Manager. The function Sortoutcheckouts() will apply the full set of rules that apply during opening hours; the Closingcheckouts() function will apply the simpler "after 7 p.m. rule".

A function like Closingcheckouts() is straightforward. The Manager can get details of the numbers of checkouts (fast and standard, idle and working) by interrogating the List data members of the Shop. Using these data, it can arrange to close checkouts as they become idle (making certain that some checkouts are left open so long as there are Customers still shopping):

```
void Manager::Closingcheckouts()
{
    /* It is after 7pm, close the fast checkouts as they
    become idle. */
    int fastidle = fs->fFastIdle.Length();
    for(int i=0; i < fastidle; i++)
        fs->RemoveIdleCheckout(kFAST);

    int standardidle = fs->fStandardIdle.Length();
    int standardworking = fs->fStandardWorking.Length();

    /*
    So long as there are checkouts working can get rid of
    all idle ones (if any).
    */
    if((standardworking > 0) && (standardidle > 0)) {
        for(i=0; i < standardidle; i++)
            fs->RemoveIdleCheckout(kSTANDARD);
        return;
    }

    /*
    If there are no customers left close all idle checkouts
    */
    if(fs->fCustomersPresent == 0) {
        for(i=0; i < standardidle; i++)
            fs->RemoveIdleCheckout(kSTANDARD);
        return;
    }
}

/*
```

```
May end up with state where there are customers still
shopping but all checkouts idle. Close all but one.
*/
for(i=0; i < standardidle - 1; i++)
    fs->RemoveIdleCheckout(kSTANDARD);
}
```

The rules that define normal operation are too complex to be captured in a single function. Once again, the functional decomposition approach has to be applied. The Manager has to consider three aspects: checking that minimum numbers of checkouts are open, checking for idle checkouts that could be closed, and checking for excessive queues that necessitate opening of checkouts. Inevitably, the Sortoutcheckouts() function becomes:

```
void Manager::Sortoutcheckouts()
{
    OpenMinimumCheckouts();
    LookAtIdleCheckouts();
    ConsiderExtraCheckouts();
}
```

The function has been decomposed into three simpler functions that allow individual consideration of the different rules. In some cases, further decomposition into yet simpler functions is necessary. Aspects of these functions are illustrated below.

Final class definitions

Figures 27.8 and 27.9 show the finalized design diagrams for classes Manager and Shop.

Use "top down functional decomposition" for complex functions

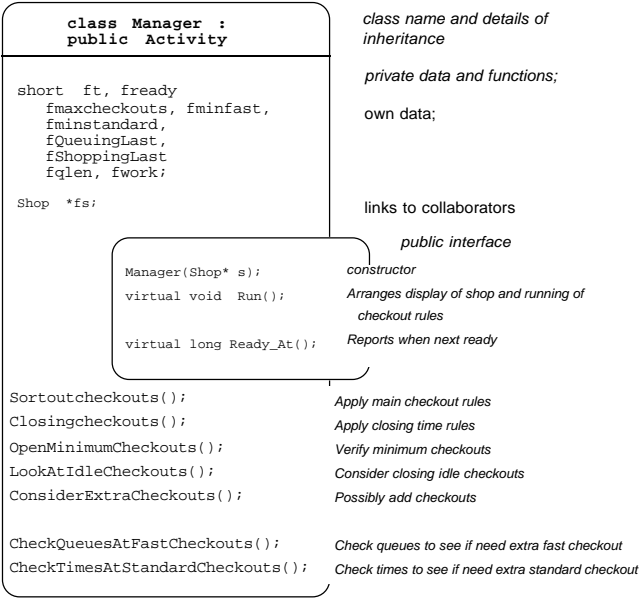


Figure 27.8 Final design diagram for class Manager.

Class Manager retains a very simple public interface. The Manager object is only used in a small number of ways by the rest of the program. However, the things that a Manager gets asked to do are complex, and consequently the class has a large number of private implementation functions.

Class Shop has an extensive public interface – many other objects ask the Shop to perform functions. Most of these member functions are simple (just increment a count, or move the requestor from one list to another); consequently, there is only limited need for auxiliary private member functions.

Diagram 27.9 also indicates that a "friend" relationship exists that allows a Manager object to break the normal protection around the private data of the Shop object.

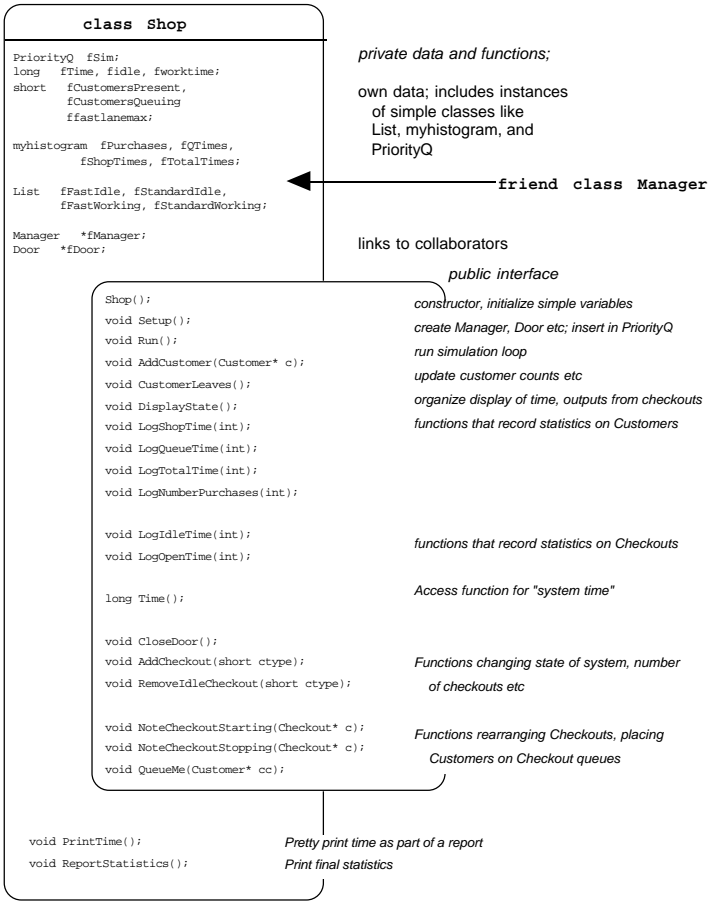


Figure 27.9 Final design diagram for class Shop.

Implementation of the more elaborate functions

The function `Shop::QueueMe(Customer*)` involves finding an appropriate Checkout for the requesting Customer object.

The code starts by determining whether the Customer is able to use both fast and standard Checkouts, or just the standard Checkouts.

Next, the function checks for idle Checkouts (both types if the customer is allowed to use the fast lanes, otherwise just the "idle standard" Checkouts). If there is a suitable idle Checkout, the Customer can be queued and the function finishes:

```
void Shop::QueueMe(Customer* cc)
{
    int fast = (cc->ItemsPurchased() <= ffastlanemax) ||
              (99 == (rand() % 100));

    /*
    If appropriate, try for an idle checkout
    */
    if(fast && (fFastIdle.Length() > 0)) {
        Checkout *c = (Checkout*) fFastIdle.Nth(1);
        c->AddCustomer(cc);
        return;
    }

    if(fStandardIdle.Length() > 0) {
        Checkout *c = (Checkout*) fStandardIdle.Nth(1);
        c->AddCustomer(cc);
        return;
    }

    If there is no idle Checkout, the function has to search through either both lists
    of working Checkouts (or just the list of standard checkouts) to find the one with
    the least load. These searches through the lists can again take advantage of
    ListIterators:

    long least_load = LONG_MAX;
    Checkout *best = NULL;

    if(fast) {
        ListIterator L1(&fFastWorking);
        L1.First();
        while(!L1.IsDone()) {
            Checkout *c = (Checkout*) L1.CurrentItem();
            long load = c->WorkLoad();
            if(load < least_load) {
                least_load = load;
                best = c;
            }
            L1.Next();
        }
    }

    ListIterator L2(&fStandardWorking);
    // Code similar to last ListIterator loop
}
```

The Customer chooses the Checkout with the smallest workload (there had better be a test to verify that there was a Checkout where the Customer could queue):

```
        if(best == NULL) {
            cout << "Store destroyed by rioting customers."
                 << endl;
            exit(1);
        }
        else best->AddCustomer(cc);

        fCustomersQueuing++;
    }
}
```

The remaining functions of class Manager also involve interrogation of the various List data members of the associated Shop object; there are also several places where ListIterators get employed to work through all entries in one of these lists. The most complex of the rules used by the Manager is that relating to the opening of extra checkouts. The rule starts by saying that these should be considered if the number of customers queuing or shopping has increased. If this is the case, then an additional fast checkout can be opened if the average queue length is too long at fast lanes. Similarly, one or more additional standard checkouts is to open if the wait time is too long. Naturally, this breaks down into separate cases handled by separate functions. The ConsiderExtraCheckouts() function establishes the context for adding checkouts:

```
void Manager::ConsiderExtraCheckouts()
{
    long shopping = fs->fCustomersPresent -
                  fs->fCustomersQueuing;
    long queuing = fs->fCustomersQueuing;

    if((shopping>fShoppingLast) || (queuing > fQueuingLast)) {
        CheckQueuesAtFastCheckouts();
        CheckTimesAtStandardCheckouts();
    }
    fShoppingLast = shopping;
    fQueuingLast = queuing;
}
```

The function that adds fast checkouts is representative of the remaining functions. It identifies circumstances that preclude the opening of new checkouts (e.g. maximum number already open) and if any pertain it abandons processing:

```
void Manager::CheckQueuesAtFastCheckouts()
{
    /*
     * if there are any idle checkouts do nothing.
     */

    int idle = fs->fFastIdle.Length() +
              fs->fStandardIdle.Length();
    if(idle != 0)
        return;

    /*
     * If don't have any fast checkouts open, can't check
```

If getting busier, add more checkouts

```
average length so do nothing. (If number of customers
is large, a fast checkout will have been created by
the minimum checkouts rule.)
*/

int num = fs->fFastWorking.Length();
if(num == 0)
    return;

int total = num + fs->fStandardWorking.Length();

if(total == fmaxcheckouts)
    return;
```

The next step involves iterating through existing working checkouts gathering information needed to determine current load. Naturally, this is done with the aid of a ListIterator:

```
int queuing = 0;
ListIterator Ll(&(fs->fFastWorking));
Ll.First();
while(!Ll.IsDone()) {
    Checkout *c = (Checkout*) Ll.CurrentItem();
    queuing += c->QueueLength();
    Ll.Next();
}

int average = queuing / num;
```

If the average queue length is too great, an extra checkout should be added:

```
if(average < fqlen)
    return;

fs->AddCheckout(kFAST);
cout << "Added a fast checkout." << endl;
}
```

Execution

You should find it fairly simple to complete the implementation (Exercise 1). The program should produce outputs like the following:

```
Time 8:30a.m.
Number of customers 65
Fast Checkouts:
Standard Checkouts:
Added a fast checkout.
Added 1 standard checkouts.
-----
Time 8:40a.m.
Number of customers 84

Fast Checkouts:
Standard Checkouts:
Time 9:30a.m.
Number of customers 34
There are 2 idle fast
checkouts.
Fast Checkouts:
```

```
*|
Standard Checkouts:
*|***
*|**

Closed 2 fast checkouts

Time 7:00p.m.
Number of customers 15
There are 2 idle fast
checkouts.
There are 10 idle standard
checkouts.
Standard Checkouts:
*|
*|
*|
*|
*|
*|
*|
Door closed.

Shop closing at 7:35p.m.
Checkout times:
Total 8720 minutes
Idle 834 minutes

Shopping times

Number of samples 2627
Average = 3.24705
Standard deviation = 3.69674
Minimum value = 0
Maximum value = 23
[ 0 :2] : 1170
[ 2 :4] : 509
[ 4 :6] : 314
[ 6 :8] : 255
[ 8 :10] : 208
[ 10 :12] : 84
[ 12 :14] : 37
[ 14 :16] : 26
[ 16 :18] : 12
[ 18 :20] : 7
[ 20 :22] : 3
[ 22 :24] : 2

[ 160 :176] : 2
[ 176 :192] : 1

Queuing times

Number of samples 2627
Average = 3.24705
Standard deviation = 3.69674
Minimum value = 0
Maximum value = 23
[ 0 :2] : 1170
[ 2 :4] : 509
[ 4 :6] : 314
[ 6 :8] : 255
[ 8 :10] : 208
[ 10 :12] : 84
[ 12 :14] : 37
[ 14 :16] : 26
[ 16 :18] : 12
[ 18 :20] : 7
[ 20 :22] : 3
[ 22 :24] : 2

[ 0 :16] : 1862
[ 16 :32] : 462
[ 32 :48] : 179
[ 48 :64] : 67
[ 64 :80] : 28
[ 80 :96] : 14
[ 96 :112] : 7
[ 112 :128] : 1
[ 128 :144] : 3
[ 144 :160] : 1
```

EXERCISES

- 1 Complete a working version of the Supermarket program.
- 2 Implement a simulation of an "office information system".

The system has a number of sources of "messages" – electronic mail, a facsimile machine, a local network, etc. The different kinds of incoming messages are all placed in a single queue for processing by the executive using the system. The user can get details of the queue displayed; these details will include the number of queued messages, and a list showing the headers for each message. These header details are to include "arrival time", "sender", and "topic" (each of these is a short string). The user can select messages (identifying them by an index number) for detailed display. Once selected, a message becomes the "current message". The system is to allow the user to display the content of the current message, requeue it, save it to file or delete it. Different message types display their content in distinct ways.

The message sources should work with data files. These text files will contain successive messages. Each message in the file starts with an "arrival time", a "sender" name, and "topic header". The content part of the message will follow; different content forms should be used for the different message types. The simulation will give message sources an opportunity to "run" at regular intervals. When a message source runs, it should read successive messages from its input file, creating appropriate message objects that get added to the main queue. Data should continue to be read, and message objects be created, until the next "arrival time" is greater than the current simulated system time.

The system is to have a simulated time scale based on actual execution time (use the functions provided in your IDE for accessing the clock). For example, six seconds of execution time could represent one minute of simulated time.

The simulation will involve a loop in which the user is prompted for a command. The program will pause until data is entered. The computer's clock is read after the user input and used to update the simulated time. All message sources are given the opportunity to "run", possibly resulting in additional messages being added to the main queue. Then the users command should be interpreted. Finally, the "simulated time" should be displayed and the user should be prompted for the next command.

28 Design and documentation: 2

The examples given in earlier chapters have in a rather informal way illustrated a design style that could be termed "object-based design".

At the most general level, there are similarities to the "top down functional decomposition" approach discussed in Chapter 15. A design process involves:

- *decomposition*, i.e. breaking a problem into smaller subparts that can be dealt with largely independently;

and

- *iteration*, a problem gets worked through many times at increasing levels of detail; provisional decisions get made, are tested through prototyping, and subsequently may be revised.

The most significant difference between object-based design and functional decomposition is in the initial focus for the decomposition process.

Top-down functional decomposition is an effective approach when the program has the basic form: input some data, compute some "function" of the data, output the result. In addition, the data must be simple in form. In such cases, the "function" that must be computed using the data serves as an obvious focus for the design process.

A focus on function

There are still many scientific, statistical, and business applications that have this relatively simple form. Examples are things like "compute the neutron flux in this reactor", or "calculate our total wage and tax bills given these employee time sheets". The function that must be computed for the data may be complex, but the overall program structure is simple.

Other programs don't fit this pattern. Many are "event handlers". They have some source of "events". These "events" may result from a user working interactively with keyed commands, menu-selections, or mouse actions. In other cases, the "events" may

*"Multi-functional
event handler
programs"*

come through external network connections reflecting actions by collaborating workers or autonomous devices. In simulation programs, the "events" may be created by the program's own logic.

"Events" represent requests for particular actions to be performed, particular functions to be executed. The program has many functions. The ones that are performed, and the sequence in which they are performed depends on the events received. The event patterns, and hence the function calls, differ in every run of the program.

Events often involve the creation (or deletion) of data elements. These data elements are of many different types. Their lifetimes vary. They are interrelated in complex ways. Frequently, simple individual data elements are linked together through pointers to build up models of more elaborate data structures.

These programs may still involve complex calculations. For example, you could have a "Computer Aided Design" (CAD) engineering program for designing nuclear reactors. Such a program would involve a sophisticated interactive component (a little like one of the better draw programs that you can get for personal computers), components for display of three dimensional structures, and – a complex function to calculate neutron fluxes. But the calculation is only one minor aspect of the overall program.

It is possible to design these programs using top-down functional decomposition. Though possible, such an approach to design is frequently problematical. There is no obvious starting point, no top from which to decompose downwards. You can identify the different main functions, e.g. edit, 3-d model, calculate, and you can decompose each of these. But then you end up with separate groups of functions that have to communicate through some shared global data. It is very easy for inconsistencies to arise where the different groups of functions embody different assumptions about the shared data.

A focus on data

Object-based programming is a design approach that helps deal with these more complex programs. The focus switches to the data.

The data will involve composites and simple data elements. For example, a CAD program has an overall structure (the engineering component being built) that is a composite made up from many individual parts of different types. Although there are different kinds of individual part (pipe, bracket, rod, ...), all have data such as dimensions, mass, material type, and all can do things like draw themselves on a screen. The individual parts can look after their own data, the composite structure can deal with things like organizing a display (identify those parts that would be visible, tell them to draw themselves).

Inherently, data objects provide a basis for problem decomposition. If you can identify a group of data values that belong together, and a set of transformations that may be applied to those data, you have found a separable component for the overall program. You will be able to abstract out that component out and think about in isolation. You can design a class that describes the data owned and the things that can be done.

Sometimes, the things that an object must do are complex (e.g. the `Manager` object's application of the scheduling rules in the Supermarket example). In such cases, you can adopt a "top-down functional decomposition approach" because you are in the same situation as before – there is one clearly defined function to perform and the data are relatively simple (the data needed will all be represented as data members of the object performing the action).

You can code and test a class that defines any one type of data and the related functionality. Then you can return to the whole problem. But now the problem has been simplified because some details are now packaged away into the already built component.

When you return to the whole problem, you try to characterize the workings program in terms of interactions amongst example objects: e.g. "the structure will ask each part in its components list to draw itself".

If you design using top-down functional decomposition, you tend to see each problem as unique. If take an object based approach, you are more likely to see commonalities.

As just noted, most object-based programs seem to have "composite" structures that group separate components. The mechanisms for "grouping" are independent of the specific application. Consequently, you can almost always find opportunities for reusing standard components like lists, dynamic arrays, priority queues. You don't need to create a special purpose storage structure; you reuse a standard class.

An object-based approach to design has two major benefits: i) a cleaner, more effective decomposition of a complex problem, and ii) the opportunity to reuse components.

The use of abstract classes and inheritance, "object oriented design", brings further benefits. These have been hinted at in the Supermarket example, and in the discussion above regarding the CAD program and the parts that it manipulates. These design benefits are explored a little more in Part V.

Opportunity for reuse

28.1 OBJECT-BASED DESIGN

In Chapter 15, on "top-down functional decomposition", it was suggested that you could begin with a phrase or one sentence summary that defines the program. That doesn't help too much for something like the Supermarket, the InfoStore, or even the RefCards example. One sentence summaries don't say much.

"The RefCards program allows a user to keep a collection of reference cards." What are "reference cards"? Does the user do anything apart from "keep" them ("keeping" doesn't sound very interesting)? Explanations as to what the program really does usually fill several pages.

So, where do you begin?

You begin by trying to answer the questions:

Beginning

- What are the objects?
- What do they own?
- What do they do?

You start by thinking about prototypical objects, not the classes and certainly not abstract class hierarchies.

***Identifying
prototypical objects***

Some ideas for the prototypical objects can come from a detailed reading of the full specification of the program (the "underline the nouns" approach). As previously noted, there are problems with too literally underlining the nouns; you may end modelling the world in too much detail. But it is a starting point for getting ideas as to things that might be among the more important objects – thus, you can pick up the need for `Customers` and `Checkouts` from the description of the Supermarket problem.

Usually, you will find that the program has a few objects that seem to be in for the duration, e.g. the `UserInteraction` and `CardCollection` objects in the `RefCards` program, or the `Shop`, `Manager`, and `Door` objects in the Supermarket example. In addition there are other objects that may be transient (e.g. the `Customers`). An important aspect of the design will be keeping track of when objects get created and destroyed.

Scenarios-1

Once you have formed at least some idea as to the objects that might be present in the executing program, it is worthwhile focussing on "events" that the program deals with and the objects that are involved. This process helps clarify what each kind of object owns and does, and also begins to establish the patterns of communication amongst classes.

You make up scenarios for each of the important "events" handled by the program. They should include the scenarios that show how objects get created and destroyed.

You must then compare the scenarios to check that you are treating the objects in a consistent manner. At the same time, you make up lists of what objects are asked to do and what data values you think that they should own.

***Products of the first
step***

Once you have seen the ways that your putative objects behave in these scenarios you compose your initial class descriptions. These will include:

- class name, e.g. `Shop`
- data owned: e.g. "several histograms, some `Lists` to store `Checkouts`, a timer value, ..."
- responsibilities: "adds a checkout (requested by `Manager`), notes when a checkout becomes idle (`Checkout`), finds a `Checkout` for a `Customer` (`Customer`), reports the time (various client classes), ..."
- uses: (summary of requests made to instances of other classes), e.g. "`Run()` (all `Activity` subclasses), `Checkout::AddCustomer()`, ..."

The responsibilities of the classes are all the things that you have seen being asked of prototypical instances in the scenarios that you have composed. It is often worthwhile noting the classes of client objects that use the functions of a class. In addition, you should note all the requests made to instances of other classes.

The pattern of requests made to and by an instance of a class identify its collaborators. If two objects are to collaborate, they have to have pointers to one another (e.g. the `Shop` had a `Manager*` pointer, and the collaborating `Manager` had a `Shop*` pointer). These pointers must get set before they need to be used.

"Collaborators"

Setting up the pointers linking collaborators hasn't been a problem in the examples presented so far. In more complex programs, the establishment of collaboration links can become an issue. Problems tend to be associated with situations where instances of one class can be created in different ways (e.g. read from a file or interactive command from a user). In one situation, it may be obvious that a link should be set to a collaborator. In the other situation, it may not be so obvious, and the link setting step may be forgotten. Forgetting links results in problems where a program seems to work intermittently.

The highlighting of collaborations in the early design stage can act as a reminder so that later on, when considering how instances of classes are created, you can remember to check that all required links are being set.

Sometimes you will get a class whose instances get asked to look after data and perform various tasks related to their data, but which don't themselves make requests to any other objects in the system. They act as "servers" rather than "clients" in all the "collaborations" in which they participate.

Isolable components

Such classes represent completely isolable components. They should be taken out of the main development. They can be implemented and tested in isolation. Then they can be reintroduced as "reusable" classes with the same standing as classes from standard libraries. The `InfoStore` example program provides an example; its `Vocab` class was isolable in this way.

You will get class hierarchies in two ways. Occasionally, the application problem will already have a hierarchy defined. The usual example quoted is a program that must manipulate "bank accounts". Now "bank accounts" are objects that do various things like accept debits and credits, report their balance, charge bank fees, and (sometimes) pay interest. A bank may have several different kinds of account, each with rather different rules regarding fees and interest payments. Here a hierarchy is obvious from the start. You have the abstract class "bank_account" which has pure virtual functions "DeductCharges()" and "AddInterest()". Then there are the various specialized subclasses ("loan_account", "savings", "checking", "checking_interest") that implement distinct versions of the virtual functions.

Class hierarchies

Other cases are more like the `Supermarket` example. There we had classes `Manager`, `Door`, `Checkout`, and `Customer` whose instances all had to behave "in the same way" so as to make it practical for the simulation system to use a single priority queue. This was handled by the introduction of an abstraction, class `Activity`, that became the base class for the other classes. Class `Activity` wasn't essential (the `Shop` could have used

four different priority queues); but its introduction greatly simplified design. The class hierarchy is certainly not one that you would have initially expected and doesn't reflect any "real world" relationship. (How many common features can you identify between "doors" and "customers"?)

Second step

Your initial classes are little more than "fuzzy blob" outlines. You have some idea as to the data owned and responsibilities but details will not have been defined. For example, you may have decided that "class Vocab owns the vocabulary and provides both fast lookup of words and listings of details", or that "class Manager handles the scheduling rules". You won't necessarily have decided the exact form of the data (hash-table or tree), you won't have all the data members (generally extra counters and flags get added to the data members when you get into more detail), and you certainly won't have much idea as to how the functions work and whether they necessitate auxiliary functions.

The next step in design is, considering the classes individually, to try to move from a "fuzzy blob" outline to something with a firm definition. You have to define the types of all data members (and get into issues like how data members should be initialized). Each member function has to be considered, possibly being decomposed into simpler auxiliary private member functions.

Outputs from design step

The output of this step should be the class declarations and lists of member functions like those illustrated in the various examples. Pseudo-code outlines should be provided for all the more complex member functions.

main()

The `main()` function is usually trivial: create the principle object, tell it to run.

Module structure

These programs are generally built from many separate files. The design process should also cover the module (file) structure and the "header dependencies". Details should be included with the rest of the design in the form of a diagram like that shown in 27.6.

Tests

As always, some thought has to be given to the testing of individual components and of the overall program.

28.2 DOCUMENTING A DESIGN

Diagrams are a much more important part of the documentation of the design of an object-based program than they were for the "top-down functional decomposition" programs.

Your documentation should include:

- "fuzzy" blob diagrams showing the classes and their principle roles (e.g. Figures 22.1 and 22.9);
- a hierarchy diagram (if needed); this could be defined in terms of the fuzzy blob classes (e.g. Figure 27.4) or the later design classes;
- scenarios for all important interactions among instances of classes;

- class "design diagrams" that summarize the data and function members of a class, (e.g. Figures 27.8 and 27.9);
- module structure;
- class declarations and member function summaries;
- pseudo-code outlines for complex functions.

29 The Power of Inheritance and Polymorphism

This chapter is another that presents just a single example program illustrating how to apply features of the C++ language. In this case, the focus is on class inheritance and the use of polymorphism. The application program is a simple game.

In the days before "Doom", "Marathon", and "Dark Forces", people had to be content with more limited games! The basic game ideas were actually rather similar. The player had to "explore a maze", "find items", and avoid being eliminated by the "monsters" that inhabited the maze. However, details of graphics and modes of interaction were more limited than the modern games. A maze would be represented as an array of characters on the screen (walls represented by '#' marks, staircases between levels by characters like '<' and so forth). Single character commands allowed the player to move the maze-explorer around within the boundaries of the maze. "Rogue" and "Moria" are typical examples of such games; you may find that you have copies on old floppies somewhere.

You aren't quite ready to put Lucas Arts out of business by writing a replacement for "Dark Forces", but if you have got this far you can write your own version of Moria.

The example code given here is simplified with lots of scope for elaboration. It employs a single level maze (or "dungeon"). A map of the dungeon is always displayed, in full, on the screen. The screen size limits the size of the map; having the complete map displayed simplifies play. (Possible elaborations include showing only those parts of the map already explored and "scrolling" maps that are too large to fit on a screen.) The map and other data are displayed using the same system features as used in the "cursor graphics" examples from Chapter 12.

Example program

The map, details of the items to be found, and the monsters that are to be avoided (or destroyed) are taken from text file input. Again, this is a simplification. Games like Moria usually generate new maps for every game played.

The playing mechanism is limited. The user is prompted for a command. After a user command has been processed, all the "monsters" get a chance to "run". A "Monster:: Run()" function captures the essence of "monsterness" i.e. a desire to eliminate the human player.

Naturally, such a game program requires numerous obvious objects – the "monsters", the "items" that must be collected, the "player" object, the dungeon

Objects everywhere

object itself. In addition there will have to be various forms of "window" object used to display other information. Since there will be many "monsters" and many "items", standard collection classes will be needed.

Windows hierarchy

There are two separate class hierarchies as well as a number of independent classes. One limited class hierarchy defines "windows". There is a basic "window" class for display of data with a couple of specializations such as a window used to output numeric data (e.g. player's "health") and a window to get input. (These "window" classes are further elaborated in the next chapter.)

"Dungeon Items" hierarchy

There is also a hierarchy of "things found in the dungeon". Some, like the "items" that must be collected, are passive, they sit waiting until an action is performed on them. Others, like the player and the monsters, are active. At each cycle of the game, they perform some action.

Polymorphic pointers

Naturally, there are many different kinds of monster. Each different kind (subclass) employs a slightly different strategy when trying to achieve their common objective of "eliminating the player". This is where the polymorphism comes in. The "dungeon" object will work with a collection of monsters. When it is the monsters' turn, the code has a loop that lets each monster "run" (`Monster *m; ...; m->Run();`). The pointer `m` is polymorphic – pointing to different forms of monster at different times. Each different form has its own interpretation of `Run()`.

29.1 THE "DUNGEON" GAME

The "dungeon" game program is to:

- Read game details from a text file. These details are to include the map layout, the initial positions and other data defining collectable items, the monsters, and the player.
- Provide a display similar to that shown in Figure 29.1. This display is to include the main map window (showing fixed features like walls, position of collectable items, and current positions of active items) and other windows that show the player's status.
- Run the game. The game terminates either by the player object acquiring all collectable items or by its "health" falling to zero.
- Operate a "run cycle" where the user enters a movement command (or "magic action" command – see below), the command is executed, and then all other active items get a chance to run.
- Arrange that the player object acquire a collectable item by moving over the point where the item is located. Acquisition of a collectable item will change one or more of the player object's "health", "wealth", or "manna" attributes. Once taken by the player object, collectable items are to be deleted from the game.
- Employ a scheme where single character commands identify directional movements or "magic actions" of the player.

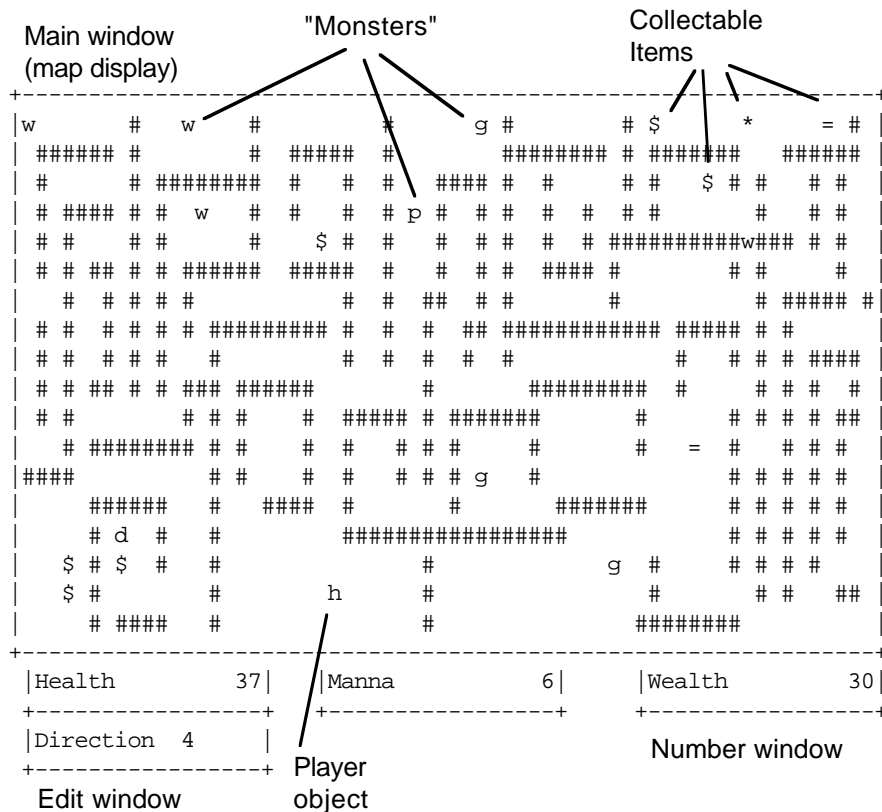


Figure 29.1 "Dungeon" game's display.

- Handle attacks by monster objects on the player. A monster object adjacent to the player will inflict damage proportional to its strength. This amount of damage is deducted from the player object's health rating. Some monster objects have the option of using a projectile weapon when not immediately adjacent to the player.
- Handle attacks by the player on a monster object. A movement command that would place the player object on a point occupied by a monster is to be interpreted as an attack on that monster. The player inflicts a fixed amount of damage. Such an attack reduces the monster's health attribute. If a monster object's health attribute falls to zero, it is to be deleted from the game.
- Handle movements. The player and most types of monster are limited in their scope for movement and cannot pass through walls or outside of the map area. More than one monster may be located on the same point of the map; monsters are allowed to occupy the same points as collectable items. When several dungeon items are located at the same point, only one is shown on the map.

The player's health and manna ratings increase slowly as moves are made.

- Support "magic action" commands. Magic action commands weaken or destroy monsters at a distance from the player; like movement commands, they are directional.

A magic action command inflicts a predefined amount of damage on any monster located on the neighbouring square in the specified direction, half that amount of damage on any monster two squares away along the specified directional axis, a quarter that amount on a monster three squares away etc. Magic does not project through walls.

Use of a magic action command consumes "manna" points. If the player object has insufficient manna points, the player suffers damage equal to twice the deficit. So, if a command requires 8 manna points and the player object's manna is 3, the manna is reduced to zero and the player object's health is reduced by 10 after executing the command.

- Provide the following basic behaviours for monster objects.

A monster object will attack the player object if it is on an adjacent point.

If not immediately adjacent to the player object, some monsters "look" for the player and, if they can "see" the player, they may advance toward it or launch a projectile.

If they are not able to detect the player object, a monster object will perform its "normal movement" function. This might involve random movement, no movement, or some more purposive behaviour.

Monster objects do not attempt to acquire collectable items.

Monster objects do not interact with other monster objects.

29.2 DESIGN

29.2.1 Preliminaries

This "preliminaries" section explores a few aspects of the program that seem pretty much fixed by the specification. The objective is to fill out some details and get a few pointers to things that should be considered more thoroughly.

For example the specification implies the existence of "class Dungeon", "class Player", "class Monster", a class for "collectable items" and so forth. We might as well jot down a few initial ideas about these classes, making a first attempt to answer the perennial questions "*What does class X do? What do instances of class X own?*". Only the most important responsibilities will get identified at this stage; more detailed consideration of specific aspects of the program will result in further responsibilities being added. Detailed analysis later on will also show that some of the classes are interrelated, forming parts of a class hierarchy.

Other issues that should get taken up at this preliminary stage are things like the input files and the form of the main program. Again, they are pretty much defined by the specification, but it is possible to elaborate a little.

main()

We can start with the easy parts – like the `main()` function! This is obviously going to have the basic form "create the principal object, tell it to run":

```
int main()
{
    Dungeon *d;
    d = new Dungeon;

    Prompt user for name of file and read in name
    ...

    d->Load(aName);

    int status = d->Run();
    Terminate(status);

    return 0;
}
```

The principal object is the "Dungeon" object itself. This has to load data from a file and run the game. When the game ends, a message of congratulations or commiserations should be printed. The `Dungeon::Run()` function can return a win/lose flag that can be used to select an appropriate message that is then output by some simple `Terminate()` function.

First idea for files

The files are to be text files, created using some standard editor. They had better specify the size of the map. It would be simplest if the map itself were represented by the '#' and ' ' characters that will get displayed. If the map is too large, the top-left portion should be used.

Following the map, the file should contain the data necessary to define the player, the collectable items, and the monsters. The program should check that these data define exactly one player object and at least one collectable item. The program can simply terminate if data are invalid. (It would help if an error message printed before termination could include some details from the line of input where something invalid was found.)

Collectable items and other objects can be represented in the file using a character code to identify type, followed by whatever number of integers are needed to initialize an object of the specified type. A sentinel character, e.g. 'q', can mark the end of the file.

A plausible form for an input file would be:

```

width and height (e.g 70 20)
several (20) lines of (70) characters, e.g.
##### ... .. #####
#           #           # ... .. # #
# ##### #           # ... .. ##### #
dungeon items
h 30 18 ...          human (i.e. player), coords, other data
w 2 2 10 ...         wandering monster, coords, ...
...
$ 26 6 0 30 0        collectable item, coords, values
q                    end mark

```

Any additional details can be resolved once the objects have been better characterized.

class Dungeon

Consideration of the `main()` function identified two behaviours required of the Dungeon object: loading a file, and running the game.

The `Dungeon::Load()` function will be something along the following lines:

```

Dungeon::Load
    Open file with name given
    Load map
    Load other data

Dungeon::load map
    read size
    loop reading lines of characters that define
        the rows of the map

Dungeon::load other data
    read type character
    while character != 'q'
        create object of appropriate type
        tell it to read its own data
        if it is a monster, add to monster collection
        if it is a collectable, add to collectable
            items collection
        if player, note it (make sure no existing player)

    check that the data defined some items to collect

```

The `Dungeon::Run()` function could have the following general form:

```

Dungeon::Run()
    Finalize setting up of displays
    Draw initial state

    while(player "alive")
        player "run"

```

```

        if(all collectables now taken)
            break;

        for each Monster m in monster collection
            m->Run();

    return (player "alive");

```

The displays must be set up. Obviously, the `Dungeon` object owns some window objects. (Some might belong to the `Player` object; this can be resolved later.) The `Dungeon` object will get primary responsibility for any work needed to set up display structures.

The main loop has two ways of terminating – "death" of player, and all collectable objects taken. The game was won if the player is alive at the end.

The `Dungeon` object owns the collection of monsters, the collection of collectable items, and the player object. Collections could use class `List` or class `DynamicArray`.

The `Player` object will need to access information held by the `Dungeon` object. For example, the `Player` object will need to know whether a particular square is accessible (i.e. not part of a wall), and whether that square is occupied by a collectable item or a monster. When the `Player` takes a collectable item, or kills a monster, the `Dungeon` should be notified so that it can update its records. Similarly, the monsters will be interested in the current position of the `Player` and so will need access to this information.

Consequently, in addition to `Load()` and `Run()`, class `Dungeon` will need many other member functions in its public interface – functions like "Accessible()", and "Remove Monster()". The full set of member functions will get sorted out steadily as different aspects of the game are considered in detail.

Most "dungeon items" will need to interact with the `Dungeon` object in some way or other. It would be best if they all have a `Dungeon*` data member that gets initialized as they are constructed.

class Player

The `Player` object's main responsibility will be getting and interpreting a command entered by the user.

Commands are input as single characters and define either movements or, in this game, directional applications of destructive "magic". The characters 1...9 can be used to define movements. If the keyboard includes a number pad, the convenient mapping is 7 = "north west", 8 = "north", 9 = "north east", 4 = "west" and so forth (where "north" means movement toward the top of the screen and "west" means movement leftwards). Command 5 means "no movement" (sometimes a user may want to delay a little, e.g. to let the player object recover from injury).

The "magic action" commands can use the keys q, w, e, a, d, z, x, and c (on a standard QWERTY keyboard, these have the same relative layout and hence define the same directional patterns as the keys on the numeric keypad).

The main `Player::Run()` function will be something like:

Movement commands

Magic action commands

Player::Run()

```

Player::Run( )
    char ch = GetUserCommand( );
    if( isdigit(ch) ) PerformMovementCommand(ch);
    else PerformMagicCommand(ch);
    UpdateState( );
    ShowStatus( );

```

***Auxiliary private
member functions
used by Run()***

It will involve several auxiliary (private) member functions of class `Player`.

A `GetUserCommand()` function can arrange to read the input. Input is echoed at the current location of the cursor. This could mess up the map display. Consequently it will be necessary to position the cursor prior to reading a command character. This work of cursor positioning and actual data input will involve interactions with window object(s).

A function `UpdateState()` can deal with the business about a `Player` object's health and manna levels increasing. A `ShowStatus()` function can keep the displays current; again this will involve interactions with windows.

The `Perform...` functions will involve interactions with the `Dungeon` object, and possibly other objects as well.

class Collectable

The collectable items could be made instances of a class `Collectable`. It does not seem that there will be any differences in their behaviours, so there probably won't be any specialized subclasses of class `Collectable`. At this stage, it doesn't appear as if `Collectable` objects will do much at all.

They have to draw themselves when asked (presumably by the `Dungeon` object when it is organizing displays); they will own a character that they use to represent themselves. They will also need to own integer values representing the amounts by which they change the `Player` object's health etc when they get taken. Some access functions will have to exist so that the `Player` object can ask for the relevant data.

A monster object moving onto the same point as a `Collectable` object will hide it. When the monster object moves away, the `Collectable` object should be redrawn. The `Dungeon` object had better arrange to get all `Collectable` objects draw themselves at regular intervals; this code could be added to the `while()` loop in `Dungeon::Run()`.

class Monster

As explained in the dungeon game specification, the basic behaviour of a `Monster` is to attack whenever possible, otherwise to advance toward the `Player` when this is possible, otherwise to continue with some "normal action". This behaviour could be defined in the `Monster::Run()` function which would involve a number of auxiliary functions:

Monster::Run()

```

Monster::Run( )
    if( CanAttack( ) )

```

```

        Attack();
    else
        if(CanDetect())
            Advance();
    else
        NormalMove();

```

Different subclasses of class `Monster` can specialize the auxiliary functions so as to vary the basic behaviour. Naturally, these functions will be declared as `virtual`.

Default definitions are possible for some member functions. The default `CanAttack()` function should return true if the `Player` object is adjacent. The default `Attack()` function would tell the `Player` object that it has been hit for a specified number of points of damage. The default implementations for the other functions could all be "do nothing" (i.e. just an empty body `{ }` for `Advance()` and `NormalMove()` and a `return 0` for `CanDetect()`).

*Auxiliary private
member functions
used by Run()*

Checking adjacency will involve getting a pointer to the `Player` object (this can be provided by the `Dungeon` object) and then asking the `Player` for its position. It might be worth having some simple class to represent (x, y) point coordinates. A `Monster` object could have an instance of class `Pt` to represent its position. The `Player` could return its coordinates as an instance of class `Pt`. The first `Pt` could be asked whether it is adjacent to the second.

29.2.2 WindowRep and Window classes

Previous experience with practical windowing systems has influenced the approach developed here for handling the display. As illustrated in Figure 29.2, the display system uses class `WindowRep` and class `Window` (and its specialized subclasses).

WindowRep

Actual communication with the screen is the responsibility of a class `WindowRep` (Window Representation). Class `WindowRep` encapsulates all the sordid details of how to talk to an addressable screen (using those obscure functions, introduced in Chapter 12, like `cgotoxy(x,y,stdout);`). In addition, it is responsible for trying to optimize output to the screen. When the `WindowRep` object gets a request to output a character at a specific point on the screen, it only performs an output operation if the character given is different from that already shown. In order to do this check, the `WindowRep` object maintains a character array in memory that duplicates the information currently on the screen.

The program will have exactly one instance of class `WindowRep`. All "window" objects (or other objects) that want to output (or input) a character will have to interact with the `WindowRep` object. (There can only be one `WindowRep` object in a program because there is only one screen and this screen has to have a unique owner that maintains consistency etc.)

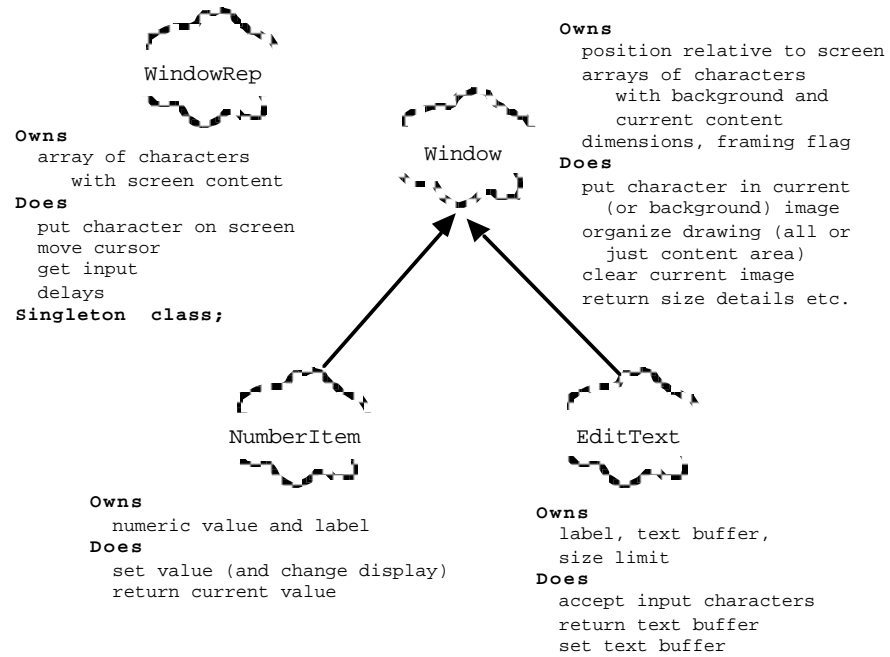


Figure 29.2 Class WindowRep and the Windows class hierarchy.

"Singleton" pattern

A class for which there can only be a single instance, an instance that must be accessible to many other objects, an instance that may have to create auxiliary data structures or perform some specialized hardware initialization – this is a common pattern in programs. A special term "singleton class" has been coined to describe this pattern. There are some standard programming "cliches" used when coding such classes, they are followed in this implementation.

The unique WindowRep object used in a program provides the following services:

- **PutCharacter**
Outputs a character at a point specified in screen coordinates.
- **MoveCursor**
Positions the "cursor" prior to an input operation.
- **GetChar**
Inputs a character, echoing it at the current cursor position
- **Clear**
Clears the entire screen.
- **CloseDown**
Closes down the windowing system and gets rid of the program's WindowRep object

There is another specialized static (class) function, `Instance()`. This handles aspects of the "singleton pattern" (programming cliché) as explained in the implementation (Section 29.3). Essentially, the job of this function is to make certain that there is an instance of class `WindowRep` that is globally available to any other object that needs it (if necessary, creating the program's unique `WindowRep` object).

Window

Window objects, instances of class `Window` or its subclasses, are meant to be things that own some displayable data (an array of characters) and that can be "mapped onto the screen". A `Window` object will have coordinates that specify where its "top-left" corner is located relative to the screen. (In keeping with most cursor-addressable screen usage, coordinate systems are 1-based rather than 0-based so the top left corner of the screen is at coordinate (1,1).) `Window` objects also define a size in terms of horizontal and vertical dimensions. Most `Window` objects are "framed", their perimeters are marked out by '-', '|', and '+' characters (as in Figure 29.1). A `Window` may not fit entirely on the screen (the fit obviously depends on the size and the origin). The `WindowRep` object resolves this by ignoring output requests that are "off screen".

`Window` objects have their own character arrays for their displayable content. Actually, they have two character arrays: a "background array" and a "current array". When told to "draw" itself, a `Window` object executes a function involving a double loop that takes characters from the "current array", works out where each should be located in terms of screen coordinates (taking into account the position of the `Window` object's top-left corner) and requests the `WindowRep` object to display the character at the appropriate point.

*Background and
current (foreground)
window contents*

The "background array" defines the initial contents of the window (possibly all blank). Before a window is first shown on the screen, its current array is filled from the background. A subsequent "clear" operation on a specific point in the window, resets the contents of the current window to be the same as the background at the specified point.

A specific background pattern can be loaded into a window by setting the characters at each individual position. In the `dungeon` game, the `Dungeon` object owns the window used to display the map; it sets the background pattern for that window to be the same as its map layout.

The `Window` class has the following public functions:

*What does a Window
do?*

- **Constructor**
Sets the size and position fields; creates arrays.
- **Destructor**
Gets rid of arrays. (The destructor is `virtual` because class `Window` is to serve as the base class of a hierarchy. In class hierarchies, base classes must always define virtual destructors.)

- `Set, Clear`
Change the character at a single point in the current (foreground) array.
- `SetBkgd`
Change the character at a single point in the background array.
- Access functions: `X, Y, Width, Height`
Return details of data members.
- `PrepareContent`
Initialize current array with copy of background and, if appropriate, add frame.
- `ShowAll, ShowContent`
Output current array via the `WindowRep`.

The class requires a few auxiliary member functions. For example, the coordinates passed to functions like `Set()` must be validated.

What does a Window own?

A `Window` owns its dimension data and its arrays. These data members should be protected; subclasses will require access to these data.

Provision for class hierarchy

The functionality of class `Window` will be extended in its subclasses. However the subclasses don't change the existing functions like `ShowAll()`. Consequently, these functions are not declared as virtual.

"Inheritance for extension" and "inheritance for redefinition"

The relationships between class `Window` and its subclasses, and class `Monster` and its subclasses, are subtly different. The subclasses of `Window` *add* functionality to a working class, but don't change its basic behaviours. Consequently, the member functions of class `Window` are non-virtual (apart from the destructor). Class `Monster` defines a general abstraction; e.g. all `Monster` object can execute some "NormalMove" function, different subclasses redefine the meaning of "NormalMove". Many of the member functions of class `Monster` are declared as virtual so as to permit such redefinition. Apart from the differences with respect to the base class member function being virtual or non-virtual, you will also see differences in the accessibility of additional functions defined by subclasses. When inheritance is being used to extend a base class, many of the new member functions appear in the public interface of the subclass. When inheritance is being used to specialize an existing base class, most of the new functions will be private functions needed to implement changes to some existing behaviour. Both styles, "inheritance for extension" and "inheritance for redefinition", are common.

Subclasses of class `Window`

This program has two subclasses for class `Window`: `NumberItem` and `EditText`. Instances of class `NumberItem` are used to display numeric values; instances of class `EditText` can be used to input commands. As illustrated in Figure 29.3, these are displayed as framed windows that are 3 rows deep by n -columns wide. The left part of the window will normally contain a textual label. The right part is used to display a string representing a (signed) numeric value or as input field where input characters get echoed (in addition to being stored in some data buffer owned by the object).

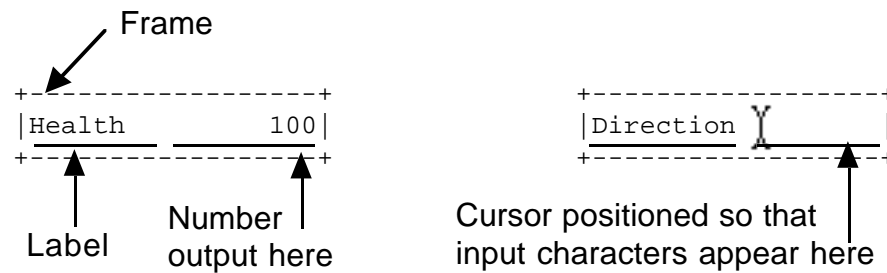


Figure 29.3 NumberItem and EditText Windows

NumberItem

The class declaration for `NumberItem` is:

```
class NumberItem : public Window {
public:
    NumberItem(int x, int y, int width, char *label,
               long initval = 0);
    void    SetVal(long newVal);
    long    GetVal() { return fVal; }
private:
    void    SetLabel(int s, char*);
    void    ShowValue();
    long    fVal;
    int     fLabelWidth;
};
```

In addition to the character arrays and dimension data that a `NumberItem` object already has because it is a kind of `Window`, a `NumberItem` owns a long integer holding the value to be displayed (and, also, an integer to record the number of characters used by the label so that numeric outputs don't overwrite the label).

The constructor for class `NumberItem` completes the normal initialization processes of class `Window`. The auxiliary private member function `SetLabel()` is used to copy the given label string into the background array. The inherited `PrepareContent()` function loads the current array from the background and adds the frame. Finally, using the auxiliary `ShowValue()` function, the initial number is converted into characters that are added to the current image array.

Once constructed, a `NumberItem` can be used by the program. Usually, usage will be restricted to just three functions – `GetVal()` (return `fVal`), `SetVal()` (changes `fVal` and uses `ShowValue()` to update the current image array), and `ShowAll()` (the function, inherited from class `Window`, that gets the window displayed).

*What does a
NumberItem own?*

*What does a
NumberItem do?*

EditText

The primary responsibility of an `EditText` object is getting character input from the user. In the dungeon game program, individual input characters are required as command characters. More generally, the input could be a multicharacter word or a complete text phrase.

An `EditText` object should be asked to get an input. It should be responsible for getting characters from the `WindowRep` object (while moving the cursor around to try to get the echoed characters to appear at a suitable point on the screen). Input characters should be added to a buffer maintained by the `EditText` object. This input process should terminate when either a specified number of characters has been received or a recognizably distinct terminating character (e.g. 'tab', 'enter') is input. (The dungeon program can use such a more generalized `EditText` object by specifying that the input operation is to terminate when a single character has been entered). Often, the calling program will need to know what character terminated the input (or whether it was terminated by a character limit being reached). The input routine can return the terminating character (a '\0' could be returned to indicate that a character count limit was reached).

The `EditText` class would have to provide an access function that lets a caller read the characters in its buffer.

The class declaration for class `EditText` is:

```
class EditText: public Window {
public:
    EditText(int x, int y, int width, char *label, short size);
    void    SetVal(char*);
    char    *GetVal() { return fBuf; }
    char    GetInput();
private:
    void    SetLabel(int s, char*);
    void    ShowValue();
    int     fLabelWidth;
    char    fBuf[256];
    int     fSize;
    int     fEntry;
};
```

What does a `EditText` own?

In addition to the data members inherited from class `Window`, a `EditText` owns a (large) character buffer that can be used to store the input string, integers to record the number of characters entered so far and the limit number. Like the `NumberItem`, an `EditText` will also need a record of the width of its label so that the input field and the label can be kept from overlapping.

What does a `NumberItem` do?

The constructor for class `EditText` completes the normal initialization processes of class `Window`. The auxiliary private member function `SetLabel()` is used to copy the given label string into the background array. The inherited `PrepareContent()` function loads the current array from the background and adds the frame. The buffer, `fBuf`, can be "cleared" (by setting `fBuf[0]` to '\0').

The only member functions used in most programs would be `GetInput()`, `GetVal()` and `ShowAll()`. Sometimes, a program might want to set an initial text string (e.g. a prompt) in the editable field (function `SetVal()`).

29.2.3 DungeonItem hierarchy

Class `Monster` is meant to be an abstraction; the real inhabitants of the dungeon are instances of specialized subclasses of class `Monster`.

Class `Monster` has to provide the following functions (there may be others functions, and the work done in these functions may get expanded later, this list represents an initial guess):

- **Constructor and destructor**
The constructor will set a `Dungeon*` pointer to link back to the `Dungeon` object and set a char data member to the symbol used to represent the `Monster` on the map view.
Since class `Monster` is to be in a hierarchy, it had better define a virtual destructor.
- **Read**
A `Monster` is supposed to read details of its initial position, its "health" and "strength" from an input file. It will need data members to store this information.
- **Access functions** to get position, to check whether "alive", ...
- **A `Run()` function** that as already outlined will work through redefinable auxiliary functions like `CanAttack()`, `Attack()`, `CanDetect()`, `Advance()` and `NormalMove()`.
- **Draw and Erase functions.**
- **A `Move` function.**
Calls `Erase()`, changes coords to new coords given as argument, and calls `Draw()`.
- **GetHit function**
Reduces "health" attribute in accord with damage inflicted by `Player`.

The example implementation has three specializations: `Ghost`, `Patrol`, and `Wanderer`. These classes redefine member functions from class `Monster` as needed.

A `Ghost` is a `Monster` that:

class `Ghost`

- uses the default "do nothing" implementation defined by `Monster::NormalMove()` along with the standard `CanAttack()`, `Attack()` functions;
- has a `CanDetect()` function that returns true when the player is within a fixed distance of the point where the `Ghost` is located (the presence of intervening walls makes no difference to a `Ghost` object's power of detection);

- has an `Advance()` function that moves the `Ghost` one square vertically, diagonally, or horizontally so as to advance directly toward the player; a `Ghost` can move through walls;
- has a high initial "health" rating;
- inflicts only a small amount of damage when attacking the `Player`.

Class `Ghost` needs to redefine only the `Advance()` and `CanDetect()` functions. Since a `Ghost` does not require any additional data it does not to change the `Read()` function.

class Patrol A `Patrol` is a `Monster` that:

- uses the default `CanAttack()`, `Attack()` functions to attack an adjacent player;
- has a `CanDetect()` function that returns true there is a clear line of sight between it and the `Player` object;
- has an `Advance()` function that instead of moving it toward the `Player` allows it to fire a projectile that follows the "line of sight" path until it hits the `Player` (causing a small amount of damage), the movement of the projectile should appear on the screen;
- has a `NormalMove()` function that causes it to follow a predefined patrol route (it never departs from this route so it does not attempt to pursue the `Player`).
- has a moderate initial "health" rating;
- inflicts a large amount of damage when making a direct attack on an adjacent player.

The patrol route should be defined as a sequence of points. These will have to be read from the input file and so class `Patrol` will need to extend the `Monster::Read()` function. The `Patrol::Read()` function should check that the given points are adjacent and that all are accessible (within the bounds of the dungeon and not blocked by walls).

Class `Patrol` will need to define extra data members to hold the route data. It will need an array of `Pt` objects (this can be a fixed sized array with some reasonable maximum length for a patrol route), an integer specifying the number of points in the actual route, an integer (index value) specifying which `Pt` in the array the `Patrol` is currently at, and another integer to define the direction that the `Patrol` is walking. (The starting point given for the `Patrol` will be the first element of the array. Its initial moves will cause it to move to successive elements of the `Pt` array; when it reaches the last, it can retrace its path by having the index decrease through the array.)

class Wanderer A `Wanderer` is a `Monster` that:

- uses the default `CanAttack()`, `Attack()` functions to attack an adjacent player;

- has a `CanDetect()` function that returns true there is a clear line of sight between it and the `Player` object;
- has an `Advance()` function that causes the `Wanderer` to move one step along the line of sight path toward the current position of the `Player`;
- has a `NormalMove()` function that causes it to try to move in a constant direction until blocked by a wall, when its movement is blocked, it picks a new direction at random;
- has a small initial "health" rating;
- inflicts a moderate amount of damage when making a direct attack on an adjacent player.

A `Wanderer` will need to remember its current direction of movement so that it can keep trying to go in the same direction. Integer data members, representing the current delta-x, delta-y changes of coordinate, could be used.

There are similarities between the `Monster` and `Player` classes. Both classes define things that have a `Pt` coordinate, a health attribute and a strength attribute. There are similarities in behaviours: both the `Player` and the `Monsters` read initial data from file, get told that they have been hit, get asked whether they are still alive, get told to draw themselves (and erase themselves), have a "move" behaviour that involves erasing their current display, changing their `Pt` coordinate and then redrawing themselves. These commonalities are sufficient to make it worth defining a new abstraction, "active item", that subsumes both the `Player` and the `Monsters`.

*Commonalities
between class `Player`
and class `Monster`*

This process of abstracting out commonalities can be repeated. There are similarities between class `Collectable` and the new class `ActiveItem`. Both are things with `Pt` coordinates, draw and erase behaviours; they respond to queries about where they are (returning their `Pt` coordinate). These common behaviours can be defined in a base class: `DungeonItem`.

The `DungeonItem` class hierarchy used in the implementation is shown in Figure 29.4.

29.2.3 Finalising the classes

Completion of the design stage involves coming up with the class declarations of all the classes, possibly diagramming some of the more complex patterns of interaction among instances of different classes, and developing algorithms for any complicated functions.

Class `Dungeon`

The finalised declaration for class `Dungeon` is:

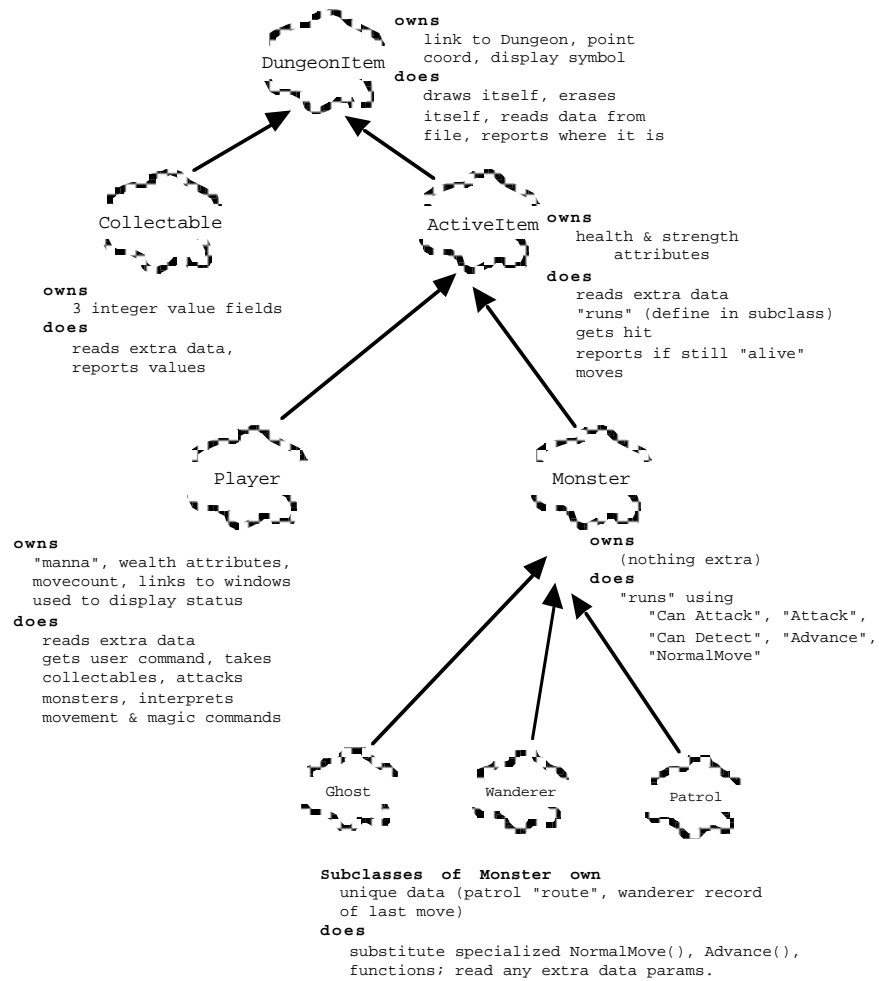


Figure 29.4 DungeonItem class hierarchy.

```

class Dungeon {
public:
    Dungeon();
    ~Dungeon();
    void Load(const char filename[]);
    int Run();
    int Accessible(Pt p) const;
    Window *Display();
    Player *Human();

    int ValidPoint(Pt p) const;

    Monster *M_at_Pt(Pt p);
    Collectable *PI_at_Pt(Pt p);
    void RemoveProp(Collectable *pi);
    void RemoveM(Monster *m);
};

```

```

        int      ClearLineOfSight(Pt p1, Pt p2, int max,
                                   Pt path[]);

private:
    int      ClearRow(Pt p1, Pt p2, int max, Pt path[]);
    int      ClearColumn(Pt p1, Pt p2, int max, Pt path[]);
    int      ClearSemiVertical(Pt p1, Pt p2, int max,
                               Pt path[]);
    int      ClearSemiHorizontal(Pt p1, Pt p2, int max,
                                  Pt path[]);
    void      LoadMap(istream& in);
    void      PopulateDungeon(istream& in);
    void      CreateWindow();

    DynamicArray  fProps;
    DynamicArray  fInhabitants;
    Player        *fPlayer;

    char          fDRep[MAXHEIGHT][MAXWIDTH];
    Window        *fDWindow;
    int           fHeight;
    int           fWidth;
};

```

The Dungeon object owns the map of the maze (represented by its data elements `fDRep[][], fHeight`, and `fWidth`). It also owns the main map window (`fDWindow`), the Player object (`fPlayer`) and the collections of Monsters and Collectables. Data members that are instances of class `DynamicArray` are used for the collections (`fInhabitants` for the Monsters, `fProps` for the Collectables).

*What does a
Dungeon own?*

The `Load()` and `Run()` functions are used by the main program. Function `Load()` makes use of the auxiliary private member functions `LoadMap()` and `PopulateDungeon()`; these read the various data and create `Monster`, `Collectable`, and `Player` object(s) as specified by the input. The auxiliary private member function `CreateWindow()` is called from `Run()`; it creates the main window used for the map and sets its background from information in the map.

*What does a
Dungeon do?*

Access functions like `Display()` and `Human()` allow other objects to get pointers to the main window and the Player object. The `ActiveItem` objects that move are going to need access to the main Window so as to tell it to clear and set the character that is to appear at a particular point.

The `ValidPoint()` function checks whether a given `Pt` is within the bounds of the maze and is not a "wall".

The functions `M_at_Pt()` and `PI_at_Pt()` involve searches through the collections of Monsters and Collectables respectively. These functions return the first member of the collection present at a `Pt` (or `NULL` if there are no objects at that `Pt`). The `Remove...` function eliminates members of the collections.

Class `Dungeon` has been given responsibility for checking whether a "clear line of sight" exists between two `Pts` (this function is called in both `Wanderer::CanDetect()` and `Patrol::CanDetect()`). The function takes as arguments the two points, a maximum range and a `Pt` array in which to return the `Pts` along the

line of sight. Its implementation uses the auxiliary private member functions `ClearRow()` etc.

The algorithm for the `ClearLineOfSight()` function is the most complex in the program. There are two easy cases; these occur when the two points are in the same row or the same column. In such cases, it is sufficient to check each point from start to end (or out to a specified maximum) making certain that the line of sight is not blocked by a wall. Pseudo code for the `ClearRow()` function is:

```
Dungeon::ClearRow(Pt p1, Pt p2, int max, Pt path[])
    delta = if p1 left of p2 then 1 else -1
    current point = p1
    for i < max do
        current point's x += delta;
        if(current point is not accessible) return fail
        path[i] = current point;
        if(current point equal p2) return success
        i++
    return fail
```

Cases where the line is oblique are a bit more difficult. It is necessary to check the squares on the map (or screen) that would be traversed by the best approximation to a straight line between the points. There is a standard approach to solving this problem; Figure 29.5 illustrates the principle.

The squares shown by dotted lines represent the character grid of the map or screen; they are centred on points defined by integer coordinates. The start point and end point are defined by integer coordinates. The real line between the points has to be approximated by a sequence of segments joining points defined by integer coordinates. These points define which grid squares are involved. In Figure 29.5 the squares traversed are highlighted by • marks.

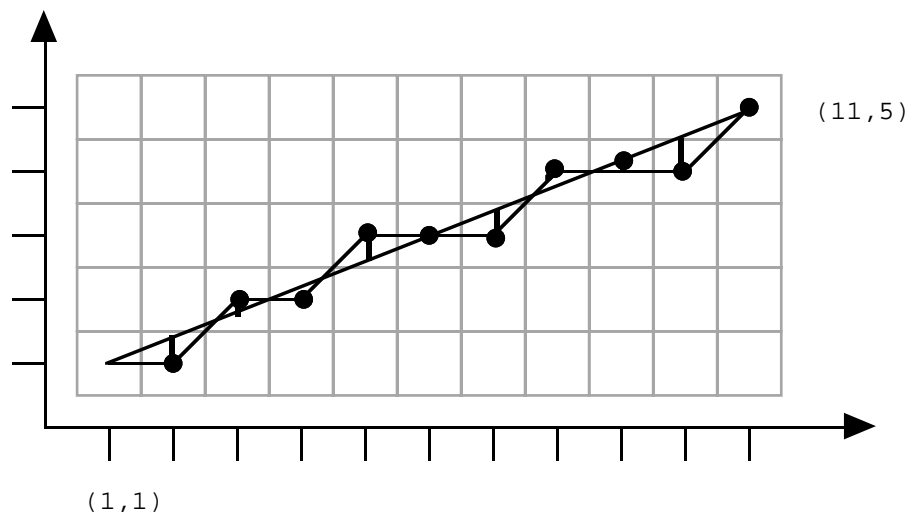


Figure 29.5 A digitized line.

The algorithm has to choose the best sequence of integer points, for example choosing point (2, 1) rather than (2, 2) and (8, 4) rather than (8, 3) or (8, 5). The algorithm works by calculating the error associated with different points (shown by the vertical lines in Figure 29.5). The correct y value for say x=8 can be calculated; the errors of taking y = 3, or 4, or 5 are easy to calculate and the best approximation is chosen. When the squares traversed have been identified, they can be checked to determine that they are not blocked by walls and, if clear, added to the path array.

The algorithm is easiest to implement using two separate versions for the cases where the change in x is larger or the change in y is larger. A pseudo-code outline for the algorithm for where the change in x is larger is as follows:

```
Dungeon::ClearSemiHorizontal(Pt p1, Pt p2, int max,
    Pt path[])
    ychange = difference in y values of two points
    xchange = difference in x values of two points
    if(xchange > max)
        return fail

    deltax = if x increasing then 1 else -1
    deltay = if y increasing then 1 else -1

    slope = change in y divided by change in x
    error = slope*deltax

    current point = p1
    for i < abs(xchange) do
        if(error*deltay>0.5)
            current point y += deltay
            error -= deltay
            Pick best next point

        error += slope*deltax
        current point x += deltax
        if(current point not accessible) return fail
        Check accessibility

        path[i] = current point
        if(current point equal p2) return success
        i++
        Add to path
    return fail
```

Class Pt

It is worth introducing a simple class to represent coordinates because many of the functions need coordinates as arguments, and there are also several places in the code where it is necessary to compare the coordinates of different objects. A declaration for class Pt is:

```
class Pt {
public:
    Pt(int x = 0, int y = 0);
    int    X() const;
    int    Y()    const;
    void    SetPt(int newx, int newy);
```

```

        void    SetPt(const Pt& other);
        int     Equals(const Pt& other) const;
        int     Adjacent(const Pt& other) const;
        int     Distance(const Pt& other) const;
    private:
        int     fx;
        int     fy;
};

```

Most of the member functions of `Pt` are sufficiently simple that they can be defined as inline functions.

Classes `WindowRep` and `Window`

The declaration for class `WindowRep` is

```

class WindowRep {
public:
    static WindowRep *Instance();
    void    CloseDown();
    void    PutCharacter(char ch, int x, int y);
    void    Clear();
    void    Delay(int seconds) const;
    char    GetChar();
    void    MoveCursor(int x, int y);
private:
    WindowRep();
    void    Initialize();
    void    PutCharacter(char ch);
    static WindowRep *sWindowRep;
    char    fImage[CG_HEIGHT][CG_WIDTH];
};

```

The size of the image array is defined by constants `CG_HEIGHT` and `CG_WIDTH` (their values are determined by the area of the cursor addressable screen, typically up to 24 high, 80 wide).

The static member function `Instance()`, and the static variable `sWindowRep` are used in the implementation of the "singleton" pattern as explained in the implementation section. Another characteristic of the singleton nature is the fact that the constructor is private; a `WindowRep` object can only get created via the public `Instance()` function.

Most of the members of class `Window` have already been explained. The actual class declaration is:

```

class Window {
public:
    Window(int x, int y, int width, int height,
           char bkgd = ' ', int framed = 1);
    virtual ~Window();
    void    Clear(int x, int y);
    void    Set(int x, int y, char ch);
    void    SetBkgd(int x, int y, char ch);

```

```

        int      X() const;
        int      Y() const;
        int      Width() const;
        int      Height() const;

        void      ShowAll() const;
        void      ShowContent() const;
        void      PrepareContent();
    protected:
        void      Change(int x, int y, char ch);
        int      Valid(int x, int y) const;
        char      Get(int x, int y, char **img) const;
        void      SetFrame();
        char      **fBkgd;
        char      **fCurrentImg;
        int      fX;
        int      fY;
        int      fWidth;
        int      fHeight;
        int      fFramed;
};

```

The declarations for the specialized subclasses of class Window were given earlier.

DungeonItem class hierarchy

The base class for the hierarchy defines a `DungeonItem` as something that can read its data from an input stream, can draw and erase itself, and can say where it is. It owns a link to the `Dungeon` object, its `Pt` coordinate and a symbol.

```

class DungeonItem {
public:
    DungeonItem(Dungeon *d, char sym);
    virtual ~DungeonItem();
    Pt          Where() const;
    virtual void Draw();
    virtual void Read(istream& in);
    virtual void Erase();
protected:
    Dungeon     *fD;
    Pt          fPos;
    char        fSym;
};

```

DungeonItem

Since class `DungeonItem` is the base class in a hierarchy, it provides a virtual destructor and makes its data members protected (allowing access by subclasses).

A `Collectable` object is just a `DungeonItem` with three extra integer data members and associated access functions that can return their values. Because a `Collectable` needs to read the values of its extra data members, the class redefines the `DungeonItem` read function.

```
Collectable    class Collectable : public DungeonItem {
                public:
                    Collectable(Dungeon* d, char sym);
                    int          Hlth();
                    int          Wlth();
                    int          Manna();
                    virtual void  Read(ifstream& in);
                private:
                    int          fHval;
                    int          fWval;
                    int          fMval;
            };

```

An `ActiveItem` is a `DungeonItem` that gets hit, gets asked if is alive, moves, and runs. Member function `Run()` is pure virtual, it has to be redefined in subclasses (because the "run" behaviours of subclasses `Player` and `Monster` are quite different). Default definitions can be provided for the other member functions. All `ActiveItem` objects have "strength" and "health" attributes. The inherited `DungeonItem::Read()` function will have to be extended to read these extra data.

```
ActiveItem    class ActiveItem : public DungeonItem {
                public:
                    ActiveItem(Dungeon *d, char sym);
                    virtual void  Read(ifstream& in);
                    virtual void  Run() = 0;
                    virtual void  GetHit(int damage);
                    virtual int    Alive() const;
                protected:
                    virtual void  Move(const Pt& newpoint);
                    Pt            Step(int dir);
                    int            fHealth;
                    int            fStrength;
            };

```

(Function `Step()` got added during implementation. It returns the coordinate of the adjacent point as defined by the `dir` parameter; 7 => north west neighbor, 8 => north neighbor etc.)

A `Player` is a specialized `ActiveItem`. The class has one extra public member function, `ShowStatus()`, and several additional private member functions that are used in the implementation of its definition of `Run()`. A `Player` has extra data members for its wealth and manna attributes, a move counter that gets used when updating health and manna. The `Player` object owns the `NumberItem` and `EditText` windows that are used to display its status and get input commands.

```
Player        class Player : public ActiveItem {
                public:
                    Player(Dungeon* d);
                    virtual void  Run();
                    virtual void  Read(ifstream& in);
                    void          ShowStatus();
                private:
                    void          TryMove(int newx, int newy);
            };

```

```

    void    Attack(Monster *m);
    void    Take(Collectable *pi);
    void    UpdateState();
    char    GetUserCommand();
    void    PerformMovementCommand(char ch);
    void    PerformMagicCommand(char ch);

    int      fMoveCount;
    int      fWealth;
    int      fManna;
    NumberItem *fWinH;
    NumberItem *fWinW;
    NumberItem *fWinM;
    EditText *fWinE;
};

```

Class `Monster` is just an `ActiveItem` with a specific implementation of `Run()` that involves the extra auxiliary functions `CanAttack()` etc. Since instances of specialized subclasses are going to be accessed via `Monster*` pointers, and there will be code of the form `Monster *m; ... ; delete m;`, the class should define a virtual destructor.

```

class Monster : public ActiveItem {
public:
    Monster(Dungeon* d, char sym);
    virtual ~Monster();
    virtual void Run();
protected:
    virtual int CanAttack();
    virtual void Attack();
    virtual int CanDetect();
    virtual void Advance();
    virtual void NormalMove() { }
};

```

Monster

Class `Ghost` defines the simplest specialization of class `Monster`. It has no extra data members. It just redefines the default (do nothing) `CanDetect()` and `Advance()` member functions.

```

class Ghost : public Monster {
public:
    Ghost(Dungeon *d);
protected:
    virtual int CanDetect();
    virtual void Advance();
};

```

Ghost

The other two specialized subclasses of `Monster` have additional data members. In the case of class `Patrol`, these must be initialized from input so the `Read()` function is redefined. Both classes redefine `Normal Move()` as well as `CanDetect()` and `Advance()`.

```

class Wanderer : public Monster {
public:
    Wanderer(Dungeon *d);
protected:
    virtual void NormalMove();
    virtual int CanDetect();
    virtual void Advance();
    int fLastX, fLastY;
    Pt fPath[20];
};

class Patrol : public Monster {
public:
    Patrol(Dungeon *d);
    virtual void Read(istream& in);
protected:
    virtual void NormalMove();
    virtual int CanDetect();
    virtual void Advance();
    Pt fPath[20];
    Pt fRoute[100];
    int fRouteLen;
    int fNdx, fDelta;
};

```

Object Interactions

Figure 29.6 illustrates some of the interactions involved among different objects during a cycle of `Dungeon::Run()`.

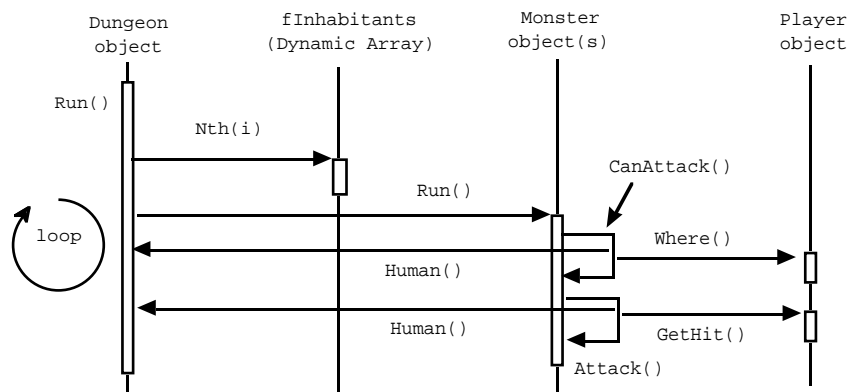


Figure 29.6 Some of the object interactions in `Dungeon::Run`.

The diagram illustrates aspects of the loop where each `Monster` object in the `fInhabitants` collection is given a chance to run. The `Dungeon` object will first interact with the `DynamicArray` `fInhabitants` to get a pointer to a particular `Monster`. This will then be told to run; its `Run()` function would call its `CanAttack()` function.

In `CanAttack()`, the `Monster` would have to get details of the `Player` object's position. This would involve first a call to member function `Human()` of the `Dungeon` object to get a pointer, and then a call to `Where()` of the `Player` object.

The diagram in Figure 29.7 illustrates the case where the `Player` is adjacent and the `Monster` object's `Attack()` function is called. This will again involve a call to `Dungeon::Human()`, and then a call to the `GetHit()` function of the `Player` object.

Figure 29.7 illustrates some of the interactions that might occur when the a movement command is given to `Player::Run()`.

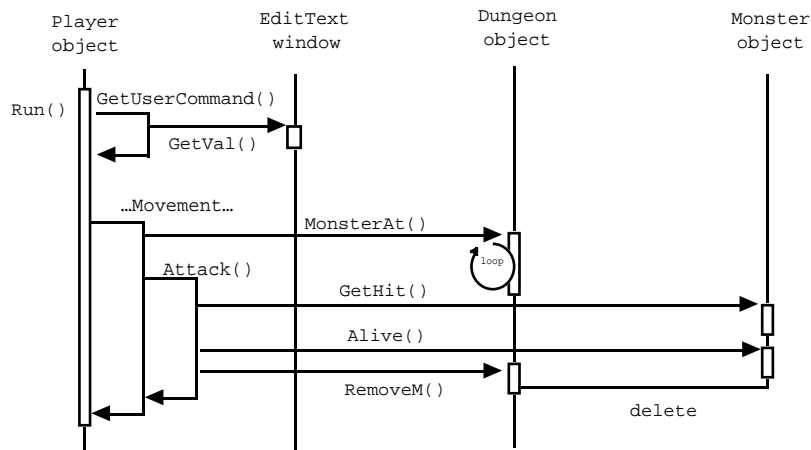


Figure 29.7 Object interactions during `Player::Run`.

The `Player` object would have had to start by asking its `EditText` window to get input. When `EditText::GetInput()` returns, the `Player` object could inspect the character string entered (a call to the `EditText`'s `GetVal()` function, not shown in diagram). If the character entered was a digit, the `Player` object's function `PerformMovementCommand` would be invoked. This would use `Step()` to determine the coordinates of the adjacent `Pt` where the `Player` object was to move. The `Player` would have to interact with the `Dungeon` object to check whether the destination point was occupied by a `Monster` (or a `Collectable`).

The diagram in Figure 29.7 illustrates the case where there is an adjacent `Monster`. The `Player` object informs the `Monster` that it has been hit. Then it checks whether the `Monster` is still alive. In the illustration, the `Monster` object has been destroyed, so the `Player` must again interact with the `Dungeon` object. This removes the `Monster` from the `fInhabitants` list (interaction with the `DynamicArray` is not shown) and deletes the `Monster`.

29.3 AN IMPLEMENTATION

The files used in the implementation, and their interdependencies are summarized in Figure 29.8.

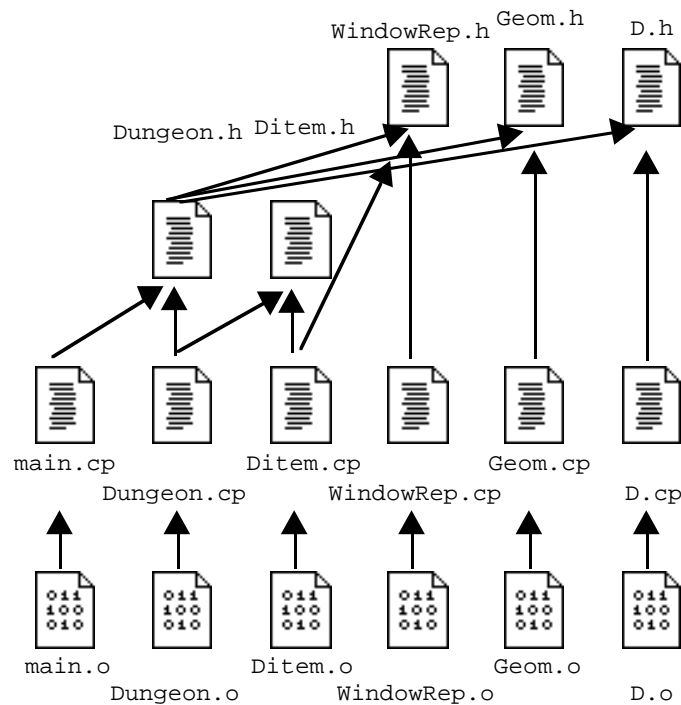


Figure 29.8 Module structure for Dungeon game example.

The files `D.h` and `D.cp` contain the `DynamicArray` code defined in Chapter 21. The `Geom` files have the definition of the simple `Pt` class. The `WindowRep` files contain `WindowRep`, `Window` and its subclasses. `Ditem.h` and `Ditem.cp` contain the declaration and definition of the classes in the `DungeonItem` hierarchy while the `Dungeon` files contain class `Dungeon`.

An outline for `main()` has already been given; function `Terminate()`, which prints an appropriate "you won" or "you lost" message, is trivial.

29.3.1 Windows classes

There are two aspects to class `WindowRep`: its "singleton" nature, and its interactions with a cursor addressable screen.

The constructor is private. `WindowRep` objects cannot be created by client code (we only want one, so we don't want to allow arbitrary creation). Clients (like the code of class `Dungeon`) always access the unique `WindowRep` object through the static member function `Instance()`.

```
WindowRep *WindowRep::Instance()
{
    if (sWindowRep == NULL)
        sWindowRep = new WindowRep;
```

```

        return sWindowRep;
    }

    WindowRep *WindowRep::sWindowRep = NULL;

```

The first time that it is called, function `Instance()` creates the `WindowRep` object; subsequent calls always return a pointer to the same object.

The constructor calls function `Initialize()` which performs any system dependent device initialization. It then sets up the image array, and clears the screen.

```

WindowRep::WindowRep()
{
    Initialize();
    for(int row = 0; row < CG_HEIGHT; row++)
        for(int col = 0; col < CG_WIDTH; col++)
            fImage[row][col] = ' ';
    Clear();
}

```

The implementation of functions like `Initialize()` involves the same system dependent calls as outlined in the "Cursor Graphics" examples in Chapter 12. Some example functions are:

```

void WindowRep::Initialize()
{
    #if defined(SYMANTEC)
    /*
    Have to change the "mode" for the 'console' screen.
    Putting it in C_CBREAK allows characters to be read one by
    one as they are typed
    */
        csetmode(C_CBREAK, stdin);
    #else
    /*
    No special initializations are needed for Borland IDE
    */
    #endif
}

void WindowRep::MoveCursor(int x, int y)
{
    {
        if((x<1) || (x>CG_WIDTH)) return;
        if((y<1) || (y>CG_HEIGHT)) return;

        #if defined(SYMANTEC)
            cgotoxy(x,y,stdout);
        #else
            gotoxy(x,y);
        #endif
    }
}

void WindowRep::PutCharacter(char ch)
{

```

```

    #if defined(SYMANTEC)
        fputc(ch, stdout);
        fflush(stdout);
    #elif
        putchar(ch);
    #endif
}

```

Functions like `WindowRep::Delay()` and `WindowRep::GetChar()` similarly repackage code from "Cursor Graphics" example.

The `WindowRep::PutCharacter()` function only does the cursor movement and character output operations when necessary. This function also keeps the `WindowRep` object's image array consistent with the screen.

```

void WindowRep::PutCharacter(char ch, int x, int y)
{
    if((x<1) || (x>CG_WIDTH)) return;
    if((y<1) || (y>CG_HEIGHT)) return;

    if(ch != fImage[y-1][x-1]) {
        MoveCursor(x,y);
        PutCharacter(ch);
        fImage[y-1][x-1] = ch;
    }
}

```

The `CloseDown()` function clears the screen, performs any device specific termination, then after a short delay lets the `WindowRep` object self destruct.

```

void WindowRep::CloseDown()
{
    Clear();
    #if defined(SYMANTEC)
        csetmode(C_ECHO, stdin);
    #endif
    sWindowRep = NULL;
    Delay(2);
    delete this;
}

```

Window

The constructor for class `Window` initializes the simple data members like the width and height fields. The foreground and background arrays are created. They are vectors, each element of which represents an array of characters (one row of the image).

```

Window::Window(int x, int y, int width, int height,
               char bkgd, int framed )
{
    fX = x-1;
    fY = y-1;
}

```

```

        fWidth = width;
        fHeight = height;
        fFramed = framed;

        fBkgd = new char* [height];
        fCurrentImg = new char* [height];
        for(int row = 0; row < height; row++) {
            fBkgd[row] = new char[width];
            fCurrentImg[row] = new char[width];
            for(int col = 0; col < width; col++)
                fBkgd[row][col] = bkgd;
        }
    }
}

```

Naturally, the main task of the destructor is to get rid of the image arrays:

```

Window::~~Window()
{
    for(int row = 0; row < fHeight; row++) {
        delete [] fCurrentImg[row];
        delete [] fBkgd[row];
    }
    delete [] fCurrentImg;
    delete [] fBkgd;
}

```

Functions like `Clear()`, and `Set()` rely on auxiliary routines `Valid()` and `Change()` to organize the real work. Function `Valid()` makes certain that the coordinates are within the window's bounds. Function `Change()` is given the coordinates, and the new character. It looks after details like making certain that the window frame is not overwritten (if this is a framed window), arranging for a request to the `WindowRep` object asking for the character to be displayed, and the updating of the array.

```

void Window::Clear(int x, int y)
{
    if(Valid(x,y))
        Change(x,y,Get(x,y,fBkgd));
}

void Window::Set(int x, int y, char ch)
{
    if(Valid(x,y))
        Change(x, y, ch);
}

```

(Function `Change()` has to adjust the `x, y` values from the 1-based scheme used for referring to screen positions to a 0-based scheme for C array subscripting.)

```

void Window::Change(int x, int y, char ch)
{
    if(fFramed) {
        if((x == 1) || (x == fWidth)) return;
        if((y == 1) || (y == fHeight)) return;
    }
}

```

```

    }

    WindowRep::Instance()->PutCharacter(ch, x + fX, y + fY);
    x--;
    y--;
    fCurrentImg[y][x] = ch;
}

```

Note the call to `WindowRep::Instance()`. This returns a `WindowRep*` pointer. The `WindowRep` referenced by this pointer is then asked to output the character at the specified point as offset by the origin of this window.

Function `SetBkgd()` simply validates the coordinate arguments and then sets a character in the background array. Function `Get()` returns the character at a particular point in either background or foreground array (an example of its use is in the statement `Get(x, y, fBkgd)` in `Window::Clear()`).

```

char Window::Get(int x, int y, char **img) const
{
    x--;
    y--;
    return img[y][x];
}

```

Function `PrepareContent()` loads the current image array from the background and, if appropriate, calls `SetFrame()` to add a frame.

```

void Window::PrepareContent()
{
    for(int row = 0; row < fHeight; row++)
        for(int col = 0; col < fWidth; col++)
            fCurrentImg[row][col] = fBkgd[row][col];
    if(fFramed)
        SetFrame();
}

void Window::SetFrame()
{
    for(int x=1; x<fWidth-1; x++) {
        fCurrentImg[0][x] = '-';
        fCurrentImg[fHeight-1][x] = '-';
    }
    for(int y=1; y < fHeight-1; y++) {
        fCurrentImg[y][0] = '|';
        fCurrentImg[y][fWidth-1] = '|';
    }
    fCurrentImg[0][0] = '+';
    fCurrentImg[0][fWidth-1] = '+';
    fCurrentImg[fHeight-1][0] = '+';
    fCurrentImg[fHeight-1][fWidth-1] = '+';
}

```

A window object's frame uses its top and bottom rows and leftmost and rightmost columns. The content area, e.g. the map in the dungeon game, cannot use these

perimeter points. (The input file for the map could define the perimeter as all "wall".)

The access functions like `X()`, `Y()`, `Height()` etc are all trivial, e.g.:

```
int Window::X() const
{
    return fX;
}
```

The functions `ShowAll()` and `ShowContent()` are similar. They have loops take characters from the current image and send the to the `WindowRep` object for display. The only difference between the functions is in the loop limits; function `ShowContent()` does not display the periphery of a framed window.

```
void Window::ShowAll() const
{
    for(int row=1;row<=fHeight; row++)
        for(int col = 1; col <= fWidth; col++)
            WindowRep::Instance()->
                PutCharacter(
                    fCurrentImg[row-1][col-1],
                    fX+col, fY+row);
}
```

NumberItem and EditText

The only complications in class `NumberItem` involve making certain that the numeric value output does not overlap with the label. The constructor checks the length of the label given and essentially discards it if display of the label would use too much of the width of the `NumberItem`.

```
NumberItem::NumberItem(int x, int y, int width, char *label,
    long initval) : Window(x, y, width, 3)
{
    fVal = initval;
    fLabelWidth = 0;
    int s = strlen(label);
    if((s > 0) && (s < (width-5)))
        SetLabel(s, label);
    PrepareContent();
    ShowValue();
}
```

(Note how arguments are passed to the base class constructor.)

Function `SetLabel()` copies the label into the left portion of the background image. Function `SetVal()` simply changes the `fVal` data member then calls `ShowValue()`.

```
void NumberItem::SetLabel(int s, char * l)
{
    fLabelWidth = s;
```

```

        for(int i=0; i< s; i++)
            fBkgd[1][i+1] = l[i];
    }

```

Function `ShowValue()` starts by clearing the area used for number display. A loop is then used to generate the sequence of characters needed, these fill in the display area starting from the right. Finally, a sign is added. (If the number is too large to fit into the available display area, a set of hash marks are displayed.)

```

void NumberItem::ShowValue()
{
    int left = 2 + fLabelWidth;
    int pos = fWidth - 1;
    long val = fVal;
    for(int i = left; i<= pos; i++)
        fCurrentImg[1][i-1] = ' ';
    if(val<0) val = -val;
    if(val == 0)
        fCurrentImg[1][pos-1] = '0';
    while(val > 0) {
        int d = val % 10;
        val = val / 10;
        char ch = d + '0';
        fCurrentImg[1][pos-1] = ch;
        pos--;
        if(pos <= left) break;
    }
    if(pos<=left)
        for(i=left; i<fWidth;i++)
            fCurrentImg[1][i-1] = '#';
    else
        if(fVal<0)
            fCurrentImg[1][pos-1] = '-';
    ShowContent();
}

```

Class `EditText` adopts a similar approach to dealing with the label, it is not shown if it would use too large a part of the window's width. The contents of the buffer have to be cleared as part of the work of the constructor (it is sufficient just to put a null character in the first element of the buffer array).

```

EditText::EditText(int x, int y, int width, char *label,
    short size) : Window(x, y, width, 3)
{
    fSize = size;
    fLabelWidth = 0;
    int s = strlen(label);
    if((s > 0) && (s < (width-8)))
        SetLabel(s, label);
    PrepareContent();
    fBuf[0] = '\0';
    ShowValue();
}

```

The `SetLabel()` function is essentially the same as that of class `NumberItem`. The `SetVal()` function loads the buffer with the given string (taking care not to overfill the array).

```
void EditText::SetVal(char* val)
{
    int n = strlen(val);
    if(n>254) n = 254;
    strncpy(fBuf, val, n);
    fBuf[n] = '\0';
    ShowValue();
}
```

The `ShowValue()` function displays the contents of the buffer, or at least that portion of the buffer that fits into the window width.

```
void EditText::ShowValue()
{
    int left = 4 + fLabelWidth;
    int i, j;
    for(i=left; i<fWidth; i++)
        fCurrentImg[1][i-1] = ' ';
    for(i=left, j=0; i<fWidth; i++, j++) {
        char ch = fBuf[j];
        if(ch == '\0') break;
        fCurrentImg[1][i-1] = ch;
    }
    ShowContent();
}
```

Function `GetInput()` positions the cursor at the start of the data entry field then loops accepting input characters (obtained via the `WindowRep` object). The loop terminates when the required number of characters has been obtained, or when a character like a space or tab is entered.

```
char EditText::GetInput()
{
    int left = 4 + fLabelWidth;
    fEntry = 0;
    ShowValue();
    WindowRep::Instance()->MoveCursor(fX+left, fY+2);
    char ch = WindowRep::Instance()->GetChar();
    while(isalnum(ch)) {
        fBuf[fEntry] = ch;
        fEntry++;
        if(fEntry == fSize) {
            ch = '\0';
            break;
        }
        ch = WindowRep::Instance()->GetChar();
    }
    fBuf[fEntry] = '\0';
    return ch;
}
```


The function does not prevent entry of long strings from overwriting parts of the screen outside of the supposed window area. You could have a more sophisticated implementation that "shifted existing text leftwards" so that display showed only the last few characters entered and text never went beyond the right margin.

29.3.2 class Dungeon

The constructor and destructor for class `Dungeon` are limited. The constructor will simply involve initializing pointer data members to `NULL`, while the destructor should delete "owned" objects like the main display window.

The `Load()` function will open the file, then use the auxiliary `LoadMap()` and `PopulateDungeon()` functions to read the data.

```
void Dungeon::Load(const char filename[])
{
    ifstream in(filename, ios::in | ios::nocreate);
    if(!in.good()) {
        cout << "File does not exist. Quitting." << endl;
        exit(1);
    }
    LoadMap(in);
    PopulateDungeon(in);
    in.close();
}
```

The `LoadMap()` function essentially reads "lines" of input. It will have to discard any characters that don't fit so will be making calls to `ignore()`. The argument `END_OF_LINE_CHAR` would normally be `'\n'` but some editors use `'\r'`.

```
const int END_OF_LINE_CHAR = '\r';

void Dungeon::LoadMap(ifstream& in)
{
    in >> fWidth >> fHeight;
    in.ignore(100, END_OF_LINE_CHAR);
    for(int row = 1; row <= fHeight; row++ ) {
        char ch;
        for(int col = 1; col <= fWidth; col++) {
            in.get(ch);
            if((row<=MAXHEIGHT) && (col <= MAXWIDTH))
                fDRep[row-1][col-1] = ch;
        }
        in.ignore(100, END_OF_LINE_CHAR);
    }

    if(!in.good()) {
        cout << "Sorry, problems reading that file. "
              "Quitting." << endl;
        exit(1);
    }
}
```

```

    cout << "Dungeon map read OK" << endl;

    if((fWidth > MAXWIDTH) || (fHeight > MAXHEIGHT)) {
        cout << "Map too large for window, only using "
            "part of map." << endl;
        fWidth = (fWidth < MAXWIDTH) ? fWidth : MAXWIDTH;
        fHeight = (fHeight < MAXHEIGHT) ?
            fHeight : MAXHEIGHT;
    }

}

```

The `DungeonItem` objects can appear in any order in the input file, but each starts with a character symbol followed by some integer data. The `PopulateDungeon()` function can use the character symbol to control a `switch()` statement in which objects of appropriate kinds are created and added to lists.

```

void Dungeon::PopulateDungeon(istream& in)
{
    char ch;
    Monster *m;
    in >> ch;
    while(ch != 'q') {
        switch(ch) {
case 'h':
            if(fPlayer != NULL) {
                cout << "Limit of one player "
                    "violated." << endl;
                exit(1);
            }
            else {
                fPlayer = new Player(this);
                fPlayer->Read(in);
            }
            break;
case 'w':
            m = new Wanderer(this);
            m->Read(in);
            fInhabitants.Append(m);
            break;
case 'g':
            m = new Ghost(this);
            m->Read(in);
            fInhabitants.Append(m);
            break;
case 'p':
            m = new Patrol(this);
            m->Read(in);
            fInhabitants.Append(m);
            break;
case '*':
case '=':
case '$':
            Collectable *prop = new Collectable(this, ch);
            prop->Read(in);
            fProps.Append(prop);

```

Create Player object

*Create different
specialized Monster
objects*

*Create Collectable
items*

```

        break;
default:
    cout << "Unrecognizable data in input file."
          << endl;
    cout << "Symbol " << ch << endl;
    exit(1);
    }
    in >> ch;
}
if(fPlayer == NULL) {
    cout << "No player! No Game!" << endl;
    exit(1);
}
if(fProps.Length() == 0) {
    cout << "No items to collect! No Game!" << endl;
    exit(1);
}
cout << "Dungeon population read" << endl;
}

```

The function verifies the requirements for exactly one `Player` object and at least one `Collectable` item.

The `Run()` function starts by creating the main map window and arranging for all objects to be drawn. The main `while()` loop shows the `Collectable` items, gets the `Player` move, then lets the `Monsters` have their turn.

```

int Dungeon::Run()
{
    CreateWindow();
    int n = fInhabitants.Length();
    for(int i=1; i <= n; i++) {
        Monster *m = (Monster*) fInhabitants.Nth(i);
        m->Draw();
    }
    fPlayer->Draw();
    fPlayer->ShowStatus();
    WindowRep::Instance()->Delay(1);

    while(fPlayer->Alive()) {
        for(int j=1; j <= fProps.Length(); j++) {
            Collectable *pi =
                (Collectable*) fProps.Nth(j);
            pi->Draw();
        }

        fPlayer->Run();
        if(fProps.Length() == 0)
            break;

        int n = fInhabitants.Length();
        for(i=1; i<= n; i++) {
            Monster *m = (Monster*)
                fInhabitants.Nth(i);
            m->Run();
        }
    }
}

```

```

    }
    return fPlayer->Alive();
}

```

(Note the need for type casts when getting members of the collections; the function `DynamicArray::Nth()` returns a `void*` pointer.)

The `CreateWindow()` function creates a `Window` object and sets its background from the map.

```

void Dungeon::CreateWindow()
{
    fDWindow = new Window(1, 1, fWidth, fHeight);
    for(int row = 1; row <= fHeight; row++)
        for(int col = 1; col <= fWidth; col++)
            fDWindow->SetBkgd(col, row,
                             fDRep[row-1][col-1]);
    fDWindow->PrepareContent();
    fDWindow->ShowAll();
}

```

Class `Dungeon` has several trivial access functions:

```

int Dungeon::Accessible(Pt p) const
{
    return (' ' == fDRep[p.Y()-1][p.X()-1]);
}

Window *Dungeon::Display() { return fDWindow; }
Player *Dungeon::Human() { return fPlayer; }

int Dungeon::ValidPoint(Pt p) const
{
    int x = p.X();
    int y = p.Y();
    // check x range
    if((x <= 1) || (x >= fWidth)) return 0;
    // check y range
    if((y <= 1) || (y >= fHeight)) return 0;
    // and accessibility
    return Accessible(p);
}

```

There are similar pairs of functions `M_at_Pt()` and `PI_at_Pt()`, and `RemoveM()` and `RemoveProp()` that work with the `fInhabitants` list of `Monsters` and the `fProps` list of `Collectables`. Examples of the implementations are

```

Collectable *Dungeon::PI_at_Pt(Pt p)
{
    int n = fProps.Length();
    for(int i=1; i<= n; i++) {
        Collectable *pi = (Collectable*) fProps.Nth(i);
        Pt w = pi->Where();
        if(w.Equals(p)) return pi;
    }
}

```

```

        }
        return NULL;
    }

    void Dungeon::RemoveM(Monster *m)
    {
        fInhabitants.Remove(m);
        m->Erase();
        delete m;
    }

```

The `ClearLineOfSight()` function checks the coordinates of the `Pt` arguments to determine which of the various specialized auxiliary functions should be called:

```

int Dungeon::ClearLineOfSight(Pt p1, Pt p2, int max, Pt path[])
{
    if(p1.Equals(p2)) return 0;
    if(!ValidPoint(p1)) return 0;
    if(!ValidPoint(p2)) return 0;

    if(p1.Y() == p2.Y())
        return ClearRow(p1, p2, max, path);
    else
        if(p1.X() == p2.X())
            return ClearColumn(p1, p2, max, path);

    int dx = p1.X() - p2.X();
    int dy = p1.Y() - p2.Y();

    if(abs(dx) >= abs(dy))
        return ClearSemiHorizontal(p1, p2, max, path);
    else
        return ClearSemiVertical(p1, p2, max, path);
}

```

The explanation of the algorithm given in the previous section dealt with cases involving rows or oblique lines that were more or less horizontal. The implementations given here illustrate the cases where the line is vertical or close to vertical.

```

int Dungeon::ClearColumn(Pt p1, Pt p2, int max, Pt path[])
{
    int delta = (p1.Y() < p2.Y()) ? 1 : -1;
    int x = p1.X();
    int y = p1.Y();
    for(int i = 0; i < max; i++) {
        y += delta;
        Pt p(x,y);
        if(!Accessible(p)) return 0;
        path[i] = p;
        if(p.Equals(p2)) return 1;
    }
    return 0;
}

```

```

int Dungeon::ClearSemiVertical(Pt p1, Pt p2, int max,
                               Pt path[])
{
    int ychange = p2.Y() - p1.Y();
    if(abs(ychange) > max) return 0;
    int xchange = p2.X() - p1.X();

    int deltax = (xchange > 0) ? 1 : -1;
    int deltay = (ychange > 0) ? 1 : -1;

    float slope = ((float)xchange)/((float)ychange);
    float error = slope*deltay;

    int x = p1.X();
    int y = p1.Y();
    for(int i=0;i<abs(ychange);i++) {
        if(error*deltax>0.5) {
            x += deltax;
            error -= deltax;
        }
        error += slope*deltay;
        y += deltay;
        Pt p(x, y);
        if(!Accessible(p)) return 0;
        path[i] = p;
        if(p.Equals(p2)) return 1;
    }
    return 0;
}

```

29.3.3 DungeonItems

DungeonItem

Class `DungeonItem` implements a few basic behaviours shared by all variants. Its constructor sets the symbol used to represent the item and sets the link to the `Dungeon` object. The body of the destructor is empty as there are no separate resources defined in the `DungeonItem` class.

```

DungeonItem::DungeonItem(Dungeon *d, char sym)
{
    fSym = sym;
    fD = d;
}

DungeonItem::~DungeonItem() { }

```

The `Erase()` and `Draw()` functions operate on the `Dungeon` object's main map Window. The call `fd->Display()` returns a `Window*` pointer. The Window referenced by this pointer is asked to perform the required operation.

```

void DungeonItem::Erase()
{

```

```

        fD->Display()->Clear(fPos.X(), fPos.Y());
    }

    void DungeonItem::Draw()
    {
        fD->Display()->Set( fPos.X(), fPos.Y(), fSym);
    }

```

All `DungeonItem` objects must read their coordinates, and the data given as input must be checked. These operations are defined in `DungeonItem::Read()`.

```

void DungeonItem::Read(istream& in)
{
    int x, y;
    in >> x >> y;
    if(!in.good()) {
        cout << "Problems reading coordinate data" << endl;
        exit(1);
    }
    if(!fD->ValidPoint(Pt(x,y))) {
        cout << "Invalid coords, out of range or"
              << "already occupied" << endl;
        cout << "(" << x << ", " << y << ")" << endl;
        exit(1);
    }
    fPos.SetPt(x,y);
}

```

Collectable

The constructor for class `Collectable` passes the `Dungeon*` pointer and `char` arguments to the `DungeonItem` constructor:

```

Collectable::Collectable(Dungeon* d, char sym) :
    DungeonItem(d, sym)
{
    fHval = fWval = fMval = 0;
}

```

Class `Collectable`'s access functions (`Wlth()` etc) simply return the values of the required data members. Its `Read()` function extends `DungeonItem::Read()`. Note the call to `DungeonItem::Read()` at the start; this gets the coordinate data. Then the extra integer parameters can be input.

```

void Collectable::Read(istream& in)
{
    DungeonItem::Read(in);
    in >> fHval >> fWval >> fMval;
    if(!in.good()) {
        cout << "Problem reading a property" << endl;
        exit(1);
    }
}

```

*Invoke inherited
Read function*

ActiveItem

The constructor for class `ActiveItem` again just initializes some data members to zero after passing the given arguments to the `DungeonItem` constructor. Function `ActiveItem::Read()` is similar to `Collectable::Read()` in that it invokes the `DungeonItem::Read()` function then reads the extra data values (`fHealth` and `fStrength`).

There are a couple of trivial functions (`GetHit()` { `fHealth -= damage;` }; and `Alive()` { `return fHealth > 0;` }). The `Move()` operation involves calls to the (inherited) `Erase()` and `Draw()` functions. Function `Step()` works out the `x, y` offset (+1, 0, or -1) coordinates of a chosen neighboring `Pt`.

```
void ActiveItem::Move(const Pt& newpoint)
{
    Erase();
    fPos.SetPt(newpoint);
    Draw();
}

Pt ActiveItem::Step(int dir)
{
    Pt p;
    switch(dir) {
    case 1: p.SetPt(-1,1); break;
    case 2: p.SetPt(0,1); break;
    case 3: p.SetPt(1,1); break;
    case 4: p.SetPt(-1,0); break;
    case 6: p.SetPt(1,0); break;
    case 7: p.SetPt(-1,-1); break;
    case 8: p.SetPt(0,-1); break;
    case 9: p.SetPt(1,-1); break;
    }
    return p;
}
```

Player

The constructor for class `Player` passes its arguments to its parents constructor and then sets its data members to 0 (NULL for the pointer members). The `Read()` function is similar to `Collectable::Read()`; it invokes the inherited `DungeonItem::Read()` and then gets the extra "manna" parameter.

The first call to `ShowStatus()` creates the `NumberItem` and `EditText` windows and arranges for their display. Subsequent calls update the contents of the `NumberItem` windows if there have been changes (the call to `SetVal()` results in execution of the `NumberItem` object's `ShowContents()` function so resulting in changes to the display).

```
void Player::ShowStatus()
{
    if(fWinH == NULL) {
```



```

        fWinH = new NumberItem(2, 20, 20, "Health", fHealth);
        fWinM = new NumberItem(30,20, 20, "Manna ", fManna);
        fWinW = new NumberItem(58,20, 20, "Wealth", fWealth);
        fWinE = new EditText(2, 22, 20, "Direction", 1);
        fWinH->ShowAll();
        fWinM->ShowAll();
        fWinW->ShowAll();
        fWinE->ShowAll();
    }
    else {
        if(fHealth != fWinH->GetVal()) fWinH->SetVal(fHealth);
        if(fManna != fWinM->GetVal()) fWinM->SetVal(fManna);
        if(fWealth != fWinW->GetVal()) fWinW->SetVal(fWealth);
    }
}

```

The Run() function involves getting and performing a command followed by update of state and display.

```

void Player::Run()
{
    char ch = GetUserCommand();
    if(isdigit(ch)) PerformMovementCommand(ch);
    else PerformMagicCommand(ch);
    UpdateState();
    ShowStatus();
}

void Player::UpdateState()
{
    fMoveCount++;
    if(0 == (fMoveCount % 3)) fHealth++;
    if(0 == (fMoveCount % 7)) fManna++;
}

```

The function PerformMovementCommand() first identifies the neighboring point. There is then an interaction with the Dungeon object to determine whether there is a Collectable at that point (if so, it gets taken). A similar interaction determines whether there is a Monster (if so, it gets attacked, after which a return is made from this function). If the neighboring point is not occupied by a Monster, the Player object moves to that location.

```

void Player::PerformMovementCommand(char ch)
{
    int x = fPos.X();
    int y = fPos.Y();
    Pt p = Step(ch - '0');
    int newx = x + p.X();
    int newy = y + p.Y();

    Collectable *pi = fD->PI_at_Pt(Pt(newx, newy));
    if(pi != NULL)
        Take(pi);
    Monster *m = fD->M_at_Pt(Pt(newx, newy));
}

```

```

        if(m != NULL) {
            Attack(m);
            return;
        }
        TryMove(x + p.X(), y + p.Y());
    }

```

The auxiliary functions, `Take()`, `Attack()`, and `TryMove()` are all simple. Function `Take()` updates the `Player` objects health and related attributes with data values from the `Collectable` item, and then arranges for the `Dungeon` to dispose of that item. Function `Attack()` reduces the `Monster` object's health (via a call to its `GetHit()` function) and, if appropriate, arranges for the `Dungeon` object to dispose of the `Monster`. Function `TryMove()` validates and then performs the appropriate movement.

The function `GetUserCommand()` arranges for the `EditText` window to input some text and then inspects the first character of the text entered.

```

char Player::GetUserCommand()
{
    fWinE->GetInput();
    char *str = fWinE->GetVal();
    return *str;
}

```

The function `PerformMagicCommand()` identifies the axis for the magic bolt. There is then a loop in which damage is inflicted (at a reducing rate) on any `Monster` objects found along a sequence of points in the given direction:

```

void Player::PerformMagicCommand(char ch)
{
    int dx, dy;
    switch (ch) {
    case 'q': dx = -1; dy = -1; break;
    case 'w': dx = 0; dy = -1; break;
    case 'e': dx = 1; dy = -1; break;
    case 'a': dx = -1; dy = 0; break;
    case 'd': dx = 1; dy = 0; break;
    case 'z': dx = -1; dy = 1; break;
    case 'x': dx = 0; dy = 1; break;
    case 'c': dx = 1; dy = 1; break;
    default:
        return;
    }
    int x = fPos.X();
    int y = fPos.Y();

    int power = 8;
    fManna -= power;
    if(fManna < 0) {
        fHealth += 2*fManna;
        fManna = 0;
    }
    while(power > 0) {
        x += dx;

```

```

        y += dy;
        if(!fD->ValidPoint(Pt(x,y))) return;
        Monster* m = fD->M_at_Pt(Pt(x,y));
        if(m != NULL) {
            m->GetHit(power);
            if(!m->Alive())
                fD->RemoveM(m);
        }
        power /= 2;
    }
}

```

Monster

The constructor and destructor functions of class `Monster` both have empty bodies for there is no work to be done; the constructor passes its arguments back to the constructor of its parent class (`ActiveItem`):

```

Monster::Monster(Dungeon *d, char sym) : ActiveItem(d, sym)
{
}

```

Function `Monster::Run()` was defined earlier. The default implementations of the auxiliary functions are:

```

int Monster::CanAttack()
{
    Player *p = fD->Human();
    Pt target = p->Where();
    return fPos.Adjacent(target);
}

void Monster::Attack()
{
    Player *p = fD->Human();
    p->GetHit(fStrength);
}

int Monster::CanDetect() { return 0; }

void Monster::Advance() { }

```

Ghost

The `Ghost::CanDetect()` function uses the `Pt::Distance()` member function to determine the distance to the `Player` (this function just takes the normal Euclidean distance between two points, rounded up to the next integral value).

```

int Ghost::CanDetect()
{
    Player *p = fD->Human();

```

```

        int range = fPos.Distance(p->Where());
        return (range < 7);
    }

```

The `Advance()` function determines the change in x, y coords that will bring the Ghost closer to the Player.

```

void Ghost::Advance()
{
    Player *p = fD->Human();
    Pt p1 = p->Where();
    int dx, dy;
    dx = dy = 0;
    if(p1.X() > fPos.X()) dx = 1;
    else
    if(p1.X() < fPos.X()) dx = -1;
    if(p1.Y() > fPos.Y()) dy = 1;
    else
    if(p1.Y() < fPos.Y()) dy = -1;

    Move(Pt(fPos.X() + dx, fPos.Y() + dy));
}

```

Wanderer

The `Wanderer::CanDetect()` function uses the `Dungeon::ClearLineOfSight()` member function to determine whether the `Player` object is visible. This function call also fills in the array `fPath` with the points that will have to be crossed.

```

int Wanderer::CanDetect()
{
    Player *p = fD->Human();
    return
        fD->ClearLineOfSight(fPos, p->Where(), 10, fPath);
}

```

The `Advance()` function moves one step along the path:

```

void Wanderer::Advance()
{
    Move(fPath[0]);
}

```

The `NormalMove()` function tries moving in the same direction as before. Directions are held by storing the delta-x and delta-y values in `fLastX` and `fLastY` data members (initialized to zero in the constructor). If movement in that general direction is blocked, a new direction is picked randomly.

```

void Wanderer::NormalMove()
{
    int x = fPos.X();
    int y = fPos.Y();

```

```

// Try to keep going in much the same direction as last time
if((fLastX != 0) || (fLastY != 0)) {
    int newx = x + fLastX;
    int newy = y + fLastY;
    if(fD->Accessible(Pt(newx,newy))) {
        Move(Pt(newx,newy));
        return;
    }
    else
    if(fD->Accessible(Pt(newx,y))) {
        Move(Pt(newx,y)); fLastY = 0;
        return;
    }
    else
    if(fD->Accessible(Pt(x,newy))) {
        Move(Pt(x,newy)); fLastX= 0;
        return; }
    }
    int dir = rand();
    dir = dir % 9;
    dir++;
    Pt p = Step(dir);
    x += p.X();
    y += p.Y();
    if(fD->Accessible(Pt(x,y))) {
        fLastX = p.X();
        fLastY = p.Y();
        Move(Pt(x,y));
    }
}

```

Movement in same direction

Movement in similar direction

Pick new direction at random

Patrol

The patrol route has to be read, consequently the inherited `Read()` function must be extended. There are several possible errors in route definitions, so `Patrol::Read()` involves many checks:

```

void Patrol::Read(istream& in)
{
    Monster::Read(in);
    fRoute[0] = fPos;
    fNdx = 0;
    fDelta = 1;
    in >> fRouteLen;
    for(int i=1; i<= fRouteLen; i++) {
        int x, y;
        in >> x >> y;
        Pt p(x, y);
        if(!fD->ValidPoint(p)) {
            cout << "Bad data in patrol route" << endl;
            cout << "(" << x << ", " << y << ")" <<
endl;
            exit(1);
        }
    }
}

```

```

        if(!p.Adjacent(fRoute[i-1])) {
            cout << "Non adjacent points in patrol"
                "route" << endl;
            cout << "(" << x << ", " << y << ")" << endl;
            exit(1);
        }
        fRoute[i] = p;
    }
    if(!in.good()) {
        cout << "Problems reading patrol route" << endl;
        exit(1);
    }
}

```

The `NormalMove()` function causes a `Patrol` object to move up or down its route:

```

void Patrol::NormalMove()
{
    if((fNdx == 0) && (fDelta == -1)) {
        fDelta = 1;
        return;
    }
    if((fNdx == fRouteLen) && (fDelta == 1)) {
        fDelta = -1;
        return;
    }
    fNdx += fDelta;
    Move(fRoute[fNdx]);
}

```

Reverse direction at start

Reverse direction at end

Move one step along route

The `CanDetect()` function is identical to `Wanderer::CanDect()`. However, instead of advancing one step along the path to the `Player`, a `Patrol` fires a projectile that moves along the complete path. When the projectile hits, it causes a small amount of damage:

```

void Patrol::Advance()
{
    Player *p = fD->Human();
    Pt target = p->Where();
    Pt arrow = fPath[0];
    int i = 1;
    while(!arrow.Equals(target)) {
        fD->Display()->Set( arrow.X(), arrow.Y(), ':');
        WindowRep::Instance()->Delay(1);
        fD->Display()->Clear( arrow.X(), arrow.Y());
        arrow = fPath[i];
        i++;
    }
    p->GetHit(2);
}

```

EXERCISES

- 1 Complete and run the dungeon game program.
- 2 This one is only for users of Borland's system.

Why should the monsters wait while the user thinks? If they know what they want to do, they should be able to continue!

The current program requires user input in each cycle of the game. If there is no input, the program stops and waits. The game is much more interesting if this wait is limited. If the user doesn't type any command within a second or so, the monsters should get their chance to run anyway.

This is not too hard to arrange.

First, the main while() loop in `Dungeon::Run()` should have a call `WindowRep::Instance()->Delay(1)`. This results in a 1 second pause in each cycle.

The `Player::Run()` function only gets called if there have been some keystrokes. If there are no keystrokes waiting to be processed, the `Dungeon::Run()` function skips to the loop that lets each monster have a chance to run.

All that is required is a system function, in the "console" library package, that allows a program to check whether input data are available (without "blocking" like a normal read function). The Borland `conio` library includes such a function.

Using the on-line help system in the Borland environment, and other printed documentation, find how to check for input. Use this function in a reorganized version of the dungeon program.

(You can achieve the same result in the Symantec system but only by utilising specialized system calls to the "Toolbox" component of the Macintosh operating system. It is all a little obscure and clumsy.)

- 3 Add multiple levels to the dungeon.

(There are various ways that this might be done. The easiest is probably to define a new class `DungeonLevel`. The `Dungeon` object owns the main window, the `Player`, and a list of `DungeonLevel` objects. Each `DungeonLevel` object owns a map, a list of collectables, and a list of monsters. You will need some way of allowing a user to go up or down levels. When you change level, the new `DungeonLevel` resets the background map in the main window and arranges for all data to be redrawn.)

- 4 Add more challenging Monsters and "traps".

(Use your own imagination.)

30 Reusable designs

The last chapter illustrated some simple uses of inheritance and polymorphism. It is these programming techniques that distinguish "Object Oriented Programming" from the "object based" (or abstract data type) style that was the main focus of Part IV of this text. These Object Oriented (OO) techniques originated in work on computer simulation of real world systems. The "dungeon game" is a simulation (of an unreal world) and so OO programming techniques are well suited to its implementation.

Although OO programming techniques were originally viewed as primarily applicable to simulations, they have over the last ten years become much more widely utilised. This greater use is largely a consequence of the increased opportunity for "reuse" that OO techniques bring to application development.

Reuse, whether of functions, components, or partial designs, always enhances productivity. If you can exploit reusable parts to handle "standard" aspects of an application, you can focus your efforts on the unique aspects of the new program. You will produce a better program, and you will get it working sooner than if you have to implement everything from scratch.

Approaches to Reuse

Reusable algorithms

If you are using the "top down functional decomposition" strategy that was illustrated in Part III, you are limited to reusing standard algorithms; the code in the function libraries implements these standard algorithms. Reusing algorithms is better than starting from scratch. These days, nobody writes their own `sin()` function, they use the version in the maths library. Computer science students are often made to rewrite the standard sorting and searching functions, but professionals use `qsort()` and `bsearch()` (standardized sort and binary search functions that are available in almost every development environment). As noted in Chapter 13, over the years huge libraries of functions have been built up, particularly in science and engineering, to perform standard calculations.

Reuse with functions

Function libraries for interactive programs

Reusable functions are helpful in the case where all you need to do is calculate something. But if you want to build an interactive program with windows and menus etc, you soon discover problems.

There are function libraries that are intended to be used when building such programs. Multi-volume reference manuals exist to describe them. For example, the Xlib reference manuals define the functions you can use to work with the X-windows interface on Unix. The series "Inside Macintosh" describes how to create windows and menus for the Mac. OS. These books include the declarations of literally hundreds of functions, and dozens of data structures. But these function libraries are very difficult to use.

Limitations of function libraries

The functions defined in these large libraries are disjoint, scattered, inconsistently named. There is no coherence. It is almost impossible to get a clear picture of how to organize a program. Instead you are faced with an arbitrary collection of functions, and the declarations of some types of structures that you have to have as globals. Programs built using just these function libraries acquire considerable entropy (chaotic structure). Each function call takes you off to some other arbitrary piece of code that rapes and pillages the global data structures.

Reusable components***Reusable classes and object based design***

The "object based" techniques presented in Part IV give you a better handle on reuse. Class libraries and object based programs allow you to reuse abstract data types. The functions and the data that they operate on are now grouped. The data members of instances of classes are protected; the compiler helps make sure that data are only accessed via the appropriate functions.

Program design is different. You start by identifying the individual objects that are responsible for particular parts of the overall data. You define their classes. Often, you will find that you can reuse standard classes, like the collection classes in Chapters 21 and 24. As well as providing working code, these classes give you a way of structuring the overall program. The program becomes a sequence of interactions between objects that are instances of standard and application specific classes.

Essentially, the unit of reuse has become larger. Programs are built at least in part from reusable components. These reusable components include collection classes and, on Unix, various forms of "widget". (A widget is essentially a class that defines a "user interface" component like a menu or an alert box.)

Reusable patterns of object interactions and program designs***Can we reuse more?***

When inheritance was introduced in Chapter 23, it was shown that this was a way of representing and exploiting similarities. Many application programs have substantial similarities in their behaviour; such similarities lead to reusable designs.

Similar patterns of interactions in different programs

You launch a program. Once it starts, it presents you with some form of "file dialog" that allows you to create a new file, or open an existing file. The file is opened. One or more windows are created. If the file existed previously, some

portions of its current contents get displayed in these new windows. The system's menu bar gets changed, or additional menus or tool bars appear associated with the new window(s). You use the mouse pointer and buttons to select a menu option and a new "tools palette" window appears alongside the document window. You select a tool from the palette. You use the tool to add data to the document.

The behaviour is exactly the same. It doesn't matter whether it is a drawing program or spreadsheet. The same patterns of behaviour are repeated.

Object oriented programming techniques provide a way of capturing common patterns of behaviour. These patterns involve standardized interactions between instances of different classes.

Reusable patterns of interaction?

Capturing standard patterns of interaction in code

The "opening sequence for a program" as just described would involve interactions between an "application" object, a "document" object, several different "window" objects, maybe a "menu manager" object and several others.

An "opening sequence" pattern could specify that the "application" object handle the initial File/New or File/Open request. It should handle such a request by creating a document object and giving it the filename as an argument to an "OpenNew()" or "OpenOld()" member function. In "OpenOld()", the document object would have to create some objects to store the data from the file and arrange to read the existing data. Once the "open" step is complete, the application object would tell the new document object to create its display structure. This step would result in the creation of various windows.

Much is standard. The standard interactions among the objects can be defined in code:

```
Application::HandleCommand( command#, ...)
    switch(command#)
newCommand:
    doc = this->DoMakeDocument();
    doc->OpenNew();
    doc->CreateDisplay();
    break;
openCommand:
    filename = this->PoseFileDialog();
    doc = this->DoMakeDocument();
    doc->OpenOld(filename, ...);
    doc->CreateDisplay();
    break;
...
```

*Default
implementation
defined*

```
Document::OpenOld(filename, ...)
    this->DoMakeDataStructures()
    this->DoRead(filename, ...)
```

*Default
implementation
defined*

```
Document::DoMakeDataStructures() ?
```

*Pure abstract
functions,
implementation is
application specific*

```
Document::DoRead(...) ?
```

Of course, each different program does things differently. The spreadsheet and drawing programs have to create different kinds of data structure and then have to read differently formatted files of data.

Utilize class inheritance

This is where inheritance comes in.

The situation is very much like that in the dungeon game with class `Monster` and its subclasses. The `Dungeon` code was written in terms of interactions between the `Dungeon` object and instances of class `Monster`. But there were never any `Monster` objects. Class `Monster` was an abstraction that defined a few standard behaviours, some with default implementations and some with no implementation. When the program ran, there were instances of specialized subclasses of class `Monster`; subclasses that owned their own unique data and provided effective implementations of the behaviours declared in class `Monster`.

An abstract class Document

Now, class `Document` is an abstraction. It defines something that can be asked to open new or old files, create displays and so forth. All kinds of document exhibit such behaviours; each different kind does things slightly differently.

Possible specialized subclasses

Specialized subclasses of class `Document` can be defined. A `SpreadSheetDoc` would be a document that owns an array of `Cell` objects where each `Cell` is something that holds either a text label, or a number, or a formula. A `DrawDoc` would be a document that owns a list of `PictureElements`. Each of these specialized subclasses would provide effective definitions for the empty `Document::DoRead()` and `Document::DoMakeDataStructures()` functions (and for many other functions as well!).

Building complete programs

A particular program won't create different kinds of document! Instead, you build the "spreadsheet" program or the "draw" program.

For the "draw" program, you would start by creating class `DrawApp` a minor specialization of class `Application`. The only thing that `DrawApp` does differently is that its version of the `DoMakeDocument()` function creates a `DrawDoc`. A `DrawDoc` is pretty much like an ordinary `Document`, but it has an extra `List` data member (to store its `PictureElements`) and, as already noted, it provides effective implementations of functions like `DoRead()`.

Such a program gets built with much of its basic structure defined in terms of classes that are specializations of standardized, reusable classes taken from a library. These reusable classes are the things like `Application`, `Document`, and `Window`. Some of their member functions are defined with the necessary code in the implementation files. Other member functions may have empty (do nothing) implementations. Still other member functions are pure virtual functions that must be given definitions in subclasses.

Reusing a design

Reusing a design

A program built in this fashion illustrates reuse on a new scale. It isn't just individual components that are being reused. Reuse now extends to design.

Design ideas are embedded in the code of those functions that are defined in the library. Thus, the "standard opening sequence" pattern implements a particular design idea as to how programs should start up and allow their users to select the data files that are to be manipulated. Another defined pattern of interactions might

specify how an `Application` object was to handle a "Quit" command (it should first give any open document a chance to save changes, tell the document to close its windows and get rid of data, delete the document, close any application windows e.g. floating tool palettes, and finally quit).

The code given for the standard classes will embody a particular "look and feel" as might be required for all applications running on a particular type of machine. The specifications for a new application would normally require compliance with "standard look and feel". If you had to implement a program from scratch, you would have to sort out things like the "standard opening sequence" and "standard quit" behaviours and implement all the code. If you have a class library that embodies the design, you simply inherit it and get on with the new application specific coding.

Default code implements the "standard look and feel"

It is increasingly common for commercial products to be built using standardized framework class libraries. A "framework class library" has the classes that provide the basic structure, the framework, for all applications that comply with a standardized design. The Integrated Development Environment that you have on your personal computers includes such a class library. You will eventually get to use that library.

Framework class libraries

A simplified example framework

The real framework class libraries are relatively complex. The rest of this chapter illustrates a simplified framework that can serve as an introduction.

While real frameworks allow for many different kinds of data and document; this "RecordFile" framework is much more restricted. Real frameworks allow for multiple documents and windows; here you make do with just one of each. Real frameworks allow you to change the focus of activity arbitrarily so one moment you can be entering data, the next moment you can be printing some graphic representation of the data. Here, the flow of control is much more predefined. All these restrictions are needed to make the example feasible. (The restrictions on flow of control are the greatest simplifying factor.)

30.1 THE RECORDFILE FRAMEWORK: CONCEPTS

The "RecordFile" framework embodies a simple design for any program that involves updating "records" in a data file. The "records" could be things like the customer records in the example in Chapter 17. It is primarily the records that vary between different program built using this framework.

Figure 30.1 shows the form of the record used in a program, "StudentMarks", built using the framework. This program keeps track of students and their marks in a particular subject. Students' have unique identifier numbers, e.g. the student number 938654. The data maintained include the student's name and the marks for assignments and exams. The name is displayed in an editable text field; the marks are in editable number entry fields. When a mark is changed, the record updates the student's total mark (which is displayed in a non-editable field.)

Example program and record

Record identifier		938654	← Unique record identifier
Student Name		Norman, Harvey	← Text in editable field
Assignment 1 (5)	4	MidSession (15)	11
Assignment 2 (10)	9	Examination (50)	0
Assignment 3 (10)	4		
Assignment 4 (10)	0	Total	28

← Number in editable field

Figure 30.1 A "record" as handled by the "RecordFile Framework".

**Starting: "New",
"Open", "Quit"**

When the "StudentMarks" program is started, it first presents the user with a menu offering the choices of "New (file)", "Open (existing file)", or "Quit". If the user selects "New" or "Open", a "file-dialog" is used to prompt for the name of the file.

**Changing the
contents of a file**

Once a file has been selected, the display changes, see Figure 30.2. It now displays details of the name of the file currently being processed, details of the number of records in the file, and a menu offering various options for adding, deleting, or modifying records.

**Handling a "New
record" command**

If the user selects "New record", the program responds with a dialog that requires entry of a new unique record identifier. The program verifies that the number entered by the user does not correspond to the identifier of any existing record. If the identifier is unique, the program changes to a record display, like that shown in Figure 30.1, with all editable fields filled with suitable default values.

**Handling "Delete
..." and "View/edit
..." commands**

If the user picks "Delete record" or "View/edit record", the program's first response is to present a dialog asking for the identifier number of an existing record. The number entered is checked; if it does not correspond to an existing record, no further action is taken.

A "Delete record" command with a valid record identifier results in the deletion of that record from the collection. A "View/edit" command leads to a record display showing the current contents of the fields for the record.

**Handling a "Close"
command**

After performing any necessary updates, a "Close" command closes the existing file. The program then again displays its original menu with the options "New", "Open", and "Quit".

**Another program,
another record
display**

"Loans" is another program built on the same framework. It is very similar in behaviour, but this program keeps track of movies that a customer has on loan from a small video store. Its record is shown in Figure 30.3.

```

CS204  Filename                                     | Number of records:                               138
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
==> New record                                     | Record count
Delete record                                     |
View/edit record                                 |
Close file                                       |

Menu,
(current choice highlighted, changed
by tabbing between choices, "enter"
to select processing)

(use 'option-space' to switch between choices, 'enter' to select)

```

Figure 30.2 The menu of commands for record manipulation.

Record identifier		16241	
+-----+		+-----+	
Customer Name Jones, David		Phone	818672
+-----+		+-----+	
Movie title		Charge \$	
+-----+		+-----+	
Gone With The Wind		4	
+-----+		+-----+	
Casablanca		4	Total 12
+-----+		+-----+	
Citizen Kane		4	
+-----+		+-----+	
		0	Year 290
+-----+		+-----+	
		0	
+-----+		+-----+	

Figure 30.3 Record from another "RecordFile" program.

The overall behaviours of the two programs are identical. It is just the records that change. With the StudentMarks program, the user is entering marks for different pieces of work. In the Loans program, the user enters the names of movies and rental charges.

30.2 THE FRAMEWORK CLASSES: OVERVIEW

The classes used in programs like "StudentMarks" and "Loans" are illustrated in the class hierarchy diagram shown in Figure 30.4.

Class browser

Most IDEs can produce hierarchy diagrams, like that in Figure 30.4, from the code of a program. Such a diagram is generated by a "class browser". A specific class can be selected, using the mouse, and then menu commands (or other controls) can be used to open the file with the class declaration or that with the definition of a member function chosen from a displayed list. A class declaration or function definition can be edited once it has been displayed. As you get more deeply into the use of classes, you may find the "browser" provides a more convenient editing environment than the normal editor provided by the IDE.

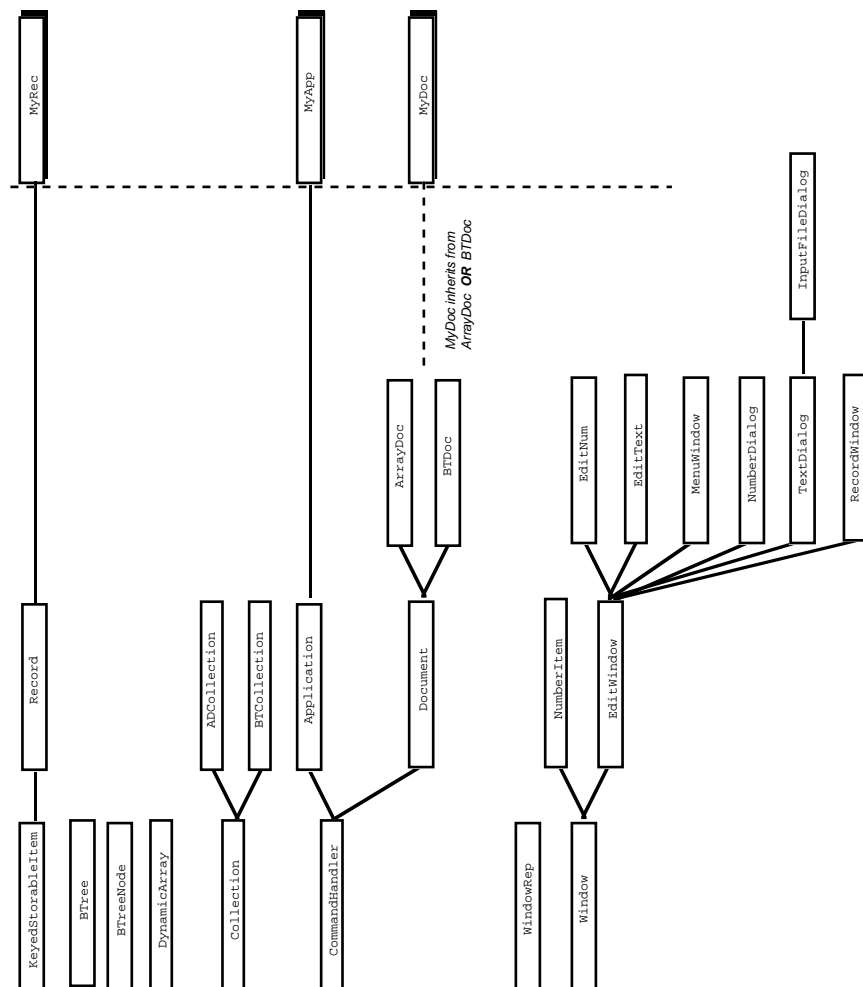


Figure 30.4 Class hierarchy for "RecordFile" Framework.

As illustrated in Figure 30.4, a program built using the framework may need to define as few as three classes. These are shown in Figure 30.4 as the classes `MyApp`, `MyDoc`, and `MyRec`; they are specializations of the framework classes `Application`, `Document`, and `Record`.

Classes `KeyedStorableItem`, `Record`, and `MyRec`

The class `KeyedStorableItem` is simply an interface (same as used in Chapter 24 for the `BTree`). A `KeyedStorableItem` is something that can report its key value (in this case, the "unique record identifier"), can say how much disk space it occupies, and can be asked to transfer its permanent data between memory and file. Its functions are "pure virtual"; they cannot be given any default definition, they must be defined in subclasses.

KeyedStorableItem

Class `Record` adds a number of additional behaviours. `Record` objects have to be displayed in windows. The exact form of the window depends on the specific kind of `Record`. So a `Record` had better be able to build its own display window, slotting in the various "EditText" and "EditNum" subwindows that it needs. Since the contents of the "edit" windows can get changed, a `Record` had better be capable of setting the current value of a data member in the corresponding edit window, and later reading back a changed value. Some of these additional functions will be pure virtual, but others may have "partial definitions". For example, every `Record` should display its record identifier. The code to add the record identifier to the display window can be coded as `Record::AddFieldsToWindow()`. A specialized implementation of `AddFieldsToWindow()`, as defined for a subclass, can invoke this standard behaviour before adding its own unique "edit" subwindows.

Record

Every specialized program built using the framework will define its own "MyRec" class. (This should have a more appropriate name such as `StudentRec` or `LoanRec`.) The "MyRec" class will define the data members. So for example, class `StudentRec` would specify a character array to hold the student's name and six integer data members to hold the marks (the total can be recomputed when needed). A `LoanRec` might have an array of fixed length strings for the names of the movies on loan.

MyRec

The specialized `MyRec` class usually wouldn't need to add any extra functionality but it would have to define effective implementations for all those pure virtual functions, like `DiskSize()` and `ReadFrom()`, declared in class `KeyedStorableItem`. It would also have to provide the rest of the implementation of functions like `Record::AddFieldsToWindow()`.

Collection classes and "adapters"

Programs work with sets of records: e.g. all the students enrolled in a course, or all the customers of the video store.

A program that has to work with a small number of records might chose to hold them all in main memory. It would use a simple collection class like a dynamic array, list, or (somewhat better) something like a binary tree or AVL tree. It would work by loading all its records from file into memory when a file was opened,

letting the user change these records and add new records, and finally write all records back to the file.

A program that needed a much larger collection of records would use something like a BTree to store them.

Collection classes

The actual "collection classes" are just those introduced in earlier chapters. The examples in this chapter use class `DynamicArray` and class `BTree`, but any of the other standard collection classes might be used. An instance of the chosen collection class will hold the different `Record` objects. Figure 30.4 includes class `DynamicArray`, class `BTree` and its auxiliary class `BTreeNode`.

The different collection classes have slightly different interfaces and behaviours. For example, class `BTree` looks after its own files. A simpler collection based on an in-memory list or dynamic array will need some additional component to look after disk transfers. But we don't want such differences pervading the main code of the framework.

Adapter classes for different collections

Consequently, the framework uses some "adapter" classes. Most of the framework code can work in terms of a "generic" `Collection` that responds to requests like "append record", "delete record". Specialized "adapter" classes can convert such requests into the exact forms required by the specific type of collection class that is used.

ADCollection and BTreeCollection

Figure 30.4 shows two adapter classes: `ADCollection` and `BTreeCollection`. An `ADCollection` object contains a dynamic array; a `BTreeCollection` owns a `BTree` object (i.e. it has a `BTree*` data member, the `BTree` object is created and deleted by code in `BTreeCollection`). These classes provide implementations of the pure virtual functions `Collection::Append()` etc. These implementations call the appropriate functions of the actual collection class object that is used to store the data records. The adapter classes also add any extra functions that may be needed in association with a specific type of collection.

Command Handlers: Application, Document and their subclasses

Although classes `Application` and `Document` have quite different specific roles there are some similarities in their behaviour. In fact, there are sufficient similarities to make it worth introducing a base class, class `CommandHandler`, that embodies these common behaviours.

A `CommandHandler` is something that has the following two primary behaviours. Firstly it "runs". "Running" means that it builds a menu, loops handling commands entered via the menu, and finally tidies up.

```
CommandHandler      void CommandHandler::Run( )
::Run()           {
                    this->MakeMenu( );
                    ...
                    this->CommandLoop( );
                    ...
                    this->Finish( );
                    }
```

The second common behaviour is embodied in the `CommandLoop()` function. This will involve menu display, and then processing of a selected menu command:

*CommandHandler
::CommandLoop()*

```
void CommandHandler::CommandLoop()
{
    while(!fFinished) {
        ...
        int c = pose menu dialog ...
        this->HandleCommand(c);
    }
}
```

A `CommandHandler` object will continue in its command handling loop until a flag, `fFinished`, gets set. The `fFinished` flag of an `Application` object will get set by a "Quit" command. A `Document` object finishes in response to a "Close" command.

As explained in the previous section, the `Application` object will have a menu with the choices "New", "Open" and "Quit". Its `HandleCommand()` function is:

Application

```
void Application::HandleCommand(int cmdnum)
{
    switch(cmdnum) {
    case cNEW:
        fDoc = this->DoMakeDocument();
        fDoc->DoInitialState();
        fDoc->OpenNew();
        fDoc->Run();
        delete fDoc;
        break;
    case cOPEN:
        fDoc = this->DoMakeDocument();
        fDoc->DoInitialState();
        fDoc->OpenOld();
        fDoc->Run();
        delete fDoc;
        break;
    case cQUIT:
        fFinished = 1;
        break;
    }
}
```

*Application::
HandleCommand()*

The "New" and "Open" commands result in the creation of some kind of `Document` object (obviously, this will be an instance of a specific concrete subclass of class `Document`). Once this `Document` object has been created, it will be told to open a new or an existing file, and then it will be told to "run".

The `Document` object will continue to "run" until it gets a "Close" command. It will then tidy up. Finally, the `Document::Run()` function, invoked via `fDoc->Run()`, will return. The `Application` object can then delete the `Document` object, and resume its "run" behaviour by again displaying its menu.

How do different applications vary?

class MyApp

The application objects in the "StudentMarks" program and "Loans" program differ only in the kind of `Document` that they create. A "MyApp" specialized

subclass of class `Application` need only provide an implementation for the `DoMakeDocument()` function. Function `Application::DoMakeDocument()` will be "pure virtual", subclasses must provide an implementation. A typical implementation will be along the following lines:

```
Document *MyApp::DoMakeDocument()
{
    return new MyDoc;
}
```

A `StudentMarkApp` would create a `StudentMarkDoc` while a `LoanApp` would create a `LoanDoc`.

Specialized subclasses of class `Application` could change other behaviours because all the member functions of class `Application` will be virtual. But in most cases only the `DoMakeDocument()` function would need to be defined.

class Document

Class `Document` is substantially more complex than class `Application`. It shares the same "run" behaviour, as defined by `CommandHandler::Run()`, and has a rather similar `HandleCommand()` function:

Document::Handle Command()

```
void Document::HandleCommand(int cmdnum)
{
    switch(cmdnum) {
    case cNEWREC:
        DoNewRecord();
        break;
    case cDELREC:
        DoDeleteRecord();
        break;
    case cVIEW:
        DoViewEditRecord();
        break;
    case cCLOSE:
        DoCloseDoc();
        fFinished = 1;
        break;
    }
}
```

Functions like `DoNewRecord()` are implemented in terms of a pure virtual `DoMakeRecord()` function. It is this `Document::DoMakeRecord()` function that gets defined in specialized subclasses so as to create the appropriate kind of `Record` object (e.g. a `StudentRec` or a `LoanRec`).

`Document` objects are responsible for several other activities. They must create the collection class object that they work with. They must put up dialogs to get file names and they may need to perform other actions such as getting and checking record numbers.

Document hierarchy

While the "adapter" classes can hide most of the differences between different kind of collection, some things cannot be hidden. As noted in the discussion above on `ADCollection` and `BTCollection`, there are substantial differences between those collections that are loaded entirely into memory from file as a program starts

and those, like the BTree based collection, where individual records are fetched as needed.

There has to be a kind of parallel hierarchy between specialized collection classes and specialized Document classes. This is shown in Figure 30.4 with the classes ArrayDoc and BTDoc. An ArrayDoc object creates an instance of an ADCollection as its Collection object while a BTDoc creates a BTCollection. Apart from DoMakeCollection() (the function that makes the Collection object), these different specialized subclasses of Document differ in their implementations of the functions that deal with opening and closing of files.

ArrayDoc and BTDoc

Different programs built using the framework must provide their own specialized Document classes – class StudentMarkDoc for the StudentMarks program or LoanDoc for the Loans program. Figure 30.4 uses class MyDoc to represent the specialized Document subclass needed in a specific program.

MyDoc

Class MyDoc won't be an immediate subclass of class Document, instead it will be based on a specific storage implementation like ArrayDoc or BTDoc.

Window hierarchy

As is commonly the case with frameworks, most of the classes are involved with user interaction, both data display and data input. In Figure 30.4, these classes are represented by the "Window" hierarchy. (Figure 30.4 also shows class WindowRep. This serves much the same role as the WindowRep class used Chapter 29; it encapsulates the low level details of how to communicate with a cursor addressable screen.)

The basic Window class is an extended version of that used in Chapter 29. It possesses the same behaviours of knowing its size and location on the screen, maintaining "foreground" and "background" images, setting characters in these images etc. In addition, these Window objects can own "subwindows" and can arrange that these subwindows get displayed. They can also deal with display of text strings and numbers at specific locations.

class Window

Class NumberItem is a minor reworking of the version from Chapter 29. An instance of class NumberItem can be used to display the current value of a variable and can be updated as the variable is changed.

NumberItem

The simple EditText class of Chapter 29 has been replaced by an entire hierarchy. The new base class is EditWindow. EditWindow objects are things that can be told to "handle input". "Handling input" involves accepting and processing input characters until a '\n' character is entered.

EditWindow

Class EditNum and EditText are simple specializations that can be used for verified numeric or text string input. An EditNum object accepts numeric characters, using them to determine the (integer) value input. An EditNum object can be told the range permitted for input data; normally it will verify that the input value is in this range (substituting the original value if an out of range value is input). An EditText object accepts printable characters and builds up a string (with a fixed maximum length).

*EditNum and
EditText*

A MenuWindow allows a user to pick from a displayed list of menu items. A MenuWindow is built up by adding "menu items" (these have a string and a numeric

MenuWindow

identifier). When a `MenuWindow` is displayed, it shows its menu items along with an indicator of "the currently selected item" (starting at the first item in the menu). A `MenuWindow` handles "tab" characters (other characters are ignored). A "tab" changes the currently selected item. The selection moves cyclically through the list of items. When "enter" ('\n') is input, the `MenuWindow` returns the numeric identifier associated with the currently selected menu item.

Classes
NumberDialog and
TextDialog

The "dialogs" display small windows centered in the screen that contain a prompt and an editable field (an instance of class `EditNum` or class `EditText`). The user must enter an acceptable value before the dialog will disappear and the program continue. Class `InputFileDialog` is a minor specialization of `TextDialog` that can check whether a string given as input corresponds to the name of an existing file.

RecordWindow

Class `RecordWindow` is a slightly more elaborate version of class `MenuWindow`. A `RecordWindow` owns a list of `EditNum` and `EditText` subwindows. "Tabbing" in a `RecordWindow` selects successive subwindows for further input.

The "MyRec" class used in a particular program implements a function, `AddFieldsToWindow()`, that populates a `RecordWindow` with the necessary `EditNum` and `EditText` subwindows.

30.3 THE COMMAND HANDLER CLASSES

30.3.1 Class declarations

CommandHandler

The declaration of class `CommandHandler` is:

```
class CommandHandler {
public:
    CommandHandler(int mainmenuid);
    virtual ~CommandHandler();

    virtual void Run();
protected:
    virtual void MakeMenu() = 0;
    virtual void CommandLoop();
    virtual void PrepareToRun() { }
    virtual void HandleCommand(int command) = 0;
    virtual void UpdateState() { }
    virtual void Finish() { }

    MenuWindow *fMenu;
    int fFinished;
    int fMenuID;
};
```

A `CommandHandler` is basically something that owns a `MenuWindow` (with an associated integer identifier). A `CommandHandler` can "run". It does this by filling in the menu entries, "preparing to run", executing its command loop, and finally

tidying up. The command loop will involve an update of status, acceptance of a command number from the `MenuWindow` and execution of `HandleCommand()`. One of the commands must set the `fFinished` flag.

Some of the functions are declared with "empty" implementations, e.g. `PrepareToRun()`. Such functions are fairly common in frameworks. They represent points where the framework designer has made provision for "unusual" behaviours that might be necessary in specific programs.

*Functions with
empty bodies*

Usually there is nothing that must be done before an `Application` displays its main menu, or after its command handling loop is complete. But it is possible that a particular program would need special action (e.g. display of a "splash screen" that identifies the program). So, functions `PrepareToRun()` and `Finish()` are declared and are called from within defined code, like that of function `Run()`. These functions are deliberately given empty definitions (i.e. `{ }`); there is no need to force every program to define actual implementations.

In contrast functions like `MakeMenu()` and `HandleCommand()` are pure virtual. These have to be given definitions before you have a working program.

*Pure virtual
functions*

Application

Class `Application` is a minor specialization of `CommandHandler`. An `Application` is a kind of `CommandHandler` that owns a `Document` that it creates in its `DoMakeDocument()` function. An `Application` provides effective implementations for the pure virtual functions `CommandHandler::MakeMenu()` and `CommandHandler::HandleCommand()`.

```
class Application : public CommandHandler {
public:
    Application();
    virtual ~Application();
protected:
    virtual void    MakeMenu();
    virtual void    HandleCommand(int command);
    virtual        Document* DoMakeDocument() = 0;
    Document        *fDoc;
};
```

Function `DoMakeDocument()` is in the protected section. In a normal program, it is only used from within `HandleCommand()` and consequently it does not need to be public. It is not made private because it is possible that it might need to be called from a function defined in some specialized subclass of class `Application`. Because function `DoMakeDocument()` is pure virtual, class `Application` is still abstract. Real programs must define a specialized subclass of class `Application`.

Document

Class `Document` is a substantially more elaborate kind of `CommandHandler`; Figure 30.5 is a design diagram for the class.

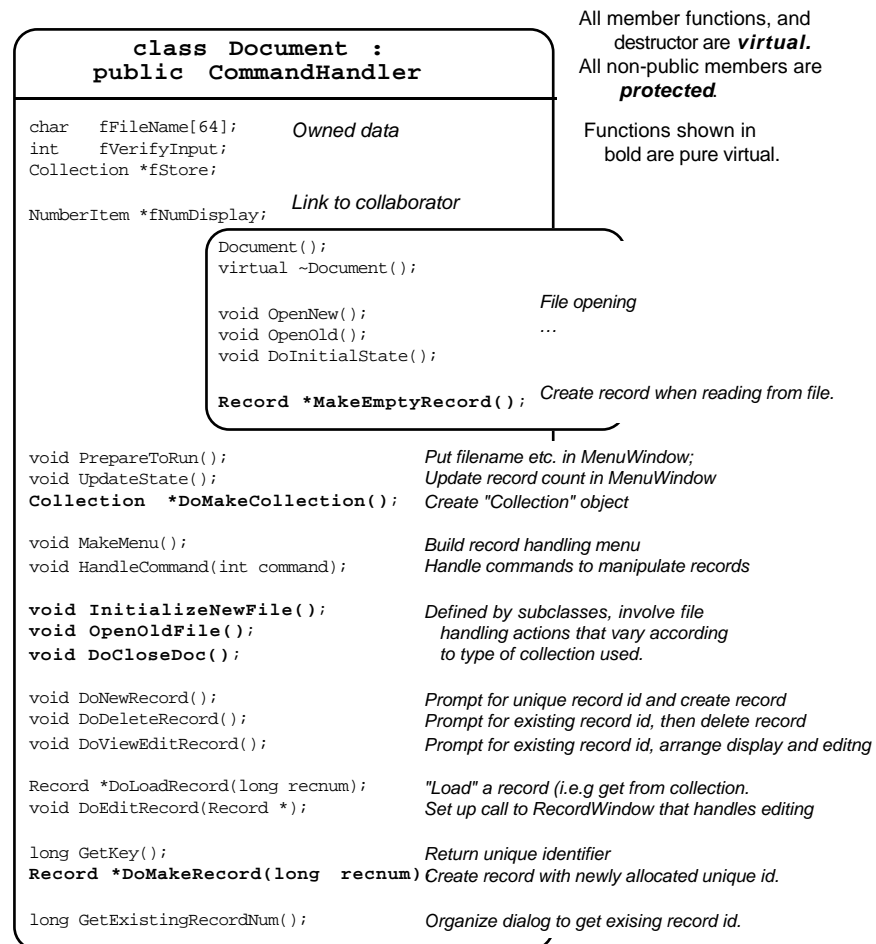


Figure 30.5 Design diagram for class Document.

Document's data members

As shown in Figure 30.5, in addition to the data members that it has because it is a CommandHandler (e.g. the MenuWindow) a Document has, as owned data members, the file "base name" (a character array), an integer flag whose setting determines whether file names should be verified, and a Collection object separately created in the heap and accessed via the fStore pointer. (Of course, this will point to an instance of some specialized subclass of class Collection.) A Document also has a link a NumberItem; this (display window) object gets created by the Document but ownership is transferred to the MenuWindow.

In some cases, the contents of fFileName will be the actual file name. But in other cases, e.g. with the BTree storage structure, there will be separate index and data files and fFileName is used simply as a base name. If a single file is used for storage, then file names can be checked when opening an old file.

All the member functions, and the destructor, are virtual so as to allow redefinition as needed by subclasses. Quite a few functions are still pure virtual;

examples include the functions for creating `Record` objects and some of those involved in file handling. The data members, and auxiliary member functions, are all `protected` (rather than `private`). Again, this is done so as to maximize the potential for adaptation in subclasses.

The public interface defines a few additional functions; most of these involve file handling and related initialization and are used in the code of `Application::HandleCommand()`.

```
class Document : public CommandHandler {
public:
    Document();
    virtual ~Document();

    virtual void      OpenNew();
    virtual void      OpenOld();
    virtual void      DoInitialState();
```

The other additional public function is one of two used to create records. Function `MakeEmptyRecord()` is used (by a `Collection` object) to create an empty record that can then be told to read data from a file:

```
virtual Record      *MakeEmptyRecord() = 0;
```

The other record creating function, `DoMakeRecord()` is used from within `Document::DoNewRecord()`; it creates a record with a new identifier. Since it is only used from within class `Document`, it is part of the protected interface.

```
protected:
    virtual void      PrepareToRun();
    virtual void      UpdateState();

    virtual void      MakeMenu();
    virtual void      HandleCommand(int command);
```

*Redefining empty and
pure virtual functions
inherited from
CommandHandler*

The default implementations of the protected functions `PrepareToRun()` and `UpdateState()` simply arrange for the initial display, and later update, of the information with the filename and the number of records.

Functions `MakeMenu()` and `HandleCommand()` define and work with the standard menu with its options for creating, viewing, and deleting records. These definitions cover for the pure virtual functions defined in the base `CommandHandler` class.

Class `Document` has two functions that use dialogs to get keys for records. Function `GetKey()` is used to get a new unique key, while `GetExistingRecordNum()` is used to get a key already associated with a record. Keys are restricted to positive non zero integers. Function `GetKey()` checks whether the given key is new by requesting the `Collection` object to find a record with the given identifier number. The function fails (returns -1) if the `Collection` object successfully finds a record with the given identifier (an "alert" can then be displayed warning the user that the identifier is already in use).

*Dialogs for getting
record identifiers*

	virtual long	GetKey();
	virtual long	GetExistingRecordNum();
Default definitions of command handling functions	<p>Class Document can provide default definitions for the functions that handle "new record", "view record" and "delete record" commands. Naturally, these commands are handled using auxiliary member functions called from HandleCommand():</p>	
	virtual void	DoNewRecord();
	virtual void	DoDeleteRecord();
	virtual void	DoViewEditRecord();
	<p>For example, DoViewEditRecord() can define the standard behaviour as involving first a prompt for the record number, handling the case of an invalid record number through an alert while a valid record number leads to calls to "load" the record and then the invocation of DoEditRecord().</p>	
Loading and editing records	virtual Record	*DoLoadRecord(long recnum);
	virtual void	DoEditRecord(Record *r);
Pure virtual member functions	<p>Function DoEditRecord() can ask the Record to create its own RecordWindow display and then arrange for this RecordWindow to handle subsequent input (until an "enter", '\n', character is used to terminate that interaction).</p> <p>The remaining member functions are all pure virtual. The function that defines the type of Collection to use is defined by a subclass defined in the framework, e.g. ArrayDoc or BTDoc. These classes also provide effective definitions for the remaining file handling functions.</p>	
Functions provided in framework subclasses	virtual Collection	*DoMakeCollection() = 0;
	virtual void	InitializeNewFile() = 0;
	virtual void	OpenOldFile() = 0;
	virtual void	DoCloseDoc() = 0;
Program specific function	<p>Function DoMakeRecord(), like the related public function MakeEmptyRecord(), must be defined in a program specific subclass ("MyDoc" etc).</p>	
	virtual Record	*DoMakeRecord(long recnum) = 0;
Data members	<p>The class declaration ends with the specification of the data members and links to collaborators:</p>	
	char	fFileName[64];
	NumberItem	*fNumDisplay;
	Collection	*fStore;
	int	fVerifyInput;
	};	

Classes BTDoc and ArrayDoc

Classes BTDoc and ArrayDoc are generally similar. They have to provide implementations of the various file handling and collection creation functions declared in class Document. Class BTDoc has the following declaration:

```
class BTDoc : public Document {
public:
    BTDoc();
protected:
    virtual Collection *DoMakeCollection();
    virtual void    InitializeNewFile();
    virtual void    OpenOldFile();
    virtual void    DoCloseDoc();
};
```

The constructor for class BTDoc simply invokes the inherited Document constructor and then changes the default setting of the "file name verification" flag (thus switching off verification). Since a BTree uses multiple files, the input file dialog can't easily check the name.

The declaration of class ArrayDoc is similar. Since it doesn't require any special action in its constructor, the interface consists of just the three protected functions.

30.3.2 Interactions

Principal interactions when opening a file

Figure 30.6 illustrates some of the interactions involved when an application object deals with an "Open" command from inside its HandleCommand() function. The illustration is for the case where the concrete "MyDoc" document class is derived from class BTDoc.

All the interactions shown in Figure 30.6 are already implemented in the framework code. A program built using the framework simply inherits the behaviour patterns shown

Inherited pattern of interactions

The only program specific aspect is the implementation of the highlighted call to DoMakeDocument(). This call to DoMakeDocument() is the first step in the process; it results in the creation of the specialized MyDoc object.

A single program specific function call

Once created, the new document object is told to perform its DoInitialState() routine in which it creates a Collection object. Since the supposed MyDoc class is derived from class BTDoc, this step results in a new BTDCollection.

Create the collection

The next step, opening the file, is relatively simple in the case of a BTDoc. First the document uses a dialog to get the file name; an InputFileDialog object is created temporarily for this purpose. Once the document has the file name, it proceeds by telling its BTreeCollection to "open the BTree". This step leads to the creation of the actual BTree object whose constructor opens both index and data files.

Open the file

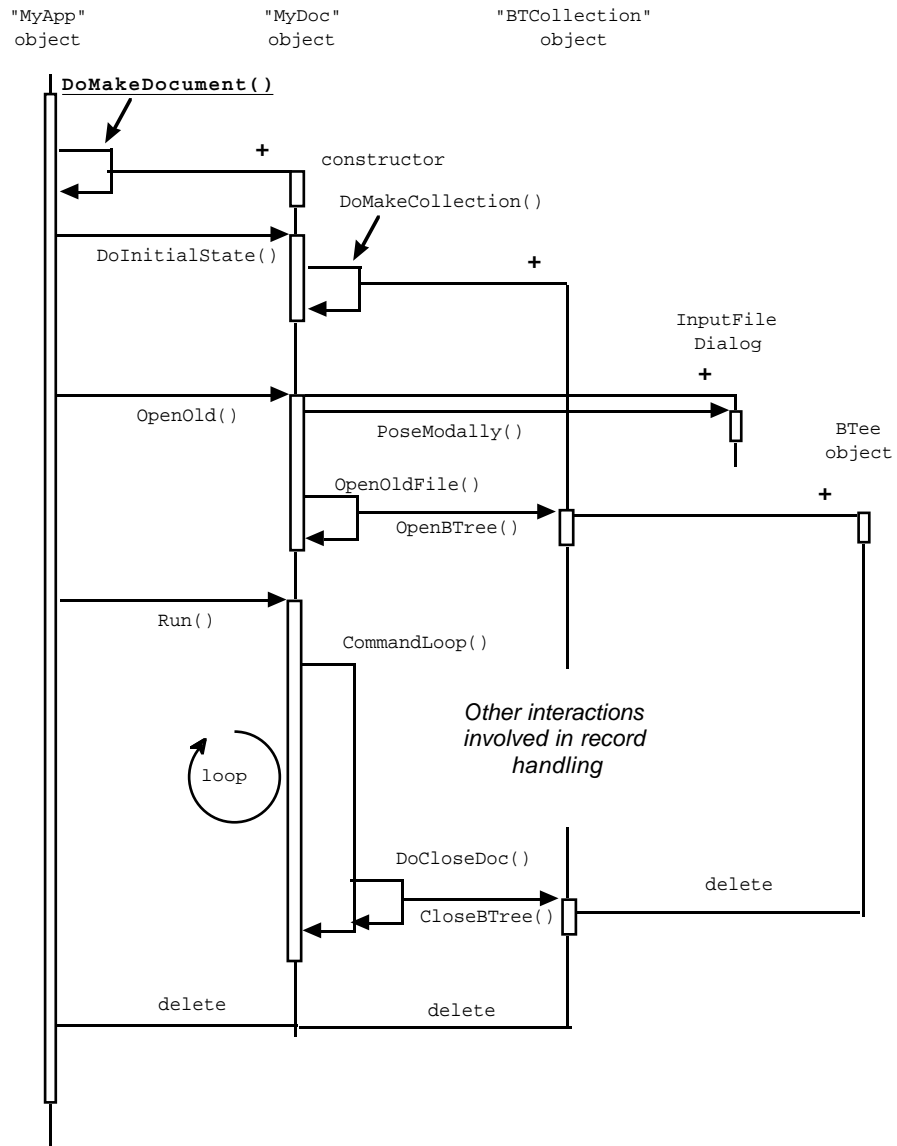


Figure 30.6 Framework defined interactions for an Application object handling an "Open" command.

The next step would get the document to execute its **Run()** function. This would involve processing any record handling commands (none shown in Figure 30.6) and eventually a "Close" command.

Closing A "Close" command gets forwarded to the collection. This responds by deleting the **BTree** object (whose destructor arranges to save all "housekeeping data" and then close its two files).

A return is then made from `Document::Run()` back to `Application::HandleCommand()`. Since the document object is no longer needed, it gets deleted. As shown in Figure 30.6 this also leads to the deletion of the `BTCollection` object.

Additional interactions for a memory based collection

A document based on an "in memory" storage structure would have a slightly more elaborate pattern of interactions because it has to load the existing records when the file is opened. The overall pattern is similar. However, as shown in Figure 30.7, the `OpenOldFile()` function results in additional interactions.

In this case, the collection object (an `ADCollection`) will be told to read all its data from the file. This will involve control switching back and forth among several objects as shown in Figure 30.7. The collection object would read the number of records and then have a loop in which records get created, read their own data, and are then added to the `DynamicArray`. The collection object has to ask the document object to actually create the new `Record` (this step, and the `MyRec::ReadFrom()` function, are the only program specific parts of the pattern).

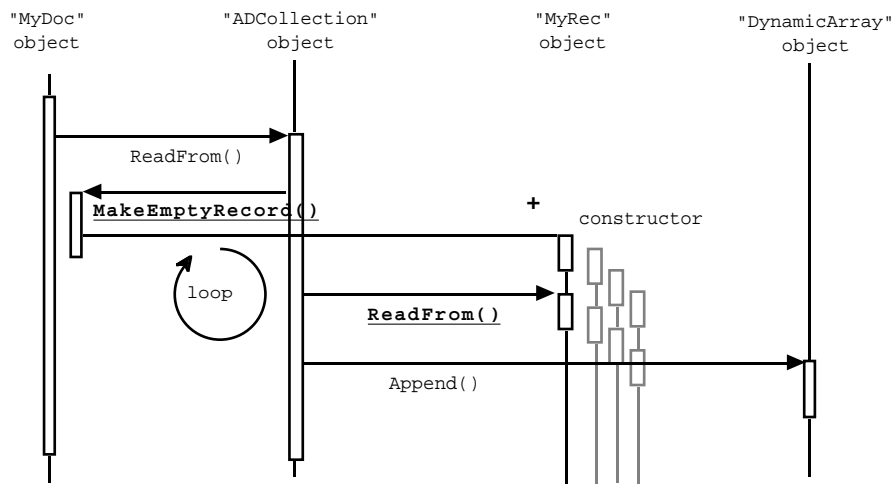


Figure 30.7 Opening a document whose contents are memory resident (interactions initiated by a call to `ArrayDoc::OpenOldFile()`).

Interactions while creating a new record

Figure 30.8 illustrates another pattern of interactions among class instances that is almost entirely defined within the framework code. It shows the overall steps involved in creating a new record (`Document::DoNewRecord()` function).

The interactions start with the document object using a `NumberDialog` object to get a (new) record number from the user. The check to determine that the number is new involves a request to the collection object to perform a `Find()` operation (this `Find()` should fail).

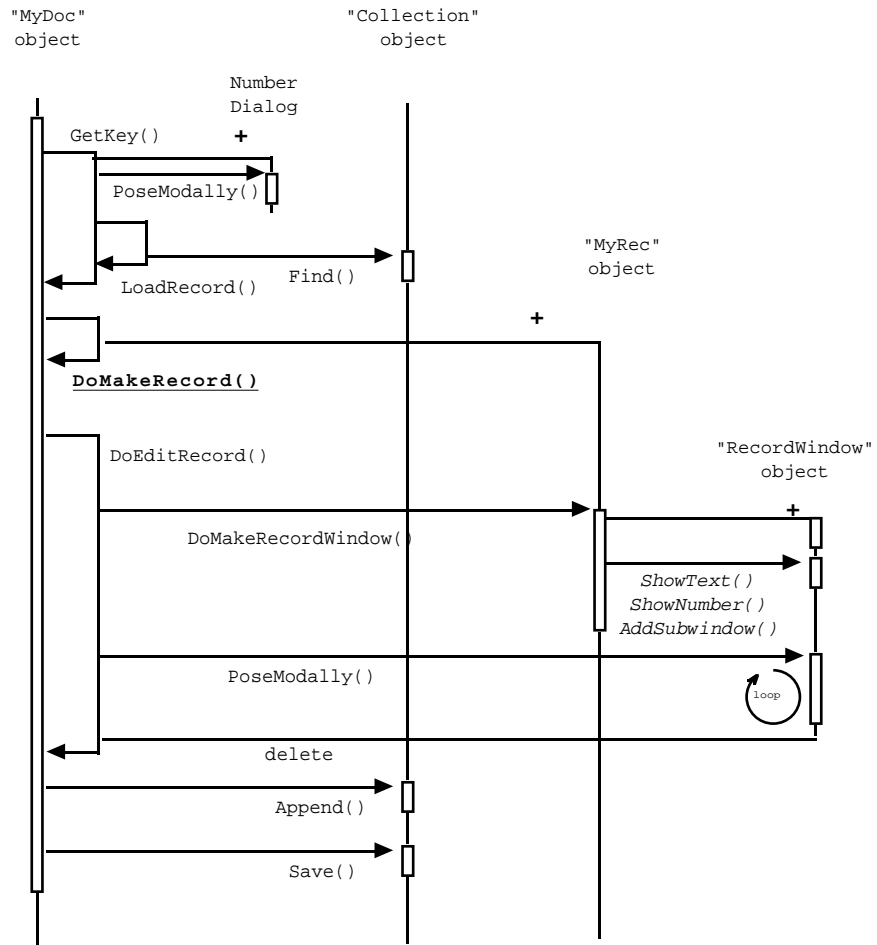


Figure 30.8 Some of the interactions resulting from a call to Document::DoNewRecord().

If the newly entered record identifier number is unique, a new record is created with the given record number. This involves the program specific implementation of the function `MyDoc::DoMakeRecord()`.

The next step involves the new `MyRec` object being asked to create an appropriate `RecordWindow`. The object created will be a standard `RecordWindow`; but the code for `MyRec::DoMakeRecordWindow()` will involve program specific actions adding a specific set of text labels and editable fields.

Once created, the `RecordWindow` will be "posed modally". It has a loop dealing with subsequent input. In this loop it will interact with the `MyRec` object (notifying it of any changes to editable fields) as well as with the editable subwindows that it contains.

When control is returned from `RecordWindow::PoseModally()`, the `MyRec` object will already have been brought up to date with respect to any changes. The

`RecordWindow` can be deleted. The new record can then be added to the `Collection` (the `Append()` operation).

Although the new record was created by the document object, it now belongs to the collection. The call to `Save()` represents an explicit transfer of ownership. Many collections will have empty implementations for `Save()` (because they don't have to do anything special with respect to ownership). A collection that uses a `BTree` will delete the memory resident record because it will already have made a permanent copy on disk (during its `Append()` operation).

30.3.3 Implementation Code

CommandHandler

The constructor for class `CommandHandler` sets its state as "unfinished" and creates a `MenuWindow`. (All `Window` objects have integer identifiers. In the current code, these identifiers are only used by `RecordWindow` and `Record` objects.)

```
CommandHandler::CommandHandler(int mainmenuid)
{
    fMenuID = mainmenuid;
    fMenu = new MenuWindow(fMenuID);
    fFinished = 0;
}
```

The destructor naturally gets rid of the `MenuWindow`.

```
CommandHandler::~~CommandHandler()
{
    delete fMenu;
}
```

The `Run()` function completes the construction of the `MenuWindow` and other initial preparations and then invokes the `CommandLoop()` function. When this returns, any special tidying up operations are performed in `Finish()`.

```
void CommandHandler::Run()
{
    this->MakeMenu();
    this->PrepareToRun();
    this->CommandLoop();
    this->Finish();
}
```

It may appear that there is something slightly odd about the code. The `MenuWindow` is created in the `CommandHandler`'s constructor, but the (virtual) function `MakeMenu()` (which adds menu items to the window) is not called until the start of `Run()`. It might seem more natural to have the call to `MakeMenu()` as part of the `CommandHandler`'s constructor.

However, that arrangement does not work. *Constructors do not use dynamic calls to virtual functions.* If the call to `MakeMenu()` was part of the `CommandHandler`'s constructor, it would be the (non existent) `CommandHandler::MakeMenu()` function that was invoked and not the desired `Application::MakeMenu()` or `Document::MakeMenu()` function.

Beware: no "virtual constructors"

Constructors that do invoke virtual functions dynamically (so called "virtual constructors") are a frequently requested extension to C++ but, for technical reasons, they can not be implemented.

Class `CommandHandler` provides the default implementation for one other function, the main `CommandLoop()`:

```
void CommandHandler::CommandLoop()
{
    while(!fFinished) {
        this->UpdateState();
        int c = fMenu->PoseModally();
        this->HandleCommand(c);
    }
}
```

Application

Class `Application` is straightforward. Its constructor and destructor add nothing to those defined by the base `CommandHandler` class:

```
Application::Application() : CommandHandler(kAPPMENU_ID)
{
}

Application::~~Application()
{
}
```

Its `MakeMenu()` function adds the three standard menu items to the `MenuWindow`. (Constants like `kAPPEMENU_ID`, `cNEW` and related constants used by class `Document` are all defined in a common header file. Common naming conventions have constants that represent "commands" take names starting with 'c' while those that define other parameters have names that begin with 'k'.)

```
void Application::MakeMenu()
{
    fMenu->AddMenuItem("New", cNEW);
    fMenu->AddMenuItem("Open", cOPEN);
    fMenu->AddMenuItem("Quit", cQUIT);
}
```

The code implementing `Application::HandleCommand()` was given earlier (in Section 30.2).

Document

The constructor for class `Document` has a few extra data members to initialize. Its destructor gets rid of the `fStore` (`Collection` object) if one exists.

```
Document::Document() : CommandHandler(kDOCUMENT_ID)
{
    fStore = NULL;
    fFileName[0] = '\0';
    fVerifyInput = 1;
}
```

The `MakeMenu()` function adds the standard menu options to the document's `MenuWindow()` function. Member `DoInitialState()` simply uses the (pure virtual) `DoMakeCollection()` function to make an appropriate `Collection` object. The function actually called would be `BTDoc::DoMakeCollection()` or `ArrayDoc::DoMakeCollection` (or other similar function) depending on the particular type of document that is being built.

```
void Document::MakeMenu()
{
    fMenu->AddMenuItem("New record", cNEWREC);
    fMenu->AddMenuItem("Delete record", cDELREC);
    fMenu->AddMenuItem("View/edit record", cVIEW);
    fMenu->AddMenuItem("Close file", cCLOSE);
}

void Document::DoInitialState()
{
    fStore = DoMakeCollection();
}
```

*Initialization
functions*

Functions `PrepareToRun()` and `UpdateState()` deal with the initial display and subsequent update of the display fields with document details. (The `NumberItem` used for output is given to the `MenuWindow` as a "subwindow". As explained in Section 30.6, subwindows "belong" to windows and, when appropriate, get deleted by the window. So although the `Document` creates the `NumberItem` and keeps a link to it, it never deletes it.)

*Display of document
details*

```
void Document::PrepareToRun()
{
    fNumDisplay = new NumberItem(0, 31, 1, 40,
                                "Number of records:", 0);
    fMenu->AddSubWindow(fNumDisplay);
}

void Document::UpdateState()
{
    fNumDisplay->SetVal(fStore->Size());
    fMenu->ShowText(fFileName, 2, 2, 30, 0, 1);
}
```


Getting keys for new and existing records

The functions `GetKey()` and `GetExistingRecordNum()` both use dialogs for input. Function `GetKey()`, via the `DoLoadRecord()` function, lets the document interact with the `Collection` object to verify that the key is not already in use:

```
long Document::GetKey()
{
    NumberDialog n("Record identifier", 1, LONG_MAX);
    long k = n.PoseModally(1);
    if(NULL == DoLoadRecord(k))
        return k;
    Alert("Key already used");
    return -1;
}

long Document::GetExistingRecordNum()
{
    if(fStore->Size() == 0) {
        Alert("No records defined.");
        return -1;
    }
    NumberDialog n("Record number", 1, LONG_MAX);
    return n.PoseModally(1);
}

Record *Document::DoLoadRecord(long recnum)
{
    return fStore->Find(recnum);
}
```

Provision for further extension

Function `DoLoadRecord()` might seem redundant, after all the request to `fStore` to do a `Find()` operation could have been coded at the point of call. Again, this is just a point where provision for modification has been built into the framework. For example, someone working with a disk based collection of records might want to implement a version that maintained a small "cache" or records in memory. He or she would find it convenient to have a redefinable `DoLoadRecord()` function because this would represent a point where the "caching" code could be added.

Function `Alert()` is defined along with the windows code. It puts up a dialog displaying an error string and waits for the user to press the "enter" key.

Running and handling commands

Class `Document` uses the inherited `CommandHandler::Run()` function. Its own `HandleCommand()` function was given earlier (Section 30.2). `Document::HandleCommand()` is implemented in terms of the auxiliary member functions `DoNewRecord()`, `DoDeleteRecord()`, `DoViewEditRecord()`, and `DoCloseDoc()`. Function `DoCloseDoc()` is pure virtual but the others have default definitions.

Functions `DoNewRecord()` and `DoEditRecord()` implement most of the interactions shown in Figure 30.8; getting the key, making the record, arranging for it to be edited through a temporary `RecordWindow` object, adding and transferring ownership of the record to the `Collection` object:

```
void Document::DoNewRecord()
{
    long key = GetKey();
```

```
        if(key<=0)
            return;
        Record *r = DoMakeRecord(key);
        DoEditRecord(r);
        fStore->Append(r);
        fStore->Save(r);
    }

    void Document::DoEditRecord(Record *r)
    {
        RecordWindow *rw = r->DoMakeRecordWindow();
        rw->PoseModally();
        delete rw;
    }
```

Function `DoDeleteRecord()` gets a record identifier and asks the `Collection` object to delete that record. (The `Collection` object is expected to return a success indicator. A failure results in a warning to the user that the key given did not exist.)

```
void Document::DoDeleteRecord()
{
    long recnum = GetExistingRecordNum();
    if(recnum<0)
        return;
    if(!fStore->Delete(recnum))
        Alert(NoRecMsg);
}
```

Function `DoViewEditRecord()` first uses a dialog to get a record number, and then "loads" that record. If the load operation fails, the user is warned that the record number was invalid. If the record was loaded, it can be edited:

```
void Document::DoViewEditRecord()
{
    long recnum = GetExistingRecordNum();
    if(recnum<0)
        return;
    Record *r = DoLoadRecord(recnum);
    if(r == NULL) {
        Alert(NoRecMsg);
        return;
    }
    DoEditRecord(r);
    fStore->Save(r);
}
```

Functions `OpenNew()` and `OpenOld()` handle standard aspects of file opening (such as display of dialogs that allow input of a file name). Aspects that depend on the type of collection structure used are handled through the auxiliary (pure virtual) functions `InitializeNewFile()` and `OpenOldFile()`:

*Standard file
handling*

```
void Document::OpenNew()
{
    TextDialog onew("Name for new file");
```

```

        onew.PoseModally("example",fFileName);
        InitializeNewFile();
    }

    void Document::OpenOld()
    {
        InputFileDialog oold;
        oold.PoseModally("example",fFileName, fVerifyInput);
        OpenOldFile();
    }

```

ArrayDoc

Class ArrayDoc has to provide implementations for the DoMakeCollection() and the related file manipulation functions.

It creates an ADCollection (a DynamicArray within an "adapter"):

Creating a collection object

```

Collection *ArrayDoc::DoMakeCollection()
{
    return new ADCollection(this);
}

```

The OpenOldFile() function has to actually open the file for input, then it must get the ADCollection to load all the data. Note the typecast on fStore. The type of fStore is Collection*. We know that it actually points to an ADCollection object. So we can use the typecast to get an ADCollection* pointer. Then it is possible to invoke the ADCollection::ReadFrom() function:

Handling file transfers

```

void ArrayDoc::OpenOldFile()
{
    fstream in;
    in.open(fFileName, ios::in | ios::nocreate);
    if(!in.good()) {
        Alert("Bad file");
        exit(1);
    }
    ((ADCollection*)fStore)->ReadFrom(in);
    in.close();
}

```

Function DoCloseDoc() is very similar. It opens the file for output and then arranges for the ADCollection object to save the data (after this, the collection has to be told to delete all its contents):

```

void ArrayDoc::DoCloseDoc()
{
    fstream out;
    out.open(fFileName, ios::out);
    if(!out.good()) {
        Alert("Can not open output");
        return;
    }
}

```

```
((ADCollection*)fStore)->WriteTo(out);
out.close();

((ADCollection*)fStore)->DeleteContents();

}
```

BTDoc

As noted earlier, a BTDoc needs to change the default setting of the "verify file names" flag (normally set to true in the Document constructor):

```
BTDoc::BTDoc() { fVerifyInput = 0; }
```

Its DoMakeCollection() function naturally makes a BTCollection object (the this argument provides the collection object with the necessary link back to its document):

```
Collection *BTDoc::DoMakeCollection()
{
    return new BTCollection(this);
}
```

The other member functions defined for class BTDoc are simply an interface to functions provided by the BTCollection object:

```
void BTDoc::InitializeNewFile()
{
    ((BTCollection*)fStore)->OpenBTree(fFileName);
}

void BTDoc::OpenOldFile()
{
    ((BTCollection*)fStore)->OpenBTree(fFileName);
}

void BTDoc::DoCloseDoc()
{
    ((BTCollection*)fStore)->CloseBTree();
}
```

*Let the BTree object
handle the files*

30.4 COLLECTION CLASSES AND THEIR ADAPTERS

The underlying collection classes are identical to the versions presented in earlier chapters. Class DynamicArray, as used in ADCollection, is as defined in Chapter 21. (The Window classes also use instances of class DynamicArray.) Class BTCollection uses an instance of class BTree as defined in Chapter 24.

Class Collection itself is purely an interface class with the following declaration:

**Abstract base class
collection**

```

class Collection {
public:
    Collection(Document *d) { this->fDoc = d; }
    virtual ~Collection() { }

    virtual void      Append(Record *r) = 0;
    virtual int       Delete(long recnum) = 0;
    virtual Record    *Find(long recnum) = 0;
    virtual void      Save(Record *r) { }

    virtual long      Size() = 0;
protected:
    Document          *fDoc;
};

```

It should get defined in the same file as class Document. A Collection is responsible for getting Record objects from disk. It has to ask the Document object to create a Record of the appropriate type; it is for this reason that a Collection maintains a link back to the Document to which it belongs.

The declarations for the two specialized subclasses are:

**Specialization using
an array**

```

class ADCollection : public Collection {
public:
    ADCollection(ArrayDoc *d) : Collection(d) { }

    virtual void      Append(Record *r);
    virtual int       Delete(long recnum);
    virtual Record    *Find(long recnum);
    virtual long      Size();

    virtual void      ReadFrom(fstream& fs);
    virtual void      WriteTo(fstream& fs);
    virtual void      DeleteContents();

protected:
    DynamicArray      fD;
};

```

and

**Specialization using
a BTree**

```

class BTCollection : public Collection {
public:
    BTCollection(BTDoc *d) : Collection(d) { }

    virtual void      Append(Record *r);
    virtual int       Delete(long recnum);
    virtual Record    *Find(long recnum);

    virtual long      Size();
    virtual void      Save(Record *r);

    void              OpenBTree(char* filename);
    void              CloseBTree();
};

```

```
private:
    BTree    *fBTree;
};
```

Each class defines implementations for the basic `Collection` functions like `Append()`, `Find()`, `Size()`. In addition, each class defines a few functions that relate to the specific form of storage structure used.

In the case of `Append()` and `Size()`, these collection class adaptors can simply pass the request on to the actual collection used:

```
void ADCollection::Append(Record *r) { fD.Append(r); }

void BTCollection::Append(Record *r)
{
    fBTree->Add(*r);
}

long ADCollection::Size()
{
    return fD.Length();
}

long BTCollection::Size()
{
    return fBTree->NumItems();
}
```

A `DynamicArray` doesn't itself support record identifiers, so the `Find()` operation on an `ADCollection` must involve an inefficient linear search:

Adapting a dynamic array to fulfil the role of Collection

```
Record *ADCollection::Find(long recnum)
{
    int n = fD.Length();
    for(int i = 1; i<=n; i++) {
        Record *r = (Record*) fD.Nth(i);
        long k = r->Key();
        if(k == recnum)
            return r;
    }
    return NULL;
}
```

This `Find()` function has to be used to implement `Delete()` because the record must first be found before the `DynamicArray::Remove()` operation can be used. (Function `Delete()` takes a record identifier, `Remove()` requires a pointer).

```
int ADCollection::Delete(long recnum)
{
    Record *r = Find(recnum);
    if(r != NULL) {
        fD.Remove(r);
        return 1;
    }
    else return 0;
}
```

*Adapting a BTree to
fulfil the role of
Collection*

```
}
```

The linear searches involved in most operations on an `ADCollection` make this storage structure unsuitable for anything apart from very small collections. A more efficient "in-memory" structure could be built using a binary tree or an AVL tree.

The corresponding functions for the `BTCollection` also involve work in addition to the basic `Find()` and `Remove()` operations on the collection. Function `BTCollection::Find()` has to return a pointer to an in-memory record. All the records in the `BTree` itself are on disk. Consequently, `Find()` had better create the in-memory record. This gets filled with data from the disk record (if it exists). If the required record is not present (the `BTree::Find()` operation fails) then the newly created record should be deleted.

```
Record *BTCollection::Find(long recnum)
{
    Record *r = fDoc->MakeEmptyRecord();
    int success = fBTree->Find(recnum, *r);
    if(success)
        return r;

    delete r;
    return NULL;
}
```

(The record is created via a request back to the document object: `fDoc->MakeEmptyRecord();`)

The function `BTree::Remove()` does not return any success or failure indicator; "deletion" of a non-existent key fails silently. A `Collection` is supposed to report success or failure. Consequently, the `BTCollection` has to do a `Find()` operation on the `BTree` prior to a `Remove()`. If this initial `BTree::Find()` fails, the `BTCollection` can report a failure in its delete operation.

```
int BTCollection::Delete(long recnum)
{
    Record *r = Find(recnum);
    if(r != NULL) {
        delete r;
        fBTree->Remove(recnum);
        return 1;
    }
    else return 0;
}
```

The other member functions for these classes mostly relate to file handling. Function `ADCollection::ReadFrom()` implements the scheme shown in Figure 30.7 for loading the entire contents of a collection into memory:

```
void ADCollection::ReadFrom(fstream& fs)
{
    long len;
    fs.read((char*)&len, sizeof(len));
    for(int i = 1; i <= len; i++) {
```

```

        Record *r = fDoc->MakeEmptyRecord();
        r->ReadFrom(fs);
        fD.Append(r);
    }
}

```

The `WriteTo()` function handles the output case, getting called when a document is closed. It writes the number of records, then loops getting each record to write its own data:

```

void ADCollection::WriteTo(fstream& fs)
{
    long len = fD.Length();
    fs.write((char*)&len, sizeof(len));
    for(int i = 1; i <= len; i++) {
        Record *r = (Record*) fD.Nth(i);
        r->WriteTo(fs);
    }
}

```

A `DynamicArray` does not delete its contents. (It can't really. It only has `void*` pointers). When a document is finished with, all in memory structures should be freed. The `ADCollection` has to arrange this by explicitly removing the records from the `DynamicArray` and deleting them.

```

void ADCollection::DeleteContents()
{
    int len = fD.Length();
    for(int i = len; i >= 1; i--) {
        Record* r = (Record*) fD.Remove(i);
        delete r;
    }
}

```

(The code given in Chapter 21 for class `DynamicArray` should be extended to include a destructor that does get rid of the associated array of `void*` pointers.)

The remaining functions of class `BTCollection` are all simple. Function `OpenBTree()` creates the `BTree` object itself (letting it open the files) while `CloseBTree()` deletes the `BTree`.

```

void BTCollection::OpenBTree(char* filename)
{
    fBTree = new BTree(filename);
}

void BTCollection::CloseBTree()
{
    delete fBTree;
}

```

Class `BTCollection` provides an implementation for `Save()`. This function is called whenever the `Document` object has finished with a `Record` object. In the

case of an in-memory collection, like `ADCollection`, no action is required. But with the `BTCollection`, it is necessary to get rid of the now redundant in-memory record.

```
void BTCollection::Save(Record *r)
{
    delete r;
}
```

30.5 CLASS RECORD

As noted earlier, class `KeyedStorableItem` is simply an interface (it is the interface for storable records as defined for class `BTree` in Chapter 24):

```
class KeyedStorableItem {
public:
    virtual ~KeyedStorableItem() { }
    virtual long Key(void) const = 0;
    virtual void PrintOn(ostream& out) const { }
    virtual long DiskSize(void) const = 0;
    virtual void ReadFrom(fstream& in) = 0;
    virtual void WriteTo(fstream& out) const = 0;
};
```

Class `Record` provides a default implementation for the `Key()` function and adds the responsibilities related to working with a `RecordWindow` that allows editing of the contents of data members (as defined in concrete subclasses).

```
class Record : public KeyedStorableItem {
public:
    Record(long recNum) { this->fRecNum = recNum; }
    virtual ~Record() { }
    virtual long Key() const { return this->fRecNum; }

    virtual RecordWindow *DoMakeRecordWindow();
    virtual void SetDisplayField(EditWindow *e);
    virtual void ReadDisplayField(EditWindow *e);
    virtual void ConsistencyUpdate(EditWindow *e) { }
protected:
    virtual void CreateWindow();
    virtual void AddFieldsToWindow();
    long fRecNum;
    RecordWindow *fRW;
};
```

*Data members
defined by class
Record*

Class `Record` defines two data members itself. One is a long integer to hold the unique identifier (or "key"), the other is a link to the `RecordWindow` collaborator.

Function `DoMakeRecordWindow()` uses an auxiliary function `CreateWindow()` to actually create the window. Once created, the window is "populated" by adding subwindows (in `AddFieldsToWindow()`).

```
RecordWindow *Record::DoMakeRecordWindow()
{
    CreateWindow();
    AddFieldsToWindow();
    return fRW;
}
```

Function `CreateWindow()` is another "unnecessary" function introduced to increase the flexibility of the framework. It is unlikely that any specific program would need to change the way windows get created, but it is possible that some program might need to use a specialized subclass of `RecordWindow`. Having the window creation step handled by a separate virtual function makes adaptation easier.

```
void Record::CreateWindow()
{
    fRW = new RecordWindow(this);
}
```

Function `Record::AddFieldsToWindow()` can add any "background" text labels to the `RecordWindow`. A concrete "MyRec" class would also add `EditNum` and `EditText` subwindows for each editable field; an example is given later in Section 30.8. Each of these subwindows is specified by position, a label (possibly empty) and a unique identifier. These window identifiers are used in subsequent communications between the `RecordWindow` and `Record` objects.

```
void Record::AddFieldsToWindow()
{
    fRW->ShowText("Record identifier ", 2, 2, 20, 0);
    fRW->ShowNumber(fRecNum, 24, 2, 10);
    // Then add data fields

    // typical code
    // EditText *et = new EditText(1001, 5, 4, 60,
    //     "Student Name ");
    //fRW->AddSubWindow(et);
    //EditNum *en = new EditNum(1002, 5, 8, 30,
    //     "Assignment 1 (5) ", 0, 5,1);
    //fRW->AddSubWindow(en);
}
```

Building the display structure

When a `RecordWindow` is first displayed, it will ask the corresponding `Record` to set values in each of its editable subwindows. This is done by calling `Record::SetDisplayField()`. The `EditWindow` passed via the pointer parameter can be asked its identity so allowing the `Record` object to select the appropriate data to be used for initialization:

```
void Record::SetDisplayField(EditWindow *e)
{
    // Typical code:
    // long id = e->Id();
    // switch(id) {
```

Showing current data values

```

//case 1001:
//      ((EditText*)e)->SetVal(fStudentName, 1);
//      break;
//case 1002:
//      ((EditNum*)e)->SetVal(fMark1,1);
//      break;
//      ...
}

```

Function `Record::ReadDisplayField()` is called when the contents of an `EditWindow` have been changed. The `Record` can identify which window was changed and hence determine which data member to update:

Getting newly edited values

```

void Record::ReadDisplayField(EditWindow *e)
{
    // Typical code:
    // long id = e->Id();
    // switch(id) {
    //case 1001:
    //      char* ptr = ((EditText*)e)->GetVal();
    //      strcpy(fStudentName, ptr);
    //      break;
    //case 1002:
    //      fMark1 = ((EditNum*)e)->GetVal();
    //      break;
    //      ...
    }
}

```

Interactions between `Record` and `RecordWindow` objects are considered in more detail in the next section (see Figure 30.9).

30.6 THE WINDOWS CLASS HIERARCHY

30.6.1 Class Responsibilities

WindowRep

The role of the unique `WindowRep` object is unchanged from that in the version given in Chapter 29. It "owns" the screen and deals with the low level details involved in input and output of characters.

The version used for the "RecordFile" framework has two small extensions. There is an extra public function, `Beep()`; this function (used by some dialogs) produces an audible tone to indicate an error. (The simplest implementation involves outputting '\a' "bell" characters.)

The other extension arranges for any characters input via `GetChar()` to be copied into the `fImage` array. This improves the consistency of screen updates that occur after data input.

The implementation of this extension requires additional `fxC`, `fyC` integer data members in class `WindowRep`. These hold the current position of the cursor. They

are updated by functions like `MoveCursor()` and `PutCharacter()`. Function `GetChar()` stores an input character at the point `(fXC, fYC)` in the `fImage` array.

These extensions are trivial and the code is not given.

Window

Class `Window` is an extended version of that given in Chapter 29. The major extension is that a `Window` can have a list (dynamic array) of "subwindows" and so has some associated functionality. In addition, `Window` objects have integer identifiers and there are a couple of extra member functions for things like positioning the cursor and outputting a string starting at a given point in the `Window`.

The additional (protected) data members are:

```
DynamicArray    fSubWindows;  "List of subwindows"
int             fId;          "Identifier"
```

The following functions are added to the public interface:

```
int             Id() const { return this->fId; }           Window identifier
```

Function `Id()` returns the `Window` object's identifier. The constructor for class `Window` is changed so that this integer identifier number is an extra (first) parameter. (The destructor is extended to get rid of subwindows.)

```
void            ShowText(const char* str, int x, int y,    Utility output
                    int width, int multiline = 0, int bkgd = 1); functions
void            ShowNumber(long value, int x, int y, int width,
                    int bkgd = 1);
void            ClearArea(int x0, int y0, int x1, int y1,
                    int bkgd);
```

These functions are just generally useful output functions. Function `ShowText()` outputs a string, possibly on multiple lines, starting at the defined `x, y` position. Function `ShowNumber()` converts a number to a text string and uses `ShowText()` to output this at the given position. Function `ClearArea()` fills the specified area with ' ' characters. Each of these functions can operate on either the "foreground" or "background" image array of the `Window`. The code for these functions is straightforward and is not given.

```
virtual void SetPromptPos(int x, int y);
```

The default implementation of `SetPromptPos()` involves a call to the `WindowRep` object's `MoveCursor()` function. It is necessary to change the `x, y` values to switch from `Window` coordinates to screen coordinates before invoking `MoveCursor()`.

The function `PrepareToDisplay()` is just an extra "hook" added to increase flexibility in the framework. It gets called just before a `Window` gets displayed allowing for any unusual special case initializations to be performed.

```
virtual void PrepareToDisplay() { }
```

Provision for subwindows

The first of the extra functions needed to support subwindows is `CanHandleInput()`. Essentially this returns "true" if a `Window` object is actually an instance of some class derived from class `EditWindow` where a `GetInput()` function is defined. In some situations, it is necessary to know which (sub)windows are editable so this function has been added to the base class for the entire windows hierarchy. By default, the function returns "false".

```
virtual int CanHandleInput() { return 0; }
```

The three main additions for subwindows are the public functions `AddSubWindow()` and `DisplayWindow()` and the protected function `Offset()`.

```
void AddSubWindow(Window *w);
void Offset(int x, int y);
void DisplayWindow();
```

Member function `AddSubWindow(Window *w)` adds the given `Window w` as a subwindow of the `Window` executing the function. This just involves appending `w` to the "list" `fSubWindows`.

When creating a subwindow, it is convenient to specify its position in terms of the enclosing window. However, the `fx`, `fy` position fields of a `Window` are supposed to be in screen coordinates. The function `Offset()`, called from `AddSubWindow()`, changes the `fx`, `fy` coordinates of a subwindow to take into account the position of its enclosing parent window.

The function `DisplayWindow()`, whose implementation is given later, prepares a window, and its subwindows for display. This involves calls to functions such as `PrepareContent()`, `PrepareToDisplay()`, and `ShowAll()`.

NumberItem

The role of class `NumberItem` is unchanged from that of the version presented in the last chapter; it just displays a numeric value along with a (possibly null) label string.

The constructor now has an additional "int id" argument at the front of the argument list that is used to set a window identifier. Function `SetVal()` also has an extra "int redraw" parameter; if this is false (which is the default) a change to the value does not lead to immediate updating of the screen.

The implementation of `NumberItem` was changed to use the new functionality like `ShowText()` and `ShowNumber()` added to the base `Window` class. The implementation code is not given, being left as an exercise.

EditWindow

The significant changes to the previously illustrated Window classes start with class EditWindow.

Class EditWindow is intended to be simply an abstraction. It represents a window object that can be asked to "get input".

"Getting input" means different things in different contexts. When an EditNum window is "getting input" it consumes digits to build up a number. A MenuWindow "gets input" by using "tab" characters to change the selected option. However, although they differ in detail, the various approaches to "getting input" share a similar overall pattern. *Getting input*

There may have to be some initialization. After this is completed, the actual input step can be performed. Sometimes, it is necessary to validate an input. If an input value is unacceptable the user should be notified and then the complete process should be repeated. *Overall input process*

The actual input step itself will involve a loop in which characters are accepted (via the WindowRep object) and are then processed. Different kinds of EditWindow process different kinds of character; thus a MenuWindow can basically ignore everything except tabs, an EditNum only wants to handle digits, while an EditText can handle more or less any (printable) character. The character input step should continue until some terminator character is entered. (The terminator character may be subclass specific.) The terminator character is itself sometimes significant; it should get returned as the result of the GetInput() function. *Consuming characters*

This overall pattern can be defined in terms of a GetInput() function that works with auxiliary member functions that can be redefined in subclasses. Pseudo-code for this GetInput() function is as follows:

```
InitializeInput();
do {
    SetCursorPosition();
    Get character (ch)
    while(ch != '\n') {
        if(!HandleCharacter(ch))
            break;
        Get character (ch)
    }
    TerminateInput();
    v = Validate();
} while (fMustValidate && !v);
return ch;
```

Inner loop getting characters until terminator

Validate

The outer loop, the do ... while() loop, makes the EditWindow keep on processing input until a "valid" entry has been obtained. (An entry is "valid" if it satisfies the Validate() member function, or if the fMustValidate flag is false).

Prior to the first character input step, the cursor is positioned, so that input characters appear at a suitable point within the window.

The "enter" key ('\n') is to terminate all input operations. The function HandleCharacter() may return false if the character ch corresponds to some

other (subclass specific) terminator; if this function returns true it means that the character was "successfully processed" (possibly by being discarded).

The virtual function `TerminateInput()` gets called when a terminating character has been read. Class `EditText` is an example of where this function can be useful; `EditText::TerminateInput()` adds a null character (`'\0'`) at the end of the input string.

In addition to checking the data entered, the `Validate()` function should deal with aspects like notifying a user of an incorrect value.

Class `EditWindow` can provide default definitions for most of these functions. These defaults make all characters acceptable (but nothing gets done with input characters), make all inputs "valid", do nothing special at input termination etc. Most subclasses will need to redefine several if not all these virtual functions.

By default, an `EditWindow` is expected to be a framed window with one line of content area (as illustrated in Figure 30.1 with its editable name and number fields).

The class declaration is:

<i>EditWindow declaration</i>	<code>class EditWindow : public Window {</code>
	<code>public:</code>
	<code> EditWindow(int id, int x, int y, int width,</code>
	<code> int height = 3, int mustValidate = 0);</code>
<i>Main GetInput() function</i>	<code> virtual char GetInput();</code>
	<code></code>
	<code> virtual int CanHandleInput() { return 1; }</code>
	<code> virtual int ContentChanged() { return 0; }</code>
	<code>protected:</code>
<i>Auxiliary functions for GetInput()</i>	<code> virtual void InitializeInput() {</code>
	<code> this->PrepareToDisplay();</code>
	<code> }</code>
	<code> virtual void SetCursorPosition() {</code>
	<code> this->SetPromptPos(fWidth-1, fHeight-1);</code>
	<code> }</code>
	<code> virtual int HandleCharacter(char ch) { return 1; }</code>
	<code> virtual void TerminateInput() { }</code>
	<code> virtual int Validate() { return 1; }</code>
	<code> int fEntry;</code>
	<code> int fMustValidate;</code>
	<code>};</code>

Other member functions The inherited function `Window::CanHandleInput()` is redefined. As specified by `EditWindow::CanHandleInput()`, all `EditWindow` objects can handle input.

A `ContentChanged()` function is generally useful. For example, an `EditNum` object has an "initial value" which it displays prior to an input operation; it can detect whether the input changes this initial value. If an `EditNum` object indicates that it has not been changed, there is no need to copy its value into a `Record` object. The `ContentChanged()` behaviour might as well be defined for class `EditWindow` although only it is only useful for some subclasses. It can be given a default definition stating "no change".

Functions like `ContentChanged()` and `InitializeInput()` have been defined in the class declaration. This is simply for convenience in presentation. They should either be defined separately in the header file as inline functions, or

defined normally in the implementation file. The definition of `GetInput()`, essentially the same as the pseudo code given, is included in Section 30.6.2.

The two data members added by class `EditWindow` are a flag to indicate whether input values should be validated (`fMustValidate`) and the variable `fEntry` (which is used in several subclasses to do things like record how many characters have been accepted).

EditWindow's extra data members

EditText

An `EditText` is an `EditWindow` that can accept input of a string. This string is held (in a 256 character buffer) within the `EditText` object. Some other object that needs text input can employ an `EditText`, require it to perform a `GetInput()` operation, and, when input is complete, can ask to read the `EditText` object's character buffer.

Naturally, class `EditText` must redefine a few of those virtual functions declared by class `EditWindow`. An `EditWindow` can simply discard characters that are input, but an `EditText` must save printable characters in its text buffer; consequently, `HandleCharacter()` must be redefined. An `EditWindow` positions the cursor in the bottom right corner of the window (an essentially arbitrary position), an `EditText` should locate the cursor at the left hand end of the text input field; so function `SetCursorPosition()` gets redefined.

Normally, there are no "validation" checks on text input, so the default "do nothing" functions like `EditWindow::Validate()` do not need to be redefined.

The declaration for class `EditText` is:

```
class EditText: public EditWindow {
public:
    EditText(int id, int x, int y, int width, char *label,
             short size = 256, int mustValidate = 0);
    void      SetVal(char* val, int redraw = 0);
    char      *GetVal() { return this->fBuf; }

    virtual int ContentChanged();
protected:
    virtual void InitializeInput();
    virtual void SetCursorPosition();
    virtual int  HandleCharacter(char ch);
    virtual void TerminateInput();

    void      ShowValue(int redraw);

    int      fLabelWidth;
    char      fBuf[256];
    int      fSize;
    int      fChanged;
};
```

Extra functions to set, and read string

Redefining auxiliary functions of GetInput()

Extra data members

Class `EditText` adds extra public member functions `SetVal()` and `GetVal()` that allow setting of the initial value, and reading of the updated value in its `fBuf`

text buffer. (There is an extra protected function `ShowValue()` that gets used in the implementation of `SetVal()`.)

Data members

Class `EditText` adds several data members. In addition to the text buffer, `fBuf`, there is an integer `fLabelWidth` that records how much of the window's width is taken up by a label. The `fSize` parameter has the same role as the size parameter in the `EditText` class used in Chapter 29. It is possible to specify a maximum size for the string. The input process will terminate when this number of characters has been entered. The class uses an integer flag, `fChanged`, that gets set if any input characters get stored in `fBuf` (so changing its previous value).

EditNum

An `EditNum` is an `EditWindow` that can deal with the input of a integer. This integer ends up being held within the `EditNum` object. Some other object that needs integer input can employ an `EditNum`, require it to perform a `GetInput()` operation, and, when input is complete, can ask the `EditNum` object for its current value.

The extensions for class `EditNum` are similar to those for class `EditText`. The class adds functions to get and set its integer. It provides effective implementations for `HandleCharacter()` and `SetCursorPosition()`. It has an extra (protected) `ShowValue()` function used by its `SetVal()` public function.

Validating numeric input

Class `EditNum()` redefines the `Validate()` function. The constructor for class `EditNum` requires minimum and maximum allowed values. If the "must validate" flag is set, any number entered should be checked against these limits.

The class declaration is:

Extra functions to set, and read integer

Redefining auxiliary functions of `GetInput()`

Data members

```
class EditNum: public EditWindow {
public:
    EditNum(int id, int x, int y, int width, char *label,
            long min, long max, int mustValidate = 0);
    void      SetVal(long, int redraw = 0);
    long      GetVal() { return this->fVal; }

    virtual int ContentChanged();

protected:
    virtual void InitializeInput();
    virtual void SetCursorPosition();
    virtual int  HandleCharacter(char ch);
    virtual void TerminateInput();
    virtual int  Validate();

    void      ShowValue(int redraw);

    int      fLabelWidth;
    long      fMin;
    long      fMax;
    long      fSetVal;
    long      fVal;
    int      fsign;
};
```

The data members include the minimum and maximum limits, the value of the `EditNum`, its "set value", and a label width. The `fsign` field is used during input to note the \pm sign of the number.

MenuWindow

A `MenuWindow` is a specialized input handling (`EditWindow`) type of window that allows selection from a menu. By default, it is a "full screen" window, (70x20).

The class declares two extra public member functions: `AddMenuItem()` and `PoseModally()`.

Function `AddMenuItem()` adds a menu item (a string and integer combination) to the `MenuWindow`. There is a limit on the number of items, function `AddMenuItem()` returns "false" if this limit is exceeded. The limit is defined by the constant `kMAXCHOICES` (a suitable limit value is 6). The menu strings are written into the window's "background image" at fixed positions; their identifier numbers are held in the array `fCmds`. ***AddMenuItem()***

Function `PoseModally()` does some minor setting up, calls `GetInput()` (using the standard version inherited from class `EditWindow`) and finally returns the selected command number. (A "modal" window is one that holds the user in a particular processing "mode". When a `MenuWindow` is "posed modally", the only thing that a user can do is switch back and forth among the different menu choices offered.) ***PoseModally() and "modal" windows***

Two of the auxiliary functions for `GetInput()` have to be redefined. Function `SetCursorPosition()` now displays a cursor marker (the characters `==>`). The position is determined by the "currently chosen" menu item (as identified by the data member `fChosen`).

Function `HandleCharacter()` is redefined. This version ignores all characters apart from "tabs". Input of a "tab" character changes the value of `fChosen` (which runs cyclically from `0...fChoices-1` where `fChoices` is the number of menu items added). Whenever the value of `fChosen` is changed, the cursor is repositioned. (The extra private member function `ClearCursorPosition()` is needed to clear the previous image of the cursor.)

<pre>class MenuWindow : public EditWindow { public: MenuWindow(int id, int x = 1, int y = 1, int width = 70, int height = 20); int AddMenuItem(const char *txt, int num); int PoseModally(); protected: virtual void SetCursorPosition(); virtual int HandleCharacter(char ch); void ClearCursorPosition(); int fCmds[kMAXCHOICES]; int fChoices; int fChosen;</pre>	<p><i>MenuWindow declaration</i></p> <p><i>Additional functionality</i></p> <p><i>Redefining auxiliary functions for GetInput()</i></p> <p><i>Data members</i></p>
---	---

```
};
```

The Dialogs: NumberDialog, TextDialog, and InputFileDialog

The "dialogs" (`NumberDialog`, `TextDialog` and its specialization `InputFileDialog`) are essentially little windows that pop up in the middle of the screen displaying a message and an editable subwindow. A `NumberDialog` works with an `EditNum` subwindow, while a `TextDialog` works with an `EditText` subwindow.

They have constructors (the constructor sets the message string and, in the case of the `NumberDialog` limits on the range permitted for input values) and a `PoseModally()` function. The `PoseModally()` function takes an input argument (a value to be displayed initially in the editable subwindow) and returns as a result (or as a result parameter) the input data received.

The `PoseModally()` function arranges for the editable subwindow to "get input" and performs other housekeeping, e.g. the `InputFileDialog` may try to open a file with the name entered by the user.

The class declarations are:

```
class NumberDialog : public EditWindow {
public:
    NumberDialog(const char* msg,
                 long min = LONG_MIN, long max = LONG_MAX);
    long PoseModally(long current);
protected:
    EditNum *fE;
};

class TextDialog : public EditWindow {
public:
    TextDialog(const char* msg);
    virtual void PoseModally(char *current,
                             char newdata[]);
protected:
    EditText *fE;
};

class InputFileDialog : public TextDialog {
public:
    InputFileDialog();
    void PoseModally(char *current, char newdata[],
                     int checked = 1);
};
```

RecordWindow

As noted earlier, class `RecordWindow` is a slightly more sophisticated version of the same idea as embodied in class `MenuWindow`. Rather than use something specific like a set of "menu items", a `RecordWindow` utilizes its list of "subwindows". "Tabbing" in a `MenuWindow` moves a marker from one item to another; "tabbing" in

a `RecordWindow` results in different editable subwindows being given the chance to "get input".

The class declaration is:

```
class RecordWindow : public EditWindow {
public:
    RecordWindow(Record *r);
    void    PoseModally();
protected:
    void        CountEditWindows();
    void        NextEditWindow();
    virtual void InitEditWindows();

    Record      *fRecord;
    int          fNumEdits;
    int          fCurrent;
    EditWindow   *fEWin;
};
```

*Auxiliary functions
for PoseModally*

Data members

The data members include the link to the associated `Record`, a count of the number of editable subwindows, an integer identifying the sequence number of the current editable subwindow and a pointer to that subwindow. (Pointers to subwindows are of course stored in the `fSubWindows` data member as declared in class `Window`.)

The constructor for class `RecordWindow` simply identifies it as something associated with a `Record`. The `RecordWindow` constructor makes it a full size (70x20) window.

The only additional public member function is `PoseModally()`. This is the function that arranges for each editable subwindow to have a turn at getting input. The function is defined as follows:

```
void RecordWindow::PoseModally()
{
    char ch;
    DisplayWindow();
    CountEditWindows();
    InitEditWindows();
    fCurrent = fNumEdits;

    do {

        NextEditWindow();
        fRecord->SetDisplayField(fEWin);

        ch = fEWin->GetInput();

        if(fEWin->ContentChanged()) {
            fRecord->ReadDisplayField(fEWin);
            fRecord->ConsistencyUpdate(fEWin);
        }
    } while(ch != '\n');
```

Initialization

*Loop until user ends
input with "enter"*

*Activate next edit
subwindow*

*Subwindow gets
input*

Update record

The initialization steps deal with things like getting the window displayed and determining the number of editable subwindows.

The main body of the `PoseModally()` function is its loop. This loop will continue execution until the user terminates all input with the "enter" key.

The code of the loop starts with calls to an auxiliary private member function that picks the "next" editable subwindow. The associated `Record` object is then told to (re)initialize the value in the subwindow. Once its contents have been reset to correspond to those in the appropriate member of the `Record`, the edit window is given its chance to "get input".

The editable item will return the character that stopped its "get input" loop. This might be the '\n' ("enter") character (in which case, the driver loop in `PoseModally()` can finish) or it might be a "tab" character (or any other character that the edit window can not handle, e.g. a '*' in an `EditNum`).

When an editable subwindow has finished processing input, it is asked whether its value has changed. If the value has been changed, the associated `Record` object is notified. A `Record` will have to read the new value and copy it into the corresponding data member. Sometimes, there is additional work to do (the "consistency update" call).

Example trace of interactions

Figure 30.9 illustrates a pattern of interactions between a `RecordWindow` and a `Record`. The example shown is for a `StudentRec` (as shown in Figure 30.1). It illustrates processes involved in changing the mark for assignment 1 from its default 0 to a user specified value.

When the loop in `RecordWindow::PoseModally()` starts, the `EditText` subwindow for the student's name will become the active subwindow. This results in the first interactions shown. The `RecordWindow` would ask the `Record` to set that display field; the `Record` would invoke `EditText::SetVal()` to set the current string. Then, the `EditText` object would be asked to `GetInput()`.

If the user immediately entered "tab", the `GetInput()` function would return leaving the `EditText` object unchanged.

After verifying that the subwindow's state was unchanged, the `RecordWindow` would arrange to move to the next subwindow. This would be the `EditNum` with the mark for the first assignment.

The `Record` would again be asked to set a subwindow's initial value. This would result in a call to `EditNum::SetVal()` to set the initial mark.

The `EditNum` subwindow would then have a chance to `GetInput()`. It would loop accepting digits (not shown in Figure 30.9) and would calculate the new value.

When this input step was finished, the `RecordWindow` could check whether the value had been changed and could get the `Record` to deal with the update.

30.6.2 Implementation Code

This section contains example code for the implementation of the window classes. Not all functions are given, but the code here should be sufficient to allow implementation. (The complete code can be obtained by ftp over the Internet as explained in the preface.)

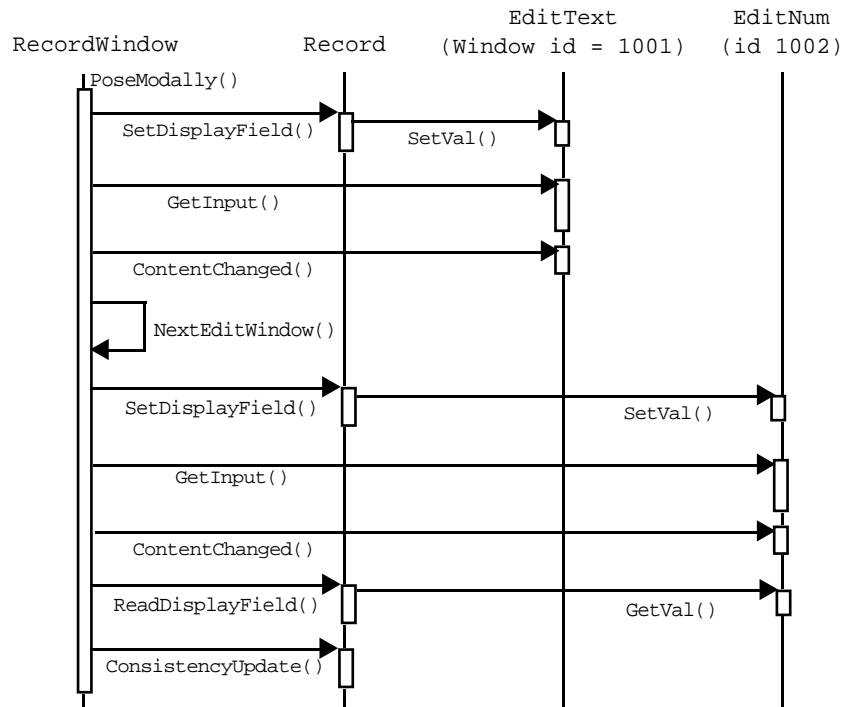


Figure 30.9 Example trace of specific object interactions while a RecordWindow is "posed modally".

Window

Most of the code is similar to that given in Chapter 29. Extra functions like ShowText() and ShowNumber() should be straightforward to code.

The DisplayWindow() function gets the contents of a window drawn, then arranges to get each subwindow displayed:

```

void Window::DisplayWindow()
{
    PrepareContent();
    PrepareToDisplay();
    ShowAll();
    int n = fSubWindows.Length();
    for(int i=1; i<=n; i++) {
        Window* sub = (Window*) fSubWindows.Nth(i);
        sub->DisplayWindow();
    }
}

```

The destructor will need to have a loop that removes subwindows from the fSubWindows collection and deletes them individually.

EditWindow

The constructor for class `EditWindow` passes most of the arguments on to the constructor of the `Window` base class. Its only other responsibility is to set the "validation" flag.

```

EditWindow::EditWindow(int id, int x, int y, int width,
                      int height, int mustValidate)
    : Window(id, x, y, width, height)
{
    fMustValidate = mustValidate;
}

```

The `GetInput()` function implements the algorithm given earlier. Characters are obtained by invoking the `GetChar()` function of the `WindowRep` object.

*Normal "get input"
behaviour*

```

char EditWindow::GetInput()
{
    int v;
    char ch;
    InitializeInput();
    do {
        SetCursorPosition();
        fEntry = 0;
        ch = WindowRep::Instance()->GetChar();
        while(ch != '\n') {
            if(!HandleCharacter(ch))
                break;
            ch = WindowRep::Instance()->GetChar();
        }
        TerminateInput();
        v = Validate();
    } while (fMustValidate && !v);
    return ch;
}

```

EditText

The `EditText` constructor is mainly concerned with sorting out the width of any label and getting the label copied into the "background" image array (via the call to `ShowText()`). The arguments passed to the `EditWindow` base class constructor fix the height of an `EditText` to three rows (content plus frame).

```

EditText::EditText(int id, int x, int y, int width,
                  char *label, short size, int mustValidate)
    : EditWindow(id, x, y, width, 3, mustValidate)
{
    fSize = size;
    fLabelWidth = 0;
    int s = (label == NULL) ? 2 : strlen(label)+2;
    width -= 6;
    fLabelWidth = (s < width) ? s : width;
}

```

```

    ShowText(label, 2, 2, fLabelWidth);
    fBuf[0] = '\0';
    fChanged = 0;
}

```

The `SetVal()` member function copies the given string into the `fBuf` array (avoiding any overwriting that would occur if the given string was too long):

```

void EditText::SetVal(char* val, int redraw)
{
    int n = strlen(val);
    if(n>254) n = 254;
    strncpy(fBuf, val, n);
    fBuf[n] = '\0';
    ShowValue(redraw);
    fChanged = 0;
}

```

The `ShowValue()` member function outputs the string to the right of any label already displayed in the `EditText`. The output area is first cleared out by filling it with spaces and then characters are copied from the `fBuf` array:

```

void EditText::ShowValue(int redraw)
{
    int left = fLabelWidth;
    int i, j;
    for(i=left; i<fWidth; i++)
        fCurrentImg[1][i-1] = ' ';
    for(i=left, j=0; i<fWidth; i++, j++) {
        char ch = fBuf[j];
        if(ch == '\0') break;
        fCurrentImg[1][i-1] = ch;
    }
    if(redraw)
        ShowContent();
}

```

***EditText::
ShowValue()***

If the string is long, only the leading characters get shown.

Function `EditText::HandleCharacter()` accepts all characters apart from control characters (e.g. "enter") and the "tab" character:

```

int EditText::HandleCharacter(char ch)
{
    if(iscntrl(ch) || (ch == kTABC))
        return 0;

    if(fEntry == 0) {
        ClearArea(fLabelWidth, 2, fWidth-1, 2, 0);
        Set(fLabelWidth, 2, ch);
        SetPromptPos(fLabelWidth+1, 2);
    }
    fBuf[fEntry] = ch;
    fChanged = 1;
    fEntry++;
}

```

***Handling input
characters
Return "fail" if
unacceptable
character***

***Initialization for first
character***

Store character


```

        if(fEntry == fSize) return 0;
        else return 1;
    }

```

Normally, an `EditText` will start by displaying the default text. This should be cleared when the first acceptable character is entered. Variable `fEntry` (set to zero in `InitializeInput()`) counts the number of characters entered. If its value is zero a `ClearArea()` operation is performed. Acceptable characters fill out the `fBuf` array (until the size limit is reached).

The remaining `EditText` member functions are all trivial:

```

int EditText::ContentChanged() { return fChanged; }

void EditText::InitializeInput()
{
    EditWindow::InitializeInput();
    fEntry = 0;
}

void EditText::SetCursorPosition()
{
    SetPromptPos(fLabelWidth, 2);
}

void EditText::TerminateInput()
{
    fBuf[fEntry] = '\0';
}

```

EditNum

Class `EditNum` is generally similar to class `EditText`. Again, its constructor is mainly concerned with sorting out the width of any label

```

EditNum::EditNum(int id, int x, int y, int width,
    char *label, long min, long max, int mustValidate)
    : EditWindow(id, x, y, width, 3, mustValidate)
{
    fMin = min;
    fMax = max;
    fSetVal = fVal = 0;
    int s = (label == NULL) ? 2 : strlen(label)+2;
    width -= 6;
    fLabelWidth = (s < width) ? s : width;
    ShowText(label, 2, 2, fLabelWidth);
}

```

It also sets the data members that record the allowed range of valid inputs and sets the "value" data member to zero.

The `SetVal()` member function restricts values to the permitted range:

```

void EditNum::SetVal(long val, int redraw)

```

```
{
    if(val > fMax) val = fMax;
    if(val < fMin) val = fMin;
    fSetVal = fVal = val;
    ShowValue(redraw);
}
```

Member `ShowValue()` has to convert the number to a string of digits and get these output (if the number is too large to be displayed it is replaced by '#' marks). There is a slight inconsistency in the implementation of `EditNum`. The initial value is shown "right justified" in the available field. When a new value is entered, it appears "left justified" in the field. (Getting the input to appear right justified is not really hard, its just long winded. As successive digits are entered, the display field has to be cleared and previous digits redrawn one place further left.)

```
void EditNum::ShowValue(int redraw)
{
    int left = fLabelWidth;
    int pos = fWidth - 1;
    long val = fVal;
    for(int i = left; i<= pos; i++)
        fCurrentImg[1][i-1] = ' ';
    if(val<0) val = -val;
    if(val == 0)
        fCurrentImg[1][pos-1] = '0';
    while(val > 0) {
        int d = val % 10;
        val = val / 10;
        char ch = d + '0';
        fCurrentImg[1][pos-1] = ch;
        pos--;
        if(pos <= left) break;
    }
    if(pos<=left)
        for(i=left; i<fWidth;i++)
            fCurrentImg[1][i-1] = '#';
    else
        if(fVal<0)
            fCurrentImg[1][pos-1] = '-';
    if(redraw)
        ShowContent();
}
```

The `HandleCharacter()` member function has to deal with a couple of special cases. An input value may be preceded by a `+` or `-` sign character; details of the sign have to be remembered. After the optional sign character, only digits are acceptable. The first digit entered should cause the display field to be cleared and then the digit should be shown (preceded if necessary by a minus sign).

```
int EditNum::HandleCharacter(char ch)
{
    // ignore leading plus sign(s)
    if((fEntry == 0) && (ch == '+'))
        return 1;
}
```

EditNum::
HandleCharacter()

Deal with initial sign

```

        if((fEntry == 0) && (ch == '-')) {
            fsign = -fsign;
            return 1;
        }
        if(!isdigit(ch))
            return 0;
        if(fEntry == 0) {
            ClearArea(fLabelWidth, 2, fWidth-1, 2, 0);
            fVal = 0;
            if(fsign<0) {
                Set(fLabelWidth, 2, '-');
                Set(fLabelWidth+1, 2, ch);
            }
            else Set(fLabelWidth, 2, ch);
        }
        fEntry++;
        fVal = fVal*10 + (ch - '0');
        return 1;
    }

```

Terminate input on non-digit

Clear entry field for first digit

Consume ordinary digits

As the digits are entered, they are used to calculate the numeric value which gets stored in fVal.

The \pm sign indication is held in fsign. This is initialized to "plus" in InitializeInput() and is used to set the sign of the final number in TerminateInput():

```

void EditNum::InitializeInput()
{
    EditWindow::InitializeInput();
    fsign = 1;
}

void EditNum::TerminateInput()
{
    fVal = fsign*fVal;
}

```

If the number entered is out of range, the Validate() function fills the entry field with a message and then, using the Beep() and Delay() functions of the WindowRep object, brings error message to the attention of the user:

```

int EditNum::Validate()
{
    if(!((fMin <= fVal) && (fVal <= fMax))) {
        ShowText("(out of range)", fLabelWidth, 2,
            fWidth-1,0,0);
        WindowRep::Instance()->Beep();
        WindowRep::Instance()->Delay(1);
        fVal = fSetVal;
        ClearArea(fLabelWidth, 2, fWidth-1, 2, 0);
        ShowValue(1);
        return 0;
    }
}

```

EditNum::Validate

Invalid entry

Failure return

```

        else return 1;
    }

```

The remaining member functions of class `EditNum` are simple:

```

int EditNum::ContentChanged() { return fVal != fSetVal; }

void EditNum::SetCursorPosition()
{
    SetPromptPos(fLabelWidth, 2);
}

```

MenuWindow

The constructor for class `MenuWindow` initializes the count of menu items to zero and adds a line of text at the bottom of the window:

```

MenuWindow::MenuWindow(int id, int x, int y, int width, int
height)
    : EditWindow(id, x, y, width, height)
{
    fChoices = 0;
    ShowText("(use 'option-space' to switch between
            'choices', 'enter' to select)",
            2, height-1, width-4);
}

```

Menu items are "added" by writing their text into the window background (the positions for successive menu items are predefined). The menu numbers get stored in the array `fCmds`.

```

int MenuWindow::AddMenuItem(const char *txt, int num)
{
    if(fChoices == kMAXCHOICES)
        return 0;
    int len = strlen(txt);
    fCmds[fChoices] = num;
    int x = 10;
    int y = 4 + 3*fChoices;
    ShowText(txt, x, y, fWidth-14);
    fChoices++;
    return 1;
}

```

***MenuWindow::
AddMenuItem()***

The `PoseModally()` member function gets the window displayed and then uses `GetInput()` (as inherited from class `EditWindow`) to allow the user to select a menu option:

```

int MenuWindow::PoseModally()
{
    DisplayWindow();
    fChosen = 0;
}

```

***MenuWindow::
PoseModally()***

```

        GetInput();
        return fCmds[fChosen];
    }

```

The "do nothing" function `EditWindow::HandleCharacter()` is redefined. Function `MenuWindow::HandleCharacter()` interprets "tabs" as commands changing the chosen item.

```

MenuWindow::      int MenuWindow::HandleCharacter(char ch)
HandleCharacter() {
    Ignore anything      if(ch != kTABC)
    except tabs          return 1;

    Clear cursor        ClearCursorPosition();

    Update chosen item   fChosen++;
                        if(fChosen==fChoices)
                        fChosen = 0;
                        SetCursorPosition();
                        return 1;
}

```

(The "tab" character is defined by the character constant `kTABC`. On some systems, e.g. Symantec's environment, actual tab characters from the keyboard get filtered out by the run-time routines and are never passed through to a running program. A substitute tab character has to be used. One possible substitute is "option-space" (`const char kTABC = 0xCA;`).

The auxiliary member functions `SetCursorPosition()` and `ClearCusorPostion()` deal with the display of the ==> cursor indicator:

```

void MenuWindow::SetCursorPosition()
{
    int x = 5;
    int y = 4 + 3*fChosen;
    ShowText("==>", x, y, 4,0,0);
    SetPromptPos(x,y);
}

void MenuWindow::ClearCursorPosition()
{
    int x = 5;
    int y = 4 + 3*fChosen;
    ShowText("  ", x, y, 4, 0, 0);
}

```

Dialogs

The basic dialogs, `NumberDialog` and `TextDialog`, are very similar in structure and are actually quite simple. The constructors create an `EditWindow` containing a

prompt string (e.g. "Enter record number"). This window also contains a subwindow, either an `EditNum` or an `EditText`.

```

NumberDialog::NumberDialog(const char *msg,
                           long min, long max)
    : EditWindow(kNO_ID, 15, 5, 35,10)
{
    fE = new EditNum(kNO_ID, 5, 5, 20, NULL, min, max, 1);
    ShowText(msg, 2, 2, 30);
    AddSubWindow(fE);
}

TextDialog::TextDialog(const char *msg)
    : EditWindow(kNO_ID, 15, 5, 35,10)
{
    fE = new EditText(kNO_ID, 5, 5, 20, NULL, 63, 1);
    ShowText(msg, 2, 2, 30);
    AddSubWindow(fE);
}

```

Dialog constructors

The `PoseModally()` functions get the window displayed, initialize the editable subwindow, arrange for it to handle the input and finally, return the value that was input.

```

long NumberDialog::PoseModally(long current)
{
    DisplayWindow();
    fE->SetVal(current, 1);
    fE->GetInput();
    return fE->GetVal();
}

void TextDialog::PoseModally(char *current, char newdata[])
{
    DisplayWindow();
    fE->SetVal(current, 1);
    fE->GetInput();
    strcpy(newdata, fE->GetVal());
}

```

PoseModally() functions

An `InputFileDialog` is simply a `TextDialog` whose `PoseModally()` function has been redefined to include an optional check on the existence of the file:

```

InputFileDialog::InputFileDialog() :
    TextDialog("Name of input file")
{
}

void InputFileDialog::PoseModally(char *current,
                                   char newdata[], int checked)
{
    DisplayWindow();
    for(;;) {

```

InputFileDialog

<i>Loop until valid file name given</i>	fE->SetVal(current, 1); fE->GetInput(); strcpy(newdata, fE->GetVal()); if(!checked)
<i>Try opening file</i>	return; ifstream in; in.open(newdata, ios::in ios::nocreate); int state = in.good(); in.close(); if(state)
<i>Warn of invalid file name</i>	return; WindowRep::Instance()->Beep(); fE->SetVal("File not found", 1); WindowRep::Instance()->Delay(1); }
	}

The Alert() function belongs with the dialogs. It displays an EditWindow with an error message. This window remains on the screen until dismissed by the user hitting the enter key.

<i>Global function Alert()</i>	void Alert(const char *msg) { EditWindow e(kNO_ID, 15, 5, 35, 10); e.DisplayWindow(); e.ShowText(msg, 2, 2, 30, 0, 0); e.ShowText("OK", 18, 6, 3, 0, 0); e.GetInput(); }
------------------------------------	---

RecordWindow

As far as the constructor is concerned, a RecordWindow is simply an EditWindow with an associated Record:

```
RecordWindow::RecordWindow(Record *r)
: EditWindow(0, 1, 1, 70, 20)
{
    fRecord = r;
}
```

<i>RecordWindow:: PoseModally()</i>	The main role of a RecordWindow is to be "posed modally". While displayed it arranges for its subwindows to "get input". (These subwindows get added using the inherited member function Window::AddSubWindow()). The code for RecordWindow::PoseModally() was given earlier.
---	---

The auxiliary functions CountEditWindows(), InitEditWindows(), and NextEditWindow() work with the subwindows list. The CountEditWindows() function runs through the list of subwindows asking each whether it "can handle input":

```
void RecordWindow::CountEditWindows()
{
```

```

        fNumEdits = 0;
        int nsub = fSubWindows.Length();
        for(int i = 1; i <= nsub; i++) {
            Window* w = (Window*) fSubWindows.Nth(i);
            if(w->CanHandleInput())
                fNumEdits++;
        }
    }
}

```

Function `InitEditWindows()`, called when the `RecordWindow` is getting displayed, arranges for the associated `Record` to load each window with the appropriate value (taken from some data member of the `Record`):

```

void RecordWindow::InitEditWindows()
{
    int nsub = fSubWindows.Length();
    for(int i = 1; i <= nsub; i++) {
        Window* w = (Window*) fSubWindows.Nth(i);
        if(w->CanHandleInput())
            fRecord->SetDisplayField((EditWindow*)w);
    }
}

```

Function `NextEditWindow()` updates the value of `fCurrent` (which identifies which editable subwindow is "current"). The appropriate subwindow must then be found (by counting through the `fSubWindows` list) and made the currently active window that will be given the chance to "get input".

```

void RecordWindow::NextEditWindow()
{
    if(fCurrent == fNumEdits)
        fCurrent = 1;
    else
        fCurrent++;
    int nsub = fSubWindows.Length();
    for(int i = 1, j = 0; i <= nsub; i++) {
        Window* w = (Window*) fSubWindows.Nth(i);
        if(w->CanHandleInput()) {
            j++;
            if(j == fCurrent) {
                fEWin = (EditWindow*) w;
                return;
            }
        }
    }
}

```

30.6.3 Using the concrete classes from a framework

The window class hierarchy has been presented in considerable detail.

Here, the detail was necessary. After all, you are going to have to get the framework code to work for the exercises. In addition, the structure and

implementation code illustrate many of the concepts underlying elaborate class hierarchies.

Apparent complexity

Class like `InputFileDialog` or `EditNum` are actually quite complex entities. They have more than thirty member functions and between ten and twenty data members. (There are about 25 member functions defined for class `Window`; other functions get added, and existing functions get redefined at the various levels in the hierarchy like `EditWindow`, `TextDialog` etc).

Simplicity of use

This apparent complexity is not reflected in their use. As far as usage is concerned, an `InputFileDialog` is something that can be asked to get the name of an input file. The code using such an object is straightforward, e.g.

```
void Document::OpenOld()
{
    InputFileDialog oold;
    oold.PoseModally("example",fFileName, fVerifyInput);
    OpenOldFile();
}
```

("Give me an InputFileDialog". "Hey, file dialog, do your stuff".)

This is typical. Concrete classes in the frameworks may have complex structures resulting from inheritance, but their use is simple.

In real frameworks, the class hierarchies are much more complex. For example, one of the frameworks has a class `TScrollBar` that handles scrolling of views displayed in windows. It is basically something that responds to mouse actions in the "thumb" or the "up/down" arrows and it changes a value that is meant to represent the origin that is to be used when drawing pictures. Now, a `TScrollBar` is a kind of `TCtrlMgr` which is a specialization of `TControl`. Class `TControl` is a subclass of `TView`, a `TView` is derived from `TCommandHandler`. A `TCommandHandler` is actually a specialization of class `TEventHandler`, and, naturally, class `TEventHandler` is derived from `TObject`.

By the time you are six or seven levels deep in a class hierarchy, you have something fairly complex. A `TScrollBar` will have a fair number of member functions defined (about 300 actually) because it can do anything a `TObject` can do, and all the extra things that a `TEventHandler` added, plus the behaviours of a `TCommandHandler`, while `TView` and others added several more abilities.

Usually, you aren't interested. When using a `TScrollBar`, all you care is that you can create one, tell it to do its thing, and that you can use a member function `GetVal()` to get the current origin value when you need it.

The reference manuals for the class libraries will generally document just the unique capabilities of the different concrete classes. So something like an `InputFileDialog` will be described simply in terms of its constructor and `PoseModally()` function.

Of course sometimes you do make some use of functions that are inherited from base classes. The `RecordWindow` class is an example. This could be described primarily in terms of its constructor, and `PoseModally()` functions. However, code using a `RecordWindow` will also use the `AddSubWindows()` member function.

30.7 ORGANIZATIONAL DETAILS

When working with a framework, your primary concerns are getting to understand the conceptual application structure that is modelled in the framework, and the role of the different classes.

However, you must also learn how to build a program using the framework. This is a fairly complex process, though the integrated development environments may succeed in hiding most of the complexities.

There are a couple of sources of difficulties. Firstly, you have all the "header" files with the class declarations. When writing code that uses framework classes, you have to #include the appropriate headers. Secondly, there is the problem of linking the code. If each of the classes is in a separate implementation file, you will have to link your compiled code with a compiled version of each of the files that contains framework code that you rely on.

*Problems with
"header" files and
linking of compiled
modules*

Headers

One way of dealing with the problem of header files is in effect to #include them all. This is normally done by having a header file, e.g. "ClassLib.h", whose contents consist of a long sequence of #includes:

*"The enormous
header file"*

```
#include "Cmdhdl.h"
#include "Application.h"
#include "Document.h"
#include "WindowRep.h"
...
#include "Dialog.h"
```

This has the advantage that you don't have to bother to work out which header files need to be included when compiling any specific implementation (.cp) file.

The disadvantage is that the compiler has to read all those headers whenever a piece of code is compiled. Firstly, this makes the compilation process slow. With a full size framework, there might be fifty or more header files; opening and reading the contents of fifty files takes time. Secondly, the compiler has to record the information from those files in its "symbol tables". A framework library may have one hundred or more classes; these classes can have anything from ten to three hundred member functions. Recording this information takes a lot of space. The compiler will need many megabytes of storage for its symbol tables (and this had better be real memory, not virtual memory on disk, because otherwise the process becomes too slow).

*Slow compilations,
large memory usage*

There are other problems related to the headers, problems that have to be sorted out by the authors of the framework. For example, there are dependencies between different classes and these have to be taken into account when arranging declarations in a header file. In a framework, these dependencies generally appear as the occurrence of data members (or function arguments) that are pointers to instances of other classes, e.g:

*Interdependencies
among class
declarations*

```

class Record {
    ...
    RecordWindow *fWin;
    ...
};

class RecordWindow {
    ...
    Record *fRec;
    ...
};

```

If the compiler encounters class `Record` before it learns about class `RecordWindow` it will complain that it does not understand the declaration `RecordWindow *fWin`. If the compiler encounters class `RecordWindow` before it learns about class `Record` it will complain that it doesn't understand the declaration `Record *fRec`.

One solution to this problem is to have dummy class declarations naming the classes before any instances get mentioned:

```

class CommandHandler;
class Application;
class Document;
class Window;
class Record;
class RecordWindow;
...
// Now get first real class declaration

class CommandHandler {
    ...
};

```

Such a list might appear at the start of the composite "ClassLib.h" file.

An alternative mechanism is to include the keyword `class` in all the member and argument declarations:

```

class RecordWindow {
public:
    RecordWindow(class Record *r);
    ...
protected:
    ...
    class Record *fRec;
    ...
};

```

Linking

The linking problem is that a particular program that uses the framework must have the code for all necessary framework classes linked to its own code.

One way of dealing with this is, once again, to use a single giant file. This file contains the compiled code for all framework classes. The "linker" has to scan this file to find the code that is needed.

Such a file does tend to be very large (several megabytes) and the linkage process may be quite slow. The linker may have to make multiple passes through the file. For example, the linker might see that a program needs an `InputFileDialog`; so it goes to the library file and finds the code for this class. Then it finds that an `InputFileDialog` needs the code for `TextDialog`; once again, the linker has to go and read through the file to find this class. In unfortunate cases, it might be necessary for a linker to read the file three or four times (though there are alternative solutions).

If the compiled code for the various library classes is held in separate files, the linker will have to be given a list with the names of all these files.

Currently the Symantec system has a particularly clumsy approach to dealing with the framework library. In effect, it copies all the source files of the framework into each "project" that is being built using that framework. Consequently, when you start, you discover that you already have 150 files in your program. This leads to lengthy compilation steps (at least for the first compilation) as well as wastage of disk space. (It also makes it easier for programmers to change the code of the framework; changing the framework is not a wise idea.)

RecordFile Framework

Figure 30.10 illustrates the module structure for a program built using the RecordFile framework. Although this example is more complex than most programs that you would currently work with, it is typical of real programs. Irrespective of whether you are using OO design, object based design, or functional decomposition, you will end up with a program composed from code in many implementation files. You will have header files declaring classes, structures and functions. There will be interdependencies. One of your tasks as a developer is sorting out these dependencies.

The figure shows the various files and the `#include` relationships between files. Sometimes a header file is `#included` within another header; sometimes a header file is `#included` by an implementation (.cp) file.

When you have relationships like class derivation, e.g. `class MyApp : public Application`, the header file declaring a derived class like `MyApp` (`My.h`) has to `#include` the header defining the base class `Application` (`Application.h`).

If the relationship is simply a "uses" one, e.g. an instance of class `RecordWin` uses an instance of class `Record`, then the `#include` of class `Record` can be placed in the `RecordWin.cp` file (though you will need a declaration like `class Record` within the `RecordWin.h` file in order to declare `Record*` data members).

The files used are as follows:

- `commands.h`
Declares constants for "command numbers" (e.g. `CNEW` etc). Used by all `CommandHandler` classes so `#included` in `CmdHdl.h`

- Document.h and Application.h
Declare the corresponding "partially implemented abstract classes".
- BTDoc.h
Declares class BTDoc, #including the Document.h class (to get a definition of the base class Document), and BTree.h to get details of the storage structure used.
- Record.h and RecordWin.h
Declare the Record and RecordWin classes. Class RecordWin is separate from the other Window classes. The main group of Window classes could be used in other programs; RecordWin is special for the type of programs considered in this chapter.
- My.h
Declares the program specific "MyApp", "MyDocument", and "MyRec" classes.
- The ".cp" implementation files
The code with definitions for the classes, additional #includes as needed.

The main program, in file main.cp, will simply declare an instance of the "MyApp" class and tell it to "run".

30.8 THE "STUDENTMARKS" EXAMPLE PROGRAM

We might as well start with the main program. It has the standard form:

```
int main()
{
    StudentMarkApp a;
    a.Run();
    return 0;
}
```

The three classes that have to be defined are StudentMarkApp, StudentMarkDoc, and StudentRec. The "application" and "document" classes are both simple:

```
class StudentMarkApp : public Application {
protected:
    virtual      Document* DoMakeDocument();
};

class StudentMarkDoc : public BTDoc {
protected:
    virtual Record      *DoMakeRecord(long recnum);
    virtual Record      *MakeEmptyRecord();
};
```

*Application and
Document classes*

The implementation for their member functions is as expected:

**Implementation of
Application and
Document classes**

```

Document *StudentMarkApp ::DoMakeDocument()
{
    return new StudentMarkDoc;
}

Record *StudentMarkDoc::DoMakeRecord(long recnum)
{
    return new StudentRec (recnum);
}

Record *StudentMarkDoc::MakeEmptyRecord()
{
    return new StudentRec (0);
}

```

Record class

The specialized subclass of class `Record` does have some substance, but it is all quite simple:

**Redefine
KeyedStorableItem
functions**

```

class StudentRec : public Record {
public:
    StudentRec (long recnum);

    virtual void    WriteTo(fstream& out) const;
    virtual void    ReadFrom(fstream& in);
    virtual long    DiskSize(void) const ;

```

**Redefine Record
functions**

```

    virtual void    SetDisplayField(EditWindow *e);
    virtual void    ReadDisplayField(EditWindow *e);
protected:
    virtual void    ConsistencyUpdate(EditWindow *e);
    virtual void    AddFieldsToWindow();

```

**Declare unique data
members**

```

    char    fStudentName[64];
    long    fMark1;
    long    fMark2;
    long    fMark3;
    long    fMark4;
    long    fMidSession;
    long    fFinalExam;

    NumberItem    *fN;
    long    fTotal;
};

```

The data required include a student name and marks for various assignments and examinations. There is also a link to a `NumberItem` that will be used to display the total mark for the student.

The constructor does whatever an ordinary `Record` does to initialize itself, then sets all its own data members:

```

StudentRec::StudentRec(long recnum) : Record(recnum)
{
    fMark1 = fMark2 = fMark3 = fMark4 =
        fMidSession = fFinalExam = 0;
}

```

```

        strcpy(fStudentName, "Nameless");
    }

```

The function `AddFieldsToWindow()` populates the `RecordWindow` with the necessary `EditText` and `EditNum` editable subwindows:

```

void StudentRec::AddFieldsToWindow()
{
    Record::AddFieldsToWindow();

    EditText *et = new EditText(1001, 5, 4, 60,
        "Student Name ");
    fRW->AddSubWindow(et);

    EditNum *en = new EditNum(1002, 5, 8, 30,
        "Assignment 1 (5) ", 0, 5,1);
    fRW->AddSubWindow(en);
    en = new EditNum(1003, 5, 10, 30,
        "Assignment 2 (10) ", 0, 10,1);
    fRW->AddSubWindow(en);
    ...
    en = new EditNum(1007, 40, 10, 30, "Examination (50) ",
        0, 60,1);
    fRW->AddSubWindow(en);

    fTotal = fMark1 + fMark2 + fMark3 + fMark4
        + fMidSession + fFinalExam;
    fN = new NumberItem(2000, 40, 14,30, "Total ", fTotal);
    fRW->AddSubWindow(fN);
}

```

Building the display structure

Adding an EditText

Adding some EditNum subwindows

Adding a NumberItem

The call to the base class function, `Record::AddFieldsToWindow()`, gets the `NumberItem` for the record number added to the `RecordWindow`. An extra `NumberItem` subwindow is added to hold the total.

Functions `SetDisplayField()` and `ReadDisplayField()` transfer data to/from the `EditText` (for the name) and `EditNum` windows:

```

void StudentRec ::SetDisplayField(EditWindow *e)
{
    long id = e->Id();
    switch(id) {
case 1001:
        ((EditText*)e)->SetVal(fStudentName, 1);
        break;
case 1002:
        ((EditNum*)e)->SetVal(fMark1,1);
        break;
case 1003:
        ((EditNum*)e)->SetVal(fMark2,1);
        break;
case 1004:
        ((EditNum*)e)->SetVal(fMark3,1);
        break;
case 1005:
        ((EditNum*)e)->SetVal(fMark4,1);
    }
}

```

Data exchange with editable windows

Setting an EditText

Setting EditNum


```

        break;
    case 1006:
        ((EditNum*)e)->SetVal(fMidSession,1);
        break;
    case 1007:
        ((EditNum*)e)->SetVal(fFinalExam,1);
        break;
    }
}

void StudentRec::ReadDisplayField(EditWindow *e)
{
    long id = e->Id();
    switch(id) {
Reading an EditText
    case 1001:
        char* ptr = ((EditText*)e)->GetVal();
        strcpy(fStudentName, ptr);
        break;
Reading an EditNum
    case 1002:
        fMark1 = ((EditNum*)e)->GetVal();
        break;
        ...
        ...
    case 1007:
        fFinalExam = ((EditNum*)e)->GetVal();
        break;
    }
}

```

The ReadFrom() and WriteTo() functions involve a series of low level read (write) operations that transfer the data for the individual data members of the record:

```

Reading from file
void StudentRec::ReadFrom(fstream& in)
{
    in.read((char*)&fRecNum, sizeof(fRecNum));
    in.read((char*)&fStudentName, sizeof(fStudentName));
    in.read((char*)&fMark1, sizeof(fMark1));
    in.read((char*)&fMark2, sizeof(fMark2));
    in.read((char*)&fMark3, sizeof(fMark3));
    in.read((char*)&fMark4, sizeof(fMark4));
    in.read((char*)&fMidSession, sizeof(fMidSession));
    in.read((char*)&fFinalExam, sizeof(fMidSession));
}

Writing to file
void StudentRec::WriteTo(fstream& out) const
{
    out.write((char*)&fRecNum, sizeof(fRecNum));
    out.write((char*)&fStudentName, sizeof(fStudentName));
    out.write((char*)&fMark1, sizeof(fMark1));
    ...
    out.write((char*)&fFinalExam, sizeof(fMidSession));
}

```

(The total does not need to be saved, it can be recalculated.)

The DiskSize() function computes the size of a record as held on disk:

```
long StudentRec::DiskSize(void) const
{
    return sizeof(fRecNum) + sizeof(fStudentName)
        + 6 * sizeof(long);
}
```

Function ConsistencyUpdate() is called after an editable subwindow gets changed. In this example, changes will usually necessitate the recalculation and redisplay of the total mark:

```
void StudentRec::ConsistencyUpdate(EditWindow *e)
{
    fTotal = fMark1 + fMark2 + fMark3 + fMark4
        + fMidSession + fFinalExam;
    fN->SetVal(fTotal, 1);
}
```

EXERCISES

1. Implement all the code of the framework and the StudentMark application.
2. Create new subclasses of class Collection and class Document so that an AVL tree can be used for storage of data records.
3. Implement the Loans program using your AVLDoc and AVLCollection classes.

The records in the files used by the Loans program are to have the following data fields:

- customer name (64 characters)
- phone number (long integer)
- total payments this year (long integer)
- an array with five entries giving film names (64 characters) and weekly charge for movies currently on loan.

The program is to use a record display similar to that illustrated in Figure 30.3.

4. There is a memory leak. Find it. Fix it correctly.
(Hint, the leak will occur with a disk based collection. It is in one of the Document functions. The "obvious" fix is wrong because it would disrupt memory based collection structures.)

31 Frameworks for Understanding

It is unlikely that you will learn how to use a "framework class library" in any of your computer science courses at university. Frameworks don't possess quite the social cachet of "normal forms for relational databases", "NP complexity analysis", "LR grammars", "formal derivation of algorithms", "Z specification" and the rest. You might encounter "Design Patterns" (which relate to frameworks) in an advanced undergraduate or graduate level course; but, on the whole, most universities pay little regard to the framework-based development methodologies.

However, if you intend to develop any substantial program on your personal computer, you should plan to use the framework class library provided with your development environment.

As illustrated by the simple "RecordFile" framework presented in the last chapter, a framework class library will embody a model for a particular kind of program. Some of the classes in the framework are concrete classes that you can use as "off the shelf" components (things like the `MenuWindow` and `NumberDialog` classes in the `RecordFile` framework). More significant are the partially implemented abstract classes. These must be extended, through inheritance, to obtain the classes that are needed in an actual program.

Of course, the framework-provided member functions already define many standard patterns of interactions among instances of these classes. Thus, in the `RecordFile` framework, most of the code needed to build a working program was already provided by the `Application`, `Document`, and `Record` classes along with helpers like `RecordWindow`.

If you can exploit a framework, you can avoid the need to reimplement all the standard behaviours that its code embodies.

The "typical framework"

The commonly encountered frameworks provide a model for what might be termed "the classic Macintosh application". With some earlier prototypes at Xerox's research center, this form of program has been around since 1984 on the Macintosh (and, from rather later, on Intel-Windows machines). It is the model embodied in

*The "classic
Macintosh
application"*

***Application,
Document, View, and
Window***

your development environment, in your word processor program, your spreadsheet, your graphics editor, and a host of other packages.

You have an "Application". It creates "Documents" (which can save their data to disk files). The contents of documents are displayed in "Views" inside "Windows". (On the Macintosh, separate documents have completely separate windows. On a Microsoft Windows system, an "Application" that supports a "multiple document interface" has a main window; the individual documents are associated with subwindows grouped within this main window). The same data may appear in multiple views (e.g. a spreadsheet can display a table of numbers in one view while another view shows a pie-chart representing the same values).

***"Pick and object, give
it a command"***

The manuals for these programs typically describe their operation in terms of "picking an object, and giving it a command". The manual writers are thinking of "objects" that are visual elements displayed in the views. In most cases, these will represent actual objects as created and manipulated by the program.

"Picking an object" generally involves some mouse actions (clicking in a box, or dragging an outline etc). The bit about "giving a command" generally refers to use of a menu, or of some form of tool bar or tool palette. For instance, in your word processor you can "pick a paragraph" and then give a command like "set 'style' to 'margin note'"; while in a drawing program you can pick a line and select "Arrows... at start". Such commands change attributes of existing objects. New objects can be created in a variety of ways, e.g. drawing, typing, or menu commands like "paste".

***"Network" data
structures***

Using the editing tools associated with a program, you build an elaborate "network" data structure. Most of the programs will represent their data in memory as a network of separate "objects" interlinked via pointers.

If the program is a word processor, these objects might include "text paragraphs", "pictures", "tables", "style records", "format run descriptors" (these associate styles with groups of characters in paragraphs, they link to their paragraphs via pointers), and so forth. The sequence of "paragraphs" and "pictures" forming a document might be defined by a list.

A draw program will have as basic elements things like lines, boxes, ovals; the basic components of a picture are instances of these classes (each instance having its own attributes like colour, size, position). Individual elements can be grouped; such groups might be represented by lists. Usually, a draw program allows multiple groups, so its primary data structure might be a list of lists of elements. The ordering in the top level list might correspond to the front to back ordering in which items are drawn.

A spreadsheet might use an array of cells whose contents are pointers to data objects like text strings, numbers, and "formulae objects". There would be other data structures associated with the array that help identify relationships amongst cells so that if a number in a cell gets changed, the dependent formulae are recalculated.

This network data structure (or a printout of a visual representation) might be all that the program produces (e.g. word processor and draw programs). In other cases, like the spreadsheet, you want to perform calculations once the data are entered.

Example frameworks

The frameworks that you are most likely to encounter are, for the Macintosh:

- Symantec's Think Class Library (TCL) Version 2
- Apple's MacApp Version 3.5

and for Intel-Windows:

- Borland's Object Windows Library (OWL) Version 2
- Microsoft Foundation Classes (MFC) Version 1.5

Several other companies have competing products for the Intel-Windows environment.

There are substantial similarities among these frameworks because they all attempt to model the same type of interactive program and all have been influenced, to varying degrees, by the earlier experimental MacApp 1 and MacApp 2 libraries. (The Borland framework includes a slightly simpler Program-Window model in addition to the "standard" Application-Document-Window-View model. If you are using the Borland framework, you will probably find it worthwhile using the simpler Program-Window model for the first one or two programs that you try to implement.)

Similar underlying model

A similar class library exists for Unix. This ET++ class library is available free via ftp on the Internet (on Netscape, try <ftp://ftp.ubilab.ubs.ch/pub/ET++/>). It might be possible to adapt ET++ for use on a Linux environment.

There are "problems" with the C++ code in all these libraries. The Borland library probably makes the best use of the current C++ language. The ET++ library was written several years ago, before C++ supported either multiple inheritance or templates and, in some respects, its coding styles are dated. The Symantec compiler does not yet support recent language features like exceptions (the library fakes exception handling with the aid of various "macros"). The dialect of C++ used for MacApp is slightly non-standard (it had to be changed so that the library could support use from dialects of Pascal and Modula2 as well as C++). The MacApp system is also relatively expensive compared with the other products and its development environment, although having some advantages for large scale projects, is not nearly as convenient to use as the Integrated Development Environments from Symantec, Borland, or Microsoft.

Some C++ limitations

The early class libraries (MacApp and ET++), and also MFC, have essentially all their classes organized in a tree structured hierarchy. This style was influenced by earlier work on class libraries for Smalltalk. There are some disadvantages with this arrangement. "Useful" behaviours tend to migrate up to the base class of the hierarchy. The base "TObject" class acquires abilities so that an instance can: transfer itself to/from disk, duplicate itself, notify other dependent objects any time it gets changed, and so forth. All classes in the hierarchy are then expected to implement these behaviours (despite the fact that in most programs `Window` objects never get asked to duplicate themselves or transfer themselves to disk). This is one

Library structures

the factors that contribute to complexity of the frameworks (like the three hundred member functions for a scroll bar class).

The structure of the "RecordFile" framework, a "forest" of separate trees, is similar to that of the more modern frameworks. These more modern frameworks can still provide abstractions representing abilities like persistence (the ability of an object to transfer itself to/from a disk file). In modern C++, this can be done by defining a pure abstract (or maybe partially implemented abstract) base class, e.g. class `Storable`. Class `Storable` defines the interface for any object that can save itself (functions like `DiskSize()`, `ReadFrom()`, `WriteTo()`). Persistent behaviours can be acquired by a class using multiple inheritance to fold "storability" in with the class's other ancestry.

In the older frameworks, the "collection classes" can only be used for objects that are instances of concrete classes derived from the base "TObject" class. More modern frameworks use template classes for their collections.

The frameworks are evolving. Apart from ET++, each of the named frameworks is commercially supported and new versions are likely to appear. This evolutionary process tends to parallelism rather than divergence. Clever features added in the release of one framework generally turn up in the following releases of the competitors.

Tutorial examples

It will take you at least a year to fully master the framework library for your development environment. But you should be able to get their tutorial examples to run immediately and be able to implement reasonably interesting programs within two or three weeks of framework usage. ET++ provides the source code for some examples but no supporting explanations or documentation. The commercial products all include documented tutorial examples in their manuals. These tutorials build a very basic program and then add more sophisticated features. The tutorial for Borland OWL is probably the best of those currently available.

There is no point in reproducing three or four different tutorials. Instead the rest of this chapter looks at some general aspects common to all the frameworks.

Chapter topics

Resources, code generators, etc

The following sections touch on topics like "Resources", "Framework Macros", the auxiliary code-generating tools (the so called "Wizards" and "Experts"), and "Graphics" programming. There are no real complexities. But together these features introduce a nearly impenetrable fog of terminology and new concepts that act as a barrier to those just starting to look at a framework library.

Persistence

There is then a brief section on support for "persistent objects". This is really a more advanced feature. Support for persistence becomes important when your program needs to save either complex "network" data structures involving many separate objects interlinked via pointer data members, or collections that are going to include instances of different classes (heterogeneous collections). If your framework allows your program to open ordinary `fstream` type input/output files (like the files used in all previous examples), you should continue to use these so long as your data are simple. You should only switch to using the more advanced

forms of file usage when either forced by the framework or when you start needing to save more complex forms of data.

The final sections look at the basic "event handling" structure common to these frameworks, and other common patterns of behaviour. *Events*

31.1 "RESOURCES" AND "MACROS"

Resources

Most of the frameworks use "Resources". "Resources" are simply predefined, pre-initialized data structures. *"Resource" data structures*

Resources are defined (using "Resource Editors") externally to any program. A file containing several "resources" can, in effect, be include in the list of compiled files that get linked to form a program. Along with the resources themselves, a resource file will contain some kind of index that identifies the various data structures by type and by an identifier name (or unique identifier number).

When a program needs to use one these predefined data structures, it makes a call to the "Resource Manager" (a part of the MacOS and Windows operating systems). The Resource Manager will usually make a copy, in the heap, of the "Resource" data structure and return a pointer to this copy. The program can then do whatever it wants with the copy.

Resources originated with the Macintosh OS when that OS and application programs were being written in a dialect of Pascal. Pascal does not support direct initialization of the elements of an array or a record structure. It requires the use of functions with separate statements that explicitly initialise each of the various individual data elements. Such functions are very clumsy. This clumsiness motivated the use of the separate "resource" data structures. These were used for things like arrays of strings (e.g. the words that appear as menu options, or the error messages that get displayed in alert box), and simple records like a "rectangle" record that defines the size and position of a program's main window.

It was soon found that resources were convenient in other ways. If the error messages were held in an external data structure, rewording of a message to make it clearer (or translation into another language) was relatively easy. The resource editor program would be used to change the resource. There was no need to change program source text and laboriously recompile and relink the code.

New resource types were defined. Resources were used for sound effects, pictures, fonts and all sorts of other data elements needed by programs. Resource bitmap pictures could be used to define the iconic images that would represent programs and data files. An "alert" resource would define the size and content of a window that could be used to display an error message. A "dialog" resource could define the form of a window, together with the text and number fields used for input of data. *All data types – sound, graphics, text, windows*

The very first resources were defined as text. The contents of such text files would be a series of things very like definitions of C structs. The files were "compiled" (by a simplified C compiler) to produce the actual resource files.

Resource editors

Subsequently, more sophisticated resource editors have been created. These resource editors allow a developer to choose the form of resource that is to be built and then to enter the data. Textual data can be typed in; visual elements are entered using "graphics editor" components within the resource editor itself. These graphics editors are modelled on standard "draw" programs and offer palettes of tools. An "icon editor" component will have tools for drawing lines, arcs, and boxes; a "window editor" component will have tools that can add "buttons", "text strings", "check boxes" and so forth. Depending on the environment that you are working with, you may have a single resource editor program that has a range of associated editor components, one for each type of resource; alternatively, you may have to use more than one resource editor program.

Macintosh resource editors generally produce resources in "compiled form" (i.e. the final resource file, with binary data representing the resources, is generated directly). The resource editors for the Windows development environments create a text file representation that gets compiled during the program "build" process. You can view the Windows resources that you define in the text format although you will mostly use the graphically oriented components of the resource editor. There are auxiliary programs for the Macintosh development environments that convert compiled resources into a textual form; it would be unusual for you to need to use these.

*"Shared" resources
for Windows OS*

The Windows development environments allow resources to be "shared". A resource file that you create for a new project can in effect "#include" other resource files. This has the advantage of saving a little disk space in that commonly used resources (e.g. the icons that go in a toolbar to represent commands like "File/Open", or "Print") need only exist in some master resource file rather than being duplicated in the resource files for each individual project. However there are disadvantages, particularly for beginners. Frequently, a beginner has the opinion that any resource associated with their project can be changed, or even deleted. Deleting the master copy of something like the main "file dialog" resource causes all sorts of problems. If you are working in a Windows development environment, check the text form of a resource file to make sure the resource is defined in the file rather than #included. The resource editor will let you duplicate a resource so that you can always get a separate, and therefore modifiable copy of a standard resource.

The individual resources of a given type have identifier numbers (e.g. MENU 1001, MENU 1002, etc). These need to be used in both the resource file and the program text where the resources are used. If the numbers used to identify particular resources are inconsistent, the program won't work. In the Macintosh environments, you just have to keep the numbers consistent for yourself.

In the Windows environments, you can use a header file that contains a series of #define constants (e.g. #define MYFILEMENU 1001). This header file can be #included in both the (source text) resource file and in program (.cp) files. This reduces the chance of inconsistent usage of resource numbers. Instead of a separate header file, you can put such declarations at the start of the resource (.rc) file. Any resource #includes and actual definitions of new resources can then be bracketed by C++ compiler directive (e.g. #ifdef RESOURCECOMPILER ... #endif) that hide these data from the C++ compiler. The resource (.rc) file may then get

#included in the .cp source text files. You will see both styles (separate header and direct #include of a resource .rc file) in your framework's examples.

Resources were invented long before development switched to C++ and OO programming approaches. Resources are just simple data structures. The frameworks will define classes that correspond to many of the resource types. For example, you might have a program that needs bitmap iconic pictures (or multicoloured pixmap pictures); for these you would use instances of some class "BitMapPicture" provided by the framework. The framework's "BitMapPicture" class will have a member function, (e.g. `void BitMapPicture::GetResource(int idnum)`) that allows a `BitMapPicture` object to load a `BITMAP` resource from a file and use the resource data to initialize its own data fields. On the whole, you won't encounter problems in pairing up resource data structures and instances of framework classes.

Resources and classes

However, there can be problems with some forms of "window" resources (these problems are more likely to occur with the Macintosh development environments as the Windows environments handle things slightly differently). You may want to define a dialog window that contains "static text" strings for prompts, "edit text" strings for text input fields, and "edit number" components for entering numeric data. Your resource editor has no problem with this; it can let you design your dialog by adding `EditText` and `EditNum` components selected from a tools palette. The data structure that it builds encodes this information in a form like an instruction list: "create an `EditNum` and place it here in the window, create an `EditText` and place it here".

The corresponding program would use an instance of some framework class "DialogWindow". Class `DialogWindow` might have a constructor that simply creates an empty window. Its `GetResource()` function would be able to load the dialog window resource and interpret the "instruction list" so populating the new window with the required subwindows. It is at this point that there may be problems.

The basic problem is that the program cannot create an instance of class `EditText` if the code for this class is not linked. Now the only reference to class `EditText` in the entire program may be the implicit reference in something like a resource data structure defining an "InputFileDialog". The linker may not have seen any reason to add the `EditText` code to the program.

You can actually encounter run-time errors where an alert box pops up with a message like "Missing component, consult developer". These occur when the program is trying to interpret a resource data structure defining a window that contains a subwindow of a type about which the program knows nothing.

Missing component, consult developer!

Your framework will have a mechanism for dealing with this possible problem. The exact mechanism varies a lot between frameworks. Generally, you have to have something like an "initialization function" that "names" all the special window related classes that get used by the program. This initialization function might have a series of (macro) calls like `ForceReference(EditText); ForceReference(EditNum)` (which ensure that the compiled program would have the code necessary to deal with `EditText` and `EditNum` subwindows).

You may find that there are different kinds of resource that seem to do more or less the same thing. For example on the Macintosh you will find a "view" resource

"Outdated" resource types

type and a "DLOG" (dialog) and its related "DITL" (dialog item list) resource types. They may seem to allow you do the same thing – build a dialog window with subwindows. They do differ. One will be the kind of resource that you use with your framework code, the other (in this case the DLOG and DITL resources) will be an outdated version that existed for earlier Pascal based development systems. Check the examples of resource usage that come with the tutorials in your IDE manuals and use the resources illustrated there.

Macros

A "macro" is a kind of textual template that can be expanded out to give possibly quite a large number of lines of text. Macros may have "arguments"; these are substituted into the expansion text at appropriate points. Macros can appear in the source text of programs. They are basically there to save the programmer from having to type out standard code again and again.

For example, you could define the following macro:

Macro definition `#define out(o, x) o.write((char*)&x, sizeof(x))`

This defines a macro named "out". The out macro takes two arguments, which it represents by 'o' and 'x'. Its expansion says that a call to "out" should be replaced by code involving a call to the `write()` function, involving a type cast, an address operator and the `sizeof()` operator.

Macro call This would allow you to use "macro calls" like the following:

```
out(ofile, fMark1);
out(ofile, fStudentName);
```

Macro expansion These "macro calls" would be "expanded" before the code was actually seen by the main part of the C++ compiler. Instead of the lines `out(ofile, fMark1)` etc the compiler would see the expansions:

```
ofile . write ( ( char * ) & fMark1, sizeof ( fMark1 ) );
ofile . write ( ( char * ) & fStudentName, sizeof (
fStudentName ) );
```

Framework defined macros The frameworks, particularly ET++ and those for the Windows environment, require lots of standard code to be added to each program-defined class that is derived from a framework-defined class. Macros have been provided so as to simplify the process of adding this standard code.

Declare and define macros Most of these framework macros occur in pairs. There will be a macro that must be "called" from within the declaration of a class, and another macro that must be called somewhere in the file containing the implementation code for that class. The first macro would declare a few extra member functions for the class; the second would provide their implementation.

For example, you might have something like the following:

- a "DECLARE_CLASS" macro that must go in the class declaration and must name the parent class(es)
- a "DEFINE_CLASS" macro that must be in the implementation.

If you wanted class `MyDoc` based on the framework's `Document` class, you might have to have something like the following:

```
class MyDoc : public Document {  
    DECLARE_CLASS(MyDoc, Document)  
    ...  
};
```

Example

with the matching macro call in the implementation:

```
// MyDoc class  
  
DEFINE_CLASS(MyDoc)  
  
Record *MyDoc::MakeEmptyRecord()  
{  
    ...  
}
```

If you are curious as to what such macros add to your code, you can always ask the compiler to stop after the macro preprocessing stage. You can then look at the expanded text which will contain the extra functions. A typical extra function would be one that returns the class's name as a text string. You don't really need to know what these functions are; but because they may get called by the framework code they have to be declared and defined in your classes.

Your framework's documentation will list (but possibly not bother to explain) the macros that you must include in your class declarations and implementation files. If you use the "Class Expert/Wizard" auxiliary programs (see next section) to generate the initial outlines for your classes, the basic macro calls will already have been added (you might still need to insert some extra information for the calls).

*Check the macros
that you must use*

31.2 ARCHITECTS, EXPERTS, AND WIZARDS

If you look at the code for the example "StudentMarks" program in Chapter 30, you will see that some is "standard", and much of the rest is concerned only with the initial construction and subsequent interaction with a display structure.

Every application built using a framework needs its own specialized subclasses of class `Application` and class `Document`. Although new subclasses are defined for each program, they have very much the same form in every program. The specialized `Application` classes may differ only in their class names (`LoanApp` versus `StudentRecApp`) and in a single `DoMakeDocument()` function. The `Document` classes also differ in their names, and have different additional data members to store pointers to program specific data classes. But again, the same

group of member functions will have to be implemented in every specialized `Document` class, and the forms of these functions will be similar.

Work with display structures is also pretty much stereotyped. The construction code is all of the form "create an `XX`-subwindow and put it here". The interaction code is basically "Which subwindow? Get data from chosen window."

Such highly stereotyped code can be generated automatically.

Automatic code generation

The "Architects", "Experts", or "Wizards" that you will find associated with your Integrated Development Environment are responsible for automatically generating the initial classes for your program and, in association with a resource editor, for simplifying the construction of any interactive display structures used by your program.

Symantec's Visual Architect packages all the processes that you need to generate the initial project with its code files, and interface resources. In the two Windows environments, these processes are split up. The Application Wizard (Expert) program generates all the basic project files. The Resource Workshop (or App Studio) resource editor program is used to add resources. The Class Wizard (Expert) program can be used to help create code for doing things like responding to a mouse-button click in an "action-button", or handling the selection of an option from a displayed menu.

Project generation

The project generation process may involve little more than entry of the basename for the new project and specification of a few parameters. The basename is used when choosing the names for specialized `Application` and `Document` classes; for example, the basename "My" would typically lead to the generation of a skeletal `class MyApp : public Application { }` and a skeletal `class MyDocument : public Document { }`. Some of the parameters that have to be entered in dialogs relate to options. For example, one of the dialogs presented during this project generation process might have a "Supports printing" checkbox. If this is checked, the menus and toolbars generated will include the standard print-related commands, and provision will be made for linking with the framework code needed to support printing. Other parameters define properties like the Windows' memory model to be used when code is compiled. (The Windows OS basically has a choice of 16-bit or 32-bit addressing.) When confronted with a dialog demanding some obscure parameter, follow the defaults specified in the framework's reference manuals.

In the Windows environments, once you have entered all necessary information into the dialogs, the Application Wizard/Expert completes its work by generating the initial project files in a directory that you will have specified. These generated files will include things equivalent to the following:

- a resource file (.rc) which will contain a few standard resources (maybe just a File/Quit menu!);
- a main.cp file which will contain the "main" program (naturally this has the standard form along the lines "*create a MyApp object and tell it to run*");

- MyApp.h, MyApp.cp, MyDocument.h, MyDocument.cp; these files will contain the class declarations and skeletal implementations for these standard classes;
- "build" and/or "make" file(s) (.mak); these file(s) contain information for the project management system including details like where to find the framework library files that have to be linked when the program gets built.

(Symantec's Visual Architect does not generate the project files at this stage. Eventually it creates a rather similar set of files, e.g. a .rsrc file with the resources, a main.cp, and so forth. The information equivalent to that held in a .mak file is placed in the main project file; it can be viewed and changed later if necessary by using one of the project's menu options.)

The MyApp.h, MyApp.cp files now contain the automatically generated text. The contents would be along the following lines:

```
// Include precompiled framework header
#include "Framework.h"

class MyApp : public Application {
    DECLARE_CLASS(MyApp, Application)
public:
    virtual void DoSplashScreen();
    virtual void DoAbout();
    virtual Document *DoMakeDocument();
};
```

*Automatically
generated MyApp.h*

```
#include "MyApp.h"
#include "MyDocument.h"

DEFINE_CLASS(MyApp)

void MyApp::DoSplashScreen()
{
    // Put some code here if you want a 'splash screen'
    // on start up
}

void MyApp::DoAbout()
{
    // Replace standard dialog with a dialog about
    // your application
    Dialog d(kDEFAULTD_ABOUT); // The default dialog
    d.PoseModally();
}

Document *MyApp::DoMakeDocument()
{
    return new MyDocument;
}
```

*Automatically
generated MyApp.cp*

The generated code will include effective implementations for some functions (e.g. the DoMakeDocument() function) and "stubs" for other functions (things like

the `DoSplashScreen()` function in the example code shown). These stub routines will be place holders for code that you will eventually have to add. Instead of code, they will just contain some comments with "add code here" directives.

Including headers

As noted in Chapter 30, you typically have to include numerous header files when compiling the code for any classes that are derived from framework classes or code that uses instances of framework classes. The framework may provide some kind of master header file, e.g. "Framework.h", which `#includes` all the separate header files.

Precompiled headers

This master header file may be supplied in "precompiled" form.

It takes a lot of time to open and read the contents of fifty or more separate header files and for the compiler to interpret all the declarations and add their information to its symbol table. A precompiled header file speeds this process up. Essentially, the compiler has been told to read a set of header files and construct the resulting symbol table. It is then told to save this symbol table to file in a form that allows it to be loaded back on some future occasion. This saved file is the "precompiled header". Your IDE manuals will contain a chapter explaining the use of precompiled headers.

User interface components

It is possible to build display structures in much the same way as was done in the example in the previous chapter. But you have to go through an entire edit-compile-link-run cycle in order to see the display structure. If one of the components is not in quite the right position, you have to start over and again edit the code.

It is much easier if the user interface components are handled using resources. The resource editor builds the data structure defining the form of the display. This data structure gets loaded and interpreted at run-time, with all the subwindows being created and placed at the specified points. Instead of dozens of statements creating and placing individual subwindows, the program code will have just a couple of lines like `RecordWindow rw; rw->GetResource(1001)`.

The resource editor used to build the views required in dialogs will allow you to move the various buttons, check boxes, edit-texts and other components around until they are all correctly positioned. The editor may even simulate the behaviour of the dialog so that you can do things like check the behaviour of "clusters of radio buttons" (if you select one radio button, the previously selected button should be deselected).

You will use the resource editor just after the initial project generation step in order to build the main windows and any dialogs that you expect to use in your program.

***Command numbers
associated with
controls***

When you add an interactive element (button, checkbox, edit-text etc) to a window you will usually be prompted for an associated "command number" (and, possibly, an identifying name). These command numbers serve much the same role as the window identifiers did in the `RecordFile` framework. When a control gets activated, a report message gets generated that will have to be handled by some other object. These report messages use the integer command numbers to indicate

what action is being requested. The process is somewhat similar to that by which changes to the `EditNum` subwindows in the `StudentMarks` program got reported to the `StudentRec` object so that it could collect the new data values.

Associating commands and command handlers

The next stage in developing the initial outline for a program typically involves establishing connections between a command number associated with a visual control (or menu item) and the object that is to respond to the request that this command number represents.

In the Windows environments, this is where you use the Class Expert (or Class Wizard). In the Symantec environment, this is just another aspect of using the Visual Architect.

The ideas of command handling are covered in more detail in Section 31.5. As explained there, the operating system and framework will work together to convert user actions (like key presses and mouse button clicks) into "commands" that get routed to appropriate "command handler objects". Commands are represented by little data structures with an integer command number and sometimes other data.

If you were using a framework to build something like the `StudentMarks` program, you would have the following commands to deal with: 1) New Document, 2) Open Document, 3) Quit, 4) New Record, 5) Delete Record, 6) ViewEdit record, 7) Change name, 8) Change assignment 1, You would have to arrange that the `Application` object dealt with commands 1...3; that the current `Document` object dealt with commands 4...6, and the current `Record` object dealt with the remainder.

Usually, you would need a separate member function in a class for each command that instances of that class have to deal with. So, the `Document` class would have to have `NewRecord()`, `DeleteRecord()` and `ViewEditRecord()` member functions. There would also have to be some mechanism to arrange that a `Document` object would call the correct member function whenever it received a command to handle.

Once a command has reached the correct instance of the appropriate class it could be dispatched to the right handler routine using a function like the following:

```
void StudentMarksDoc::HandleCommand(int commandnum)
{
    switch(commandnum) {
        case cNEW_REC: NewRecord(); break;
        ...
    }
```

*Example of
Commands and
Command Handlers*

*Dispatching a
command to the
class's handler
function*

This was basically the mechanism used in the `RecordFile` framework example and is essentially the mechanism used in Symantec's Think Class Library.

In the Windows environment, the idea is much the same but the implementation differs. Instead of a `HandleCommand()` member function with a `switch` statement, a Windows' "command handler" class will define a table that pairs command

numbers with function identifiers. You can imagine a something like the following:

```
Dispatch_table StuMarkDocTbl[] = {
    { cNEW_REC, StudentMarkDoc::NewRecord }
    { cDEL_REC, StudentMarkDoc::DeleteRecord }
    { cVIEW_REC, StudentMarkDoc::ViewEditRecord }
};
```

***Dispatch by table
lookup***

(It is not meant to be syntactically correct! It is just meant to represent an array whose entries consist of the records pairing command numbers and function names.) There will be code in the framework to search through such an array checking the command numbers; when the number matching the received command is found, the corresponding function gets called.

Now a substantial part of the code for these command handling mechanisms can be standardized. Stub functions like:

```
void StudentMarkDoc::NewRecord()
{
    // Put some code here
}
```

and the `HandleCommand()` function (or the corresponding table for a Windows program) can all be produced once the generator has been told the command numbers and the function names.

The "Class Expert" (or equivalent) will have a set of dialogs that you use to select a command number, pick the class whose instances are to handle that command, and name the function that will handle the command. (For many standard Windows commands, the name of the handler function is predefined.)

Once these data have been entered, the "Class Expert" will add the extra code with function declarations, and stub definitions to the appropriate files. Macros are used for the function dispatch tables; again, there are separate `DECLARE` and `DEFINE` macros. If you looked at the contents of the files at this stage you would find something along the following lines:

***StuDoc.h after Class
Expert has run***

```
// Include precompiled framework header
#include "Framework.h"

class StudentMarkDoc : public Document {
    DECLARE_CLASS(StudentMarkDoc, Document )
public:
    ...
protected:
    void    NewRecord();
    void    DeleteRecord();
    void    ViewEditRecord();
    ...

    DECLARE_RESPONSE_TABLE(StudentMarkDoc)
};
```

```
#include "StuDoc.h"
#include "StudentRec.h"

DEFINE_CLASS(StudentMarkDoc)

DEFINE_RESPONSE_TABLE(StudentMarkDoc, Document)
    COMMAND(cNEW_REC, NewRecord)
    COMMAND(cDEL_REC, DeleteRecord)
    COMMAND(cVIEW_REC, ViewEditRecord)
END_RESPONSE_TABLE

void StudentMarkDoc::NewRecord()
{
    // Put some code here
}
```

*StuDoc.cp after Class
Expert has run*

In the Symantec Visual Architect, this is the point where all the project specific files get created and then all the framework source files are copied into the project.

A Program that does Nothing – but does everything right

The code that is generated can be compiled and will run. The main window will be displayed, a File/Quit menu option will work, a default "About this program ..." dialog can be opened. Your program may even be able to read and write empty files of an appropriate type (file types are specified in the Application Expert phase of this generation process).

It does all the standard things correctly. But of course all the real routines are still just stubs:

```
void StudentMarkDoc::NewRecord()
{
    // Put some code here
}
```

You must now start on the real programming work for your project.

31.3 GRAPHICS

The programs that you build using the frameworks work with the "windows" supported by the underlying operating system (MacOS, Windows, or X-terminal and Unix). Output is all done using the OS graphics primitives (of course filestreams are still used for output to files).

The OS will provide a "graphics library" (strictly, the X-lib graphics library is not part of the Unix OS). This will be a function library with one hundred or more output related functions. These will include functions for drawing lines, arcs, rectangles, individual characters, and character strings.

*Library of graphics
functions*

Because these graphics functions are provided by the OS and not by the development environment, they may not be listed in the IDE manuals. You can get a list of the functions by opening the appropriate header file (Quickdraw.h on the Macintosh, Windows.h on Windows). The environment's help system may contain descriptions of the more frequently used graphics functions. Examples illustrating the use of the graphics library are not normally included in the manuals for the development environment. There are however many books that cover basic graphics programming for Windows or for Macintosh (the ET++ library uses graphics functions similar to those of the Macintosh, internally these are converted into calls to X-lib functions).

The basic graphics calls shouldn't in fact cause much problem; they are all pretty much intuitive. The function library will have functions like the following:

```
void MoveTo(short h, short v);
void Move(short dh, short dv);
void LineTo(short h, short v);
void FrameRect(const Rect *r);
void PaintRect(const Rect *r);
void PaintOval(const Rect *r);
void FrameArc(const Rect *r, short startAngle,
              short arcAngle);
void DrawChar(short ch);
void DrawText(const void *textBuf, short firstByte,
              short byteCount)
```

(If you look in the header files, you may encounter "extern Pascal" declarations. These are actually requests to the C++ compiler to put arguments on the stack in "reverse" order, as was done by a Pascal compiler. This style is largely an historical accident relating to the earlier use of Pascal in both Macintosh and Windows OSs.)

Although the basic functions are quite simple to use, there can be complications. These usually relate to the maintenance of collections of "window attributes".

Window attributes

Each window displayed on the screen has many associated attributes. For example, a window must have a defined depth (1-bit for black and white, or 8-bit colour, or 16-bit colour etc). There will be default foreground (black) and background (white) colours. There will be one or more attributes that define how bitmaps get copied onto the screen; these "mode" (or "brush" or "pen") attributes allow the program to choose whether to simply copy an image onto the screen or to use special effects like using the pattern of bits in an image to invert chosen bits already displayed on the screen. Other attributes will define how the characters are to be drawn; these attributes will include one that defines the "font" (e.g. 'Times', 'Helvetica' – different fonts define differently shaped forms for the letters of the normal "Roman" alphabet and, possibly, define different special characters), another attribute will specify the size of characters. There will be attribute to specify the thickness of lines. Still another will define the current x, y drawing point.

Ports, Graphics Contexts, Device Contexts

These attributes are normally held in a data structure, the form of which is defined by the graphics library. On the Macintosh, this will be a `Port` data

structure; the X-lib library refers to it as a "Graphics Context"; in the Windows environments it is a "Device Context".

Macintosh and X programmers don't have any real problems with these structures. A new `Port` (or equivalent) gets allocated for each window that gets opened. A program can change a `Port`'s attributes as needed. For Windows programmers, things can sometimes get a bit complex. The Windows OS preallocates some graphics related data structures; these must be shared among all programs. Device context structures (normally accessed via framework defined "wrapper" classes) may have to use shared information rather than allocating new structures. Developers working on Windows will at some stage have to supplement their IDE manuals with a textbook that provides a detailed presentation of Windows graphics programming.

31.4 PERSISTENT DATA

If the data for your program are something simple like a list of "student records" (as in the example in Chapter 30), you should not have any problems with input and output. The automatically generated code will include stub routines like the following (the function names will differ):

```
void StudentRecDoc::WriteTo(fstream& out)
{
    // Put your code here
}
```

The code that you would have to add would be similar to that illustrated for the `ArrayDoc` class in Chapter 30. You would output the length of your list, then you would iterate down the list getting each `StudentRec` object to write its own data.

The input routine would again be similar to that shown for class `ArrayDoc`. The number of records would be read, there would then be a loop creating `StudentRec` objects and getting them to read themselves. After you had inserted your code, the input function would be something like:

```
void StudentRecDoc::ReadFrom(fstream& in)
{
    // Put your code here
    // OK!
    int len;
    in.read((char*)&len, sizeof(len));
    for(int i = 0; i < len; i++) {
        StudentRec *r = new StudentRec;
        r->ReadFrom(in);
        fList.Append(r);
    }
}
```

You would however have problems if you were writing a program, like that discussed briefly in Chapter 23, where your data was a list of different

Heterogeneous collection

CircuitElements. If you remember the example, class CircuitElement was an abstract class; the objects that you would actually have would be instances of subclasses like class Battery and class Resistor.

You could store the various Battery, Resistor, Wire etc objects in a single list. This would be a "heterogeneous collection" (a collection of different kinds of data element). Most of your code would work fine with this collection, including the WriteTo() function which could be coded along the following lines:

```
void CircuitDoc::WriteTo(fstream& out)
{
    int len = fList.Length();
    out.write((char*) &len, sizeof(len));
    ListIterator LI((&fList));
    LI.Start();
    while(!LI.IsDone()) {
        CircuitElement* e =
            (CircuitElement*) LI.CurrentItem();
        e->WriteTo(out);
        LI.Next();
    }
}
```

Input problems!

The problems arise with the input. Obviously, the input routine cannot create CircuitElement objects because CircuitElement is a pure abstract class. The input routine has to create a Battery, or a Wire, or whatever else is appropriate for the next data item that is to be read from the file.

Solution: class identifying tokens in the file

If you have to deal with heterogeneous collections, you need some scheme whereby objects that write themselves to file start by *outputting some token that identifies their class*. Then you can have an input routine along the following lines:

```
void CircuitDoc::ReadFrom(fstream& in)
{
    int len;
    in.read((char*)&len, sizeof(len));
    for(int i = 0; i < len; i++) {
        Input of class token
        Create instance 'e' of appropriate class
        e->ReadFrom(in);
        fList.Append(e);
    }
}
```

This business about outputting class tokens when objects write themselves to file, then inputting these tokens and interpreting them so as to create an instance of the correct class, all involves a fair amount of code. But this code is now pretty much standardized.

Framework support

The standard code can be implemented by the framework. It will be called something like the "Streamable" component, or "Persistent" component, or "Storable" component. It requires a few extra functions in each class (e.g. a function to write a "class token", which is normally a string incorporating the class name). The framework may require extra macros in the class, e.g. a

`DECLARE_STREAMABLE` macro call in the class declaration and a matching `DEFINE_STREAMABLE` in the implementation. A class using the mechanism may also have to inherit from some `Streamable` base class. (Details differ between frameworks; read your framework manuals.)

There is a second problem related to input and output of complex structures. This one is actually rather harder to deal with; but, again, a "standard" solution has been found. This standard solution should be built into your framework's code.

The problem relates to pointer data members in networks. Figure 31.1 *Pointers* illustrates a simplified version of a "network" data structure for a word processor program. The document has a list of "paragraph" objects. A paragraph has two pointer data members; one links to a "style" object and the other to a block of text. Each paragraph has its own text, but different paragraphs may share the same style object.

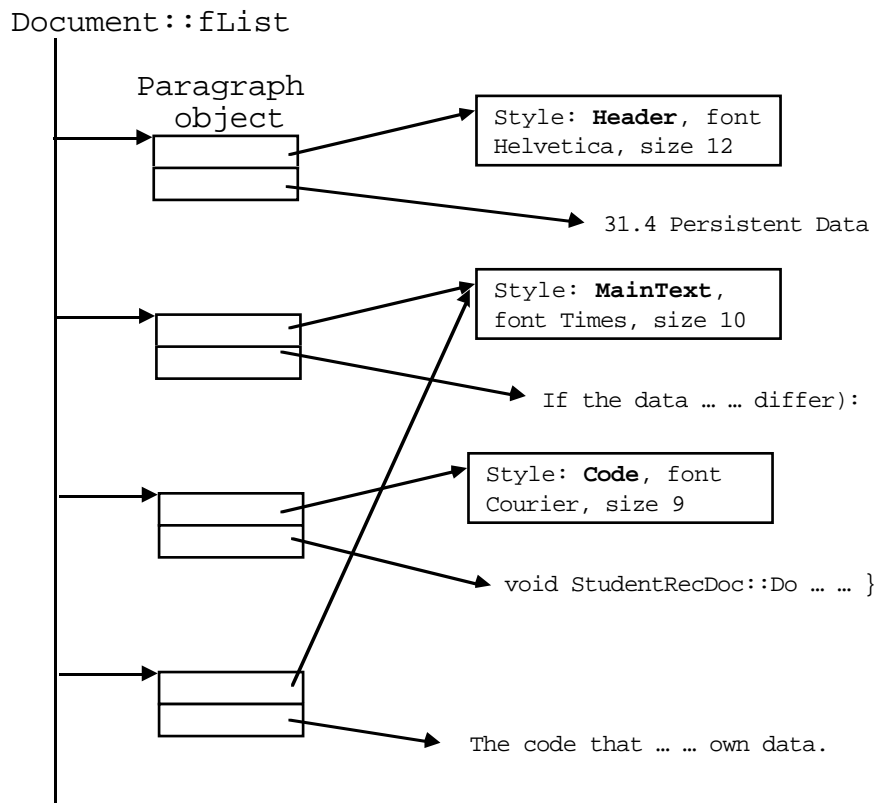


Figure 31.1 Example of "network" data structure with pointers.

You can't have anything like the following code:

```

class Paragraph {
    DECLARE_CLASS(Paragraph)
    DECLARE_STREAMABLE
public:

```

```

        ...
        void    WriteTo(fstream& out)
        ...
private:
    char    *fText;

    Style    *fStyle;
}

This is an incorrect implementation void Paragraph::WriteTo(fstream& out)
{
    Save "Paragraph" class token
    out.write((char*)&fText, sizeof(fText));
    out.write((char*)&fStyle, sizeof(fStyle));
}

```

That Paragraph::WriteTo() function does not save any useful data. Two addresses have been sent to a file. Those memory addresses will mean nothing when the file gets reread.

The following is slightly better, but still not quite right:

```

void Paragraph::WriteTo(fstream& out)
{
    Save "Paragraph" class token
    long len = strlen(fText);
    out.write((char*)&len, sizeof(len));
    out.write(fText, len);
    fStyle->WriteTo(out);
}

```

This would save the character data and, assuming Style objects know how to save their data, would save the associated styles.

However, you should be able to see the error. The Style record named "MainText" will get saved twice. When the data are read back, the second and fourth paragraphs will end up with separate styles instead of sharing the same style record. The word processor program may assume that paragraphs share styles so that if you change a style (e.g. the "MainText" style should now use 11 point) for one paragraph all others with the same style will change. This won't work once the file has been read and each paragraph has its own unshared style record.

What you want is for the output procedure to produce a file equivalent to the following:

```

Object 1: class Paragraph
20
31.4 Persistent Data
Object 2: class Style
Header Helvetica 12
End Object 2
End Object 1
Object 3: class Paragraph
286
If the data ...):

```

```

    Object 4: class Style
        MainText Times 10
    End Object 4
End Object 3
Object 5: class Paragraph
    69
    void ... }
    Object 6: class Style
        Code Courier 9
    End Object 6
End Object 5
Object 7: class Paragraph
    The code ... data.
    See Object 4
End Object 7

```

When the output has to deal with an object that has already been saved, like the style object for the "MainText" style, a reference back to the earlier copy is all that gets sent to file.

If a file in this form is read, it is possible to reconstruct the original network.

Now getting all of this to work is relatively complicated. It involves modified stream classes that may seem generally similar to the familiar `fstream` classes but which perform additional work. It is these modified streams that know how to deal with pointers and know not to save an object more than once.

The code is pretty much standardized and should be in your framework's "Streamable" component. You will have to read the reference manuals for your framework and the related tutorial examples to see how you are supposed to deal with pointer data members in your objects.

The framework's support for streamability may also include definitions of overloaded operator functions. These will allow you to write code in the form:

```
outfile << fStyle;
```

rather than

```
fStyle->WriteTo(outfile);
```

31.5 THE EVENT HANDLERS

Basics of Event Handling

Programs written for personal computers, and Unix/X-windows, are "event driven".

User actions like keystrokes and mouse clicks are picked up by the operating system. This generates "event" data structures that it queues for processing by the currently running program. Event data structures contain a number of data fields. One will be an integer event code (distinguishing keystroke event, left/right mouse button event etc). The interpretation of the others data fields depends on the event. The data structure for a mouse event will contain x, y coordinate information, while a key event has data to identify which key.

Events

Operator functions

As well as these primary input events, there are others like an "update window" event. The OS will itself generate this type of event. An "update window" event gets sent to a program when the viewable area of an associated window is changed (for instance, another window, possibly belonging to some other program, has been closed letting the user see more of the window associated with the "update" event.

The program picks these events up and interprets them. At the heart of all these programs you will find an event loop like the following:

```
"Main event loop"    do {
                        GetNextEvent(anEvent);
                        HandleEvent(anEvent);
                    } while(!Finished);
```

**Operating system's
function
GetNextEvent()**

Function "GetNextEvent()" will be provided by the operating system. It actually switches control from the current program to the operating system. The operating system will check the event queue that it has maintained for the program. If there is an event queued, the OS copies the information into the data structure passed in the call and causes a return from the function call. The program can then continue with the execution of its own "HandleEvent()" function.

If there are no events queued for a program, the call to "GetNextEvent()" will "block" that program. The OS will mark the program as "suspended" and, if there is another program that can run, the OS may arrange for the CPU to start executing that other program. When some input does arrive for the blocked program, the OS arranges for it to resume. (The OS might occasionally generate an "idle event" and return this to an otherwise "blocked" program. This gives the program the chance to run briefly at regular intervals so as to do things like flash a cursor.)

**Application::
HandleEvent()**

The framework will have this "main event loop" in some function called from `Application::Run()`. The function `Application::HandleEvent()` will perform the initial steps involved in responding to the user's input.

These initial steps involve first sorting out the overall type of event (key, mouse, update, ...) and then resolving subtypes. A mouse event might be related to selecting an item from a menu, or it could be a "click" in a background window that should be handled by bringing that window to the front, or it could be a "click" that relates to the currently active window. A key event might signify character input but it might involve a "command" or "accelerator" key, in which case it should really be interpreted in the same way as a menu selection.

This initial interpretation will determine how the event should be handled and identify the object that should be given this responsibility. Sometimes, the handling of the low-level event will finish the process. Functions can return, unwinding the call stack until the program is again in the main event loop asking the OS for the next event.

**Example, dealing
with a simple "update
event"**

For example, if the `Application` receives an "update window" event it will also get a "window identifier" as used by the OS. It can match this with the OS window identifiers that it knows are being used by its windows and subwindows, and so determine which of its window objects needs updating. This window object can be sent an "update" or "paint" request. The receiving window should deal with this request by arranging to redraw its contents. Processing finishes when the contents have been redrawn.

More typically, the objects that are given the low-level events to handle will turn them into "commands". Like "events", these commands are going to be represented by simple data structures (the most important, and only mutually common data field being the identifying command number). "Commands" get handed back to the Application object which must route them to the appropriate handler.

Conversion of low-level events to higher level commands

For example, if the Application object determines that the low level event was a mouse-down in a menu bar, or keying of an accelerator key, it will forward details to a MenuManager object asking it to sort out what the user wants. The MenuManager can deal with an accelerator key quite easily; it just looks up the key in a table and returns a "command" data structure with the matching command number filled in. Dealing with a mouse-initiated menu selection takes a little more time. The MenuManager arranges with the OS for the display of the menu and waits until the user completes a selection. Using information on the selection (as returned by the OS) the MenuManager object can again determine the command number corresponding to the desired action. It returns this to the Application object in a "command" data structure.

MenuManager turns a mouse event into command

A mouse click in a control window (an action button, a scroll bar, or something similar) will result in that window object executing its DoClick() function (or something equivalent). This will simply create a command data structure, filling in the command number that was assigned to the control when it was built in the resource editor.

A "control" window turns a mouse event into a command

The Application then has the responsibility of delivering these command data structures (and, also, ordinary keystrokes) to the appropriate object for final processing.

Normally, there will be many potential "handlers" any one of which might be responsible for dealing with a command. For example, the application might have two documents both open; each document will have at least one window open; these windows would contain one or more views. Views, windows, documents and even the application itself are all potentially handlers for the command.

The frameworks include code that establishes a "chain of command". This organization simplifies the process of finding the correct handler for a given command.

"Command Handler Chain"

At any particular time, there will be a current "target" command handler object that gets the first chance at dealing with a command. This object will be identified by a pointer data member, CommandHandler *fTarget, in the Application object. Usually, the target will be a "View" object inside the frontmost window.

"First handler", "Target", or "Gopher"

The Application starts the process by asking the target object to deal with the command (fTarget->DoCommand(aCommand);). The target checks to determine whether the command aCommand is one that it knows how to deal with. If the target is the appropriate handler, it deals with the command as shown in earlier examples.

If the target cannot deal with the command, it passes the buck. Each CommandHandler object has a data member, CommandHandler *fNextHandler, that points to the next handler in the chain. A View object's fNextHandler will identify the enclosing window (or subwindow). A window that is enclosed within another window identifies the enclosing window as its "next handler". A top level window usually identifies the object that created it (a Document or Application)

Passing the buck

as its next handler. Document objects pass the buck to the Application that created them.

The original target passes the command on: `fNextHandler->DoCommand(aCommand)`. The receiving `CommandHandler` again checks the command, deals with it if appropriate, while passing on other commands to its next handler.

For example, suppose the command is `cQUIT` (from the File/Quit menu). This would be offered to the frontmost `View`. The `View` might know about `cRECOLOR` commands for changing colours, but not about `cQUIT`. So the command would get back to the enclosing `Window`. `Window` objects aren't responsible for `cQUIT` commands so the command would get passed back to the `Document`. A `Document` can deal with things like `cCLOSE` but not `cQUIT`. So, the command reaches the `Application` object. An `Application` object knows what should happen when a `cQUIT` command arrives; it should execute `Application::DoQuit()`.

Keyed input can be handled in much the same way. If an ordinary key is typed, the `fTarget` object can be asked to perform its `DoKeyPress()` function (or something equivalent). If the target is something like an `EditText`, it will consume the character; otherwise, it can offer it to the next handler. Most `CommandHandler` classes have no interest in keyed input so the request soon gets back to the end of the handler chain, the `Application` object. It will simply discard keyed input (and any other unexpected commands that might arrive).

Defining the initial target

The initial target changes as different windows are activated. Most of the code for dealing with these changes is implemented in the framework. A programmer using the framework has only a few limited responsibilities. If you create a `Window` that contains several `Views`, you will normally have to identify which of these is the default target when that `Window` is displayed. (The user may change the target by clicking the mouse within a different `View`.) You may have to specify your chosen target when you build the display structure in the resource editor (one of the dialogs may have a "Target" field that you have to fill in). Alternatively, it may get done at run time through a function call like `Application::SetTarget(CommandHandler*)`.

Exceptions to the normal event handler pattern

In most situations, an input event gets handled and the program soon gets back to the main event loop. However, drawing actions are an exception.

A mouse down event in a `Window` (or `View`) that supports drawing results in that `Window` taking control. The `Window` remains in control until the mouse button is released. The `Window` will be executing a loop in which it picks up mouse movements. (Operating systems differ. There may be actual movement events generated by the OS that get given to the program. Alternatively, the program may keep "polling the mouse", i.e. repeatedly asking the OS for the current position of the mouse.) Whenever the mouse moves, a drawing action routine gets invoked. (This loop may have to start with some code that allows the `Window` to specify that it wants to "capture" incoming events. When the mouse is released, the `Window` may have to explicitly release its hold on the event queue.)

Drawing routines

Most of this code is framework supplied. The application programmer using the framework has to write only the function that provides visual feedback during the drawing operation and, possibly, a second function that gets called when drawing activity ends. This second function would be responsible for giving details of the

drawn item to the `Document` object so that this can add the new item to its list of picture elements.

CommandHandler classes

The frameworks provide the `CommandHandler` class hierarchy. Class `CommandHandler` will be an abstract class that defines some basic functionality (your framework will use different function names):

```
class CommandHandler : public ? {
...
public:
    void SetNextHandler(CommandHandler* newnext)
        { fNextHandler = newnext; }
...
    void DoCommand(...);
    void HandleCommand();
...
    void DoKeyPress(char ch)
        { fNextHandler->DoKeyPress(ch); }
protected:
    CommandHandler *fNextHandler;
};
```

The `SetNextHandler()` function is used, shortly after a `CommandHandler` gets created, so as to thread it into the handler chain. Command handling may use a single member function or, as suggested, there may be a function, `DoCommand()`, that determines whether a command can be handled (passing others up the chain) while a second function, `HandleCommand()`, sorts out which specialized command handling function to invoke.

The frameworks have slightly different class hierarchies. Generally, classes `Application`, `Document`, and `Window` will be subclasses of `CommandHandler`. There will be a host of specialized controls (button, scroll bar, check box etc) that are also all derived from `CommandHandler`. There may be other classes, e.g. some frameworks have a `Manager` class that fills much the same role as `Document` (ownership of the main data structures, organization of views and windows) but is designed for programs where there are no data that have to be saved to file (e.g. a game or a terminal emulator program).

The framework code will handle all the standard commands like "File/New", "File/Save As...". The developer using the framework simply has to provide the effective implementations for the necessary functions like `DoMakeDocument()`.

***Framework handles
standard commands***

All the program specific commands (e.g. a draw program's "add line", "recolor line", "change thickness" commands etc) have to be integrated into the framework's command handling scheme. Most of these extra commands change the data being manipulated; a few may change parameters that control the display of the data. The commands that change the data should generally get handled by the specialized `Document` class (because the `Document` "owns" the data). Those that change display parameters could be handled by a `View` or `Window`.

***Application specific
command handling
added to Document
and View classes***

The Class Expert program (or equivalent) should be used to generate the initial stubs for these additional command handling routines. The developer can then place the effective data manipulation code in the appropriate stub functions.

With the overall flow of control largely defined by the framework, the concerns of the development programmer change. You must first decide on how you want to hold your data. In simple cases, you will have a "list" of data elements; with this list (a framework supplied collection class) being a data member that you add to the document. Then, you can focus in turn on each of the commands that you want to support. You decide how that command should be initiated and how it should change the data. You then add the necessary initiating control or menu item; get the stub routines generated using the "Class Expert"; finally, you fill in these stub routines with real code.

On the whole, the individual commands should require only a small amount of processing. A command handling routine is invoked, the data are changed, and control quickly returns to the framework's main event loop. Of course, there will be exceptions where lengthy processing has to be done (e.g. command "Recalculate Now" on a spreadsheet with 30,000 formulae cells).

31.6 THE SAME PATTERNS IN THE CODE

The frameworks just evolved. The Smalltalk development environments of the early 1980s already provided many of the framework features. The first of the main family of frameworks was MacApp 1.0 (≈1986). It had the low level event handling, conversion of OS events into framework commands, and a command handler hierarchy of a fairly standard form. Subsequently, framework developers added new features, and rearranged responsibilities so as to achieve greater utility.

Recently, some of those who had helped develop the frameworks reviewed their work. They were able to retrospectively abstract out "design patterns". These "design patterns" define solutions to similar problems that frequently reoccur.

Pattern: Chain of Responsibility

The last section has just described a pattern – "Chain of Responsibility". The arrangements for the command handler chain were originally just something necessary to get the required framework behaviour. But you can generalize. The "Chain of Responsibility" pattern is a way of separating the sender of a request from the object that finally handles the request. Such a separation can be useful. It reduces the amount of "global knowledge" in a program (global knowledge leads to rigidity, brittleness, breakdown). It increases flexibility, other objects can be inserted into the chain of responsibility to intercept normal messages when it is necessary to take exceptional actions.

Another pattern, "Singleton", was exploited in Chapter 29 for the WindowRep class. The ADCollection and BTCollection classes of Chapter 29 illustrate the Adapter pattern.

These "design patterns" are a means for describing the interactions of objects in complex programs. The published collections of patterns is a source of ideas. After all, it documents proven ways of solving problems that frequently reoccurred in the frameworks and the applications built on those frameworks.

As you gain confidence by programming with your framework you should also study these design patterns to get more ideas on proven problem solving strategies. You will find the patterns in "Design Patterns: Elements of Reusable Object-Oriented Software" by E. Gamma, R. Helm, R. Johnson, and J. Vlissides (Addison-Wesley).

EXERCISES

- 1 Use your framework to implement a version of the StudentMarks program. Employ a (framework provided) list or a dynamic array class for the collection.

Coding guidelines

*These are guidelines for CSCI121 Spring Session 1997. They do **not** have "departmental status". Lecturers for other subjects require different styles. Confirm with lecturer when starting a subject.*

These guidelines contain internal cross references indicated by underlining.

Program

"Physically" a program is built by compiling and linking together:

- a file with the main program (usually named main.cpp)
- a number of separate modules (X.h/X.cpp file pairs)
- functions taken from standard libraries.

Module

A module is a separately compilable part of a C/C++ program. A module will consist of two files – the "header" file and the "implementation" file. The header file defines the services that the module provides to the rest of the program. The implementation file contains the code that is used to supply those services.

There are different styles of module. These suit different approaches to program design.

A module for a program designed using top-down functional decomposition:

The header file will contain:

- A list of function prototypes and comments summarizing the functions provided (using a term from another programming language, you could say that these are the functions that are "exported" by the module).
- Constants and typedefs might also be in the header file; and sometimes there may even be struct declarations. However, these should be minimized.

The implementation file will contain:

- Declarations of any structs used only by functions defined in that module.
- Definitions of any constant data used by those functions.
- Definitions of any static (file scope) variables used to retain state information between successive calls to functions "exported" by the module.
- Definitions of the functions listed in the module's header file (the "exported" functions).
- Definitions of any static auxiliary functions needed to simplify implementation of "exported" functions.

A module for a program designed using an object based (or object oriented) approach

A module for an object based program will either define a single class, typically one representing an abstract data type such as a "PriorityQueue" that may be used in many different programs, or a group of closely collaborating classes whose instances are always used together (a simple example would be a module that defines class List and class ListIterator).

The header file would contain:

- class declaration(s),
- Definitions of "inline functions",
- Declarations of any "free functions" related to the class;
- and, of course, some comments explaining the role that the class's instances fulfil;

The implementation file would contain:

- Declarations of auxiliary structs or even a declaration and definition of an auxiliary class (e.g. class BinaryTree might have class TreeNode declared and defined in its implementation file);
- Definitions of all member functions of the class(es) apart from those defined by inline functions in the header;
- Definition of any associated free functions;
- Definition of any class data members (static data members)

Definition and declaration

In C++:

- a "forward" declaration
introduces the name of a function, struct, enum or class.
- a declaration
introduces a variable, or specifies the members of a struct or class
- a definition
allocates space for a variable, provides code for a function, introduces name and allocates space and value for a constant.

By default, a variable declaration is also its definition. The following statements all instruct a compiler to record a variable name in its symbol table and arrange allocation of storage space. (If the variable is a global or filescope variable, that really does mean arrange allocation of storage space. If the variable is a local variable of a function, it means recalculating the size of the "stack frame" for that function and allocation of a relative position within the stack frame where the variable will be located.)

```
int gCount;  
int gStartXCoord = 30;  
char gMsg[] = "Hello World";  
static int sFrequencyCount;  
static char *sErr = "Incorrect number of arguments on command line";
```

If you want to declare that a variable, e.g. gWorld, does exist but is *not* to be defined in the current module, you must write an extern declaration:

```
extern int gWorld;
```

The appropriate use of "extern" variable declarations is explained later in the note concerning "global variables".

Variables can only be defined once in any scope.

A function can be declared more than once, just write down its prototype. It should have only one definition.

```
void DoThis();           // declaration  
void DoThis();           // declared again, never mind  
void DoThis();           // compiler getting bored reading this  
void DoThis();  
  
void DoThis() { ... }    // finally the definition
```


Make sure that the prototype in a declaration matches the definition; otherwise the compiler will think that you are talking about different functions.

The name of a struct or class type may be declared any number of times, but there must be only one specification of its members. Declarations like the following ("forward declarations") simply tell the compiler that the various structs and classes will be fully declared and defined properly *somewhere*; meantime the compiler is to accept the name as the name of a data type and deal with any code that involves pointers to or references to data of these types. (If the compiler has simply seen a name declared, it can't deal with any code that tries to make use of variables of that type. The compiler must have read a complete declaration in order to deal with code that declares instances of a class or struct type, attempts to declare the form of another structure with a data member of a type for which only the name is known, or invokes member functions or accesses data members via a pointer or reference to a variable of that type.)

```
struct Thing1;           // all these are "forward declarations"
struct Thing2;
class MyClass;
class MyTree;
class MyTreeNode;
```

A struct or class is declared properly when you specify its members:

```
struct Thing1 { int fXPos; int fYPos; char *fName; };
class MyThing {
public:    MyThing ();
        int Add(void* data, int key);
        ...
private:
        ...
        static int      sMyThingClassCounter;
};
```

Note that a class declaration does not define any static (class) data members. These must be explicitly defined in the implementation file:

```
int MyThing::sMyThingClassCounter = 0;
```

Class Declaration

A class declaration should have the following standard form (note, not allowing for inheritance and "object oriented" styles):

```
class MyClass {
public:
    // First, constructors
    // (some classes have none --- or more accurately they use a default
    //      constructor provided by the compiler)
    // Most classes have several --- corresponding to different ways
    // to initialize instances of the class
    MyClass();
    MyClass(const MyClass& other);
    ...
    // Possibly a "destructor"
    // not all classes require a destructor
    ~MyClass();
    // Public member functions grouped in categories
    // access functions --- that simply get info from instance
    // of class --- should be declared const
    int NumItems() const;
    const char *Name() const;
    ...
    void ChangeName(const char* newName);
    ...
    // Standardize names of functions for input and output of class
```

```
// instance
void    ReadFrom(istream& input);
void    WriteTo(ostream& output) const;
...
// You may have "static" (class) public member functions,
//    but probably you won't
static int Counter();
// If your class insists on having friends, name them here
friend class MyOtherClass;
private:
    // private auxiliary functions that help implement the public
    //    functions
    int    AuxDoAdd(...)
    ...
    // data members
    int    fIdentifier;
    char    *fName;
    ...
    // possibly some static class data
    static char *sNameInfo[];
};

// inline functions here
const char *MyClass::Name() const { return fName; }

// declare (maybe define) any associated free functions
inline istream& operator>>(istream& input, MyClass& myc)
    { myc.ReadFrom(input); return input; }
inline ostream& operator<<(ostream& output, const MyClass& myc)
    { myc.WriteTo(output); return output; }
```

Inline functions are primarily a minor optimisation feature. As a general rule, only define a function as inline if it is a "one-liner" such as a function providing read access to a private data member.

Function prototype

A function prototype specifies:

- name of function
- type of result
- types of arguments
- if the function is a member of a class, there may be an additional const qualifier (indicating whether the function can change the instance for which it was invoked)
- if exceptions are being used, there may be an additional exception specification (exceptions are not covered in CSCI121).

Some lecturers insist that function prototypes should give solely the types of arguments:

```
void MoveTo(int, int);
```

Others like the function prototypes to identify the arguments:

```
void MoveTo(int horiz, int vert);
```

Check your lecturer's preferences, or lose marks. For CSCI121, you will specify the names of arguments in function prototypes.

Argument names given in a function declaration don't have to match those used in the actual definition. So you could, if you were stupid, declare `MoveTo()` with arguments `horiz`, and `vert` then define it to use arguments `x` and `y`:

```
void MoveTo(int x, int y) { ... x ... y }
```

See also "function interface" and "functions that return error codes" for further comments on functions.

Header file

As noted earlier, a header file defines the "services" (functions, classes, whatever) provided by a separately compilable module. Header files are `#included` in modules that are "clients" for those services. If you want to use a function defined in a different module (possibly one of the standard library modules) you must `#include` the appropriate header file in your own module (possibly in your `.h` header file, possibly in your `.cpp` implementation file – a little more advice given later).

In general, a header file can contain:

- `#include` directives on other header files
- function prototypes
- inline function definitions
- struct declaration
- class declaration
- const definitions
- enum declarations
- typedef statements
- extern variable declarations

Although legal and often quite appropriate, const enum typedef and extern variable declarations can all cause problems with large projects and their use should be minimized.

A header file must NOT contain:

- variable definitions – either global or static
- function definitions other than those explicitly or implicitly declared as inline

A variable definition in a header file is always an error. The compiler/linker system will pick up the error if the variable has global scope (it will get multiply defined, once for each implementation file that `#includes` the header). A variable declared in a header as having static scope is a more subtle problem. It won't be picked up by the compiler or linker. Each compiled module that `#includes` the header will have its own separate copy of the variable. The programmer probably thinks that there is a common shared copy and consequently can't understand why the program doesn't run correctly.

The appearance of a function definition in a header file will result in multiple copies of the code being generated and a link error. (Of course, inline function definitions are an exception to this rule.)

Although all the elements listed above can appear in any header file, the type of elements mainly used will vary according to whether the header is for a program designed by top down functional decomposition or for an object based program. A header file should follow the standard format; these are shown below for both styles (along with an example of how not to write a header file).

Example 1:

This is for a module in a program designed using top-down functional decomposition. It had a component that stores data records in a disk file. It defines a set of functions that can be used by the rest of the program and has a structure declaration that specifies the kind of data that should be passed to these functions. Note how the author has adopted a consistent naming style.

`datamanager.h`

```
#ifndef DATAMAN_H
#define DATAMAN_H
/*
Header for datamanager component that keeps
records in name filed. Provides access to keyed records
as described in project docs.
*/
struct dtmn_data {
    void    *p_data;
    int     datasize;
```

```
};

struct dataman; // for "opaque" pointer to a datamanager

// open returns NULL if couldn't open the file, otherwise
// it is a pointer to the datamanager that works with the named file
dataman *dtmn_open(const char *filename);
void dtmn_close(dataman *manager);
// "get" tries to load record with specified key,
// it allocates space on heap for actual data and puts
// size and address in record passed as argument
// return 0 if all OK, non-zero error codes see main docs.
int dtmn_get(dataman *manager, long key, dtmn_data& record);
// "save" puts data described by record into file
// if there is already an entry with that key, the old entry is replaced
void dtmn_save(dataman *manager, long key, const dtmn_data& record);
// "delete", remove entry, with specified key, from file
void dtmn_delete(dataman *manager, long key);

#endif
```

Example 2

The following illustrates **poor** style for a module.

The header describes a set of apparently unrelated functions. The module lacks the conditional bracketing that should always be present. The naming styles are inconsistent. The compiler will not be able to deal with this header if the `iostream` header is not `#included` before this one. The function signatures show lack of consistency with respect to passing of arguments.

stuff.h

```
const int RecordSize = 53;
const int SizeRecrd = 34;
struct Record { int l;
char name[71]; double DataValue;
long Intvalue;      char thing[SizeRecrd];
      char Thing[RecordSize];};
int StopIt(int, int, int, int);
void WhenReady(double);
void PrntRecord(ostream&, Record);
void Readrec(Record&, istream&);
int chck_rcrd(Record*);
void Matrixmultiply(double m[38][38], double n[38][38],
double a[38][38]);
void i(double l[38][38]);
typedef long Quendop;
const max = 103;
const int SzeRecrd = sizeof(Record);
#define speedygonzales
Quendop Fudge(Quendop);
void foo(int*, int&, int i=max);
```

Don't ever write anything like that! A module is supposed to represent a recognizable component part of a program, not a heap of junk and odd and ends.

Example 3:

Here is a header for a module that defines a single class, `MyClass`. A unique name, here **MYCLASS_H**, shown in bold should be made up to identify the module:

myclass.h

```
#ifndef MYCLASS_H
#define MYCLASS_H

/*
Block comment explaining what this class provides to rest of program.
```

```
*/

#ifndef _STRING_H
#include <string.h>
#endif

class ostream;
class MyClass {
public:
    ...
    int      CheckName(char* str) const;
    void      WriteTo(ostream& out) const;
    ...
private:
    char      *fName;
    ...
};

inline int  MyClass::CheckName(char *str) const { return 0==strcmp(str,fName); }
#endif
```

The entire contents of a header file should be bracketed by the `#ifndef`, that tests whether this header has already been seen by the compiler, and matching `#endif` (essentially this prevents mistakes that would occur if the same header file was `#included` more than once by a client module).

Here there is a conditional include of `<string.h>` (the compiler will read the system file `string.h` at this point if this file has not previously been read). This is necessary because later, with the inline function definition for `CheckName()`, we have something in the header file that depends on the contents of `string.h`.

Next, there is a simple "forward" declaration of the class name `ostream`. This is necessary as there are references to `ostream` objects in the function prototypes of `MyClass`. The compiler must be assured that `ostream` is a valid name of some type of data object. We don't need to `#include <iostream.h>` as there is nothing in this header that depends on any property of an `ostream` object (we simply need to know that they exist). But, of course, if we don't `#include iostream.h` here, the implementation file for `MyClass` will need to `#include` it so that the compiler can check any output operations that we invoke in the code of `MyClass::WriteTo()`.

Inline function definitions, if any, should come at the end of the header file.

Leave some blank lines after the `#endif` (this avoids the possibility of an obscure bug that can otherwise occur when shifting files from PC or Mac to Unix).

Implementation file

As with header files, there are some variations in the prototypical form for an implementation file for a program using functional decomposition and one using classes.

Implementation file for top-down functional decomposition

`datamanager.cpp`

```
/*
Block comment
    more details about module
*/

// next get includes on header files
#include <stdlib.h>
#include <fstream.h>
#include ...
#include "datamanager.h"

// watch out for "header ate my file" bug --- a header that has a mistake
// like an incorrectly matched block comment bracket, or even a missing
// blank line after final #endif, can cause compiler to ignore all subsequent
```

```
// input to end of file!

// next declare any const values,
// then if there are auxiliary structs used solely within this
// module get them declared

const int kBUFFERSIZE = 256; // max allowed for input line
...

struct dataman {
    fstream fMainFile;
    fstream fIndexFile;
    ...
};

// Globals?????
//   Any Gobals should be defined in
//   a special "globals" module with a header file that has a
//   set of extern data declarations and an implementation file
//   that has just a set of variable definitions.
// In rare cases, where there are only one or two global variables
// these can be define in main.cpp and referenced via explicit
// external declarations in other modules.

extern int gErrorNumber; // defined in main.cpp

// In even rarer cases, a module other than main.cpp or globals.cpp can
// define global variables that are referenced from other modules
// The header file for the defining module should contain the extern
// declaration for these "exported" variables
// int aValueWeMightExport; // put extern declaration in header
// if it is really needed as global

// Filescope variables
// These are variables that maintain state data between successive calls
// to functions in this module.

static int sCountOfFilesNowOpen = 0;

// If you will be defining auxiliary static filescope functions
// then list their declarations here (definitions later)

static long ComputeHashValue(const char *source);
static long KeyToAddress(long key);

// Now get the function definitions
// Order? Just something reasonably logical
// Repeat the static linkage specifier on any functions that are supposed
// to be filescope.

dataman *dtmn_open(const char *filename)
{
    ...
}

...

static long KeyToAddress(long key)
{
    ...
}
```

Implementation file for classes

MyClass.cpp

```
/*
Block comment
    more details about class
*/

// next get includes on header files
#include <stdlib.h>
#include <iostream.h>
...
#include "MyClass.h"

// const values,

const int KMSGSTRING[] = "Hello World";
...

// then if there are auxiliary structs (or classes) used solely within this
// module get them declared (& defined if a class with member functions)

struct KeyPtrPair { void *fdata; int fkey; };

// Globals?????
//   Gobals in an object based program - no.
//   There are conventions that you will learn later, such as
//   singleton classes, that you can use in place of globals.
//   For now, don't even think about them in class based modules

// Filescope variables???
//   Rare, mostly they are replaced by static class members
//   but there are occasions when filescope can be used.
//   For now, forget them in class based modules

// If this module is for a class with "static" class data members, define them
// here. Note, in this case you do NOT repeat the keyword static.

char *MyClass::sNameInfo[] = { "Tues", "Wed", "Thurs" };

// Auxiliary static filescope functions (not class members)
// You may have such functions even if you are using classes. For example
// you may need code to format a string in a particular way; this not really
// a "behaviour" of your class, simply something that you need to do in
// several parts of your class's code.

static void ReformatString(const char *source, int control, char buff[], int
    maxl);

// Now get the function definitions
// Do NOT use static qualifier if defining a static member
// function of a class.

MyClass::MyClass()
{
    ...
}

...

static long ReformatString(const char*source, int control, char buff[], int maxl)
{
    ...
}
```

Inline function

Define a function (in a header file) with the qualifier "inline". The compiler then has the option of coding a separate function that gets called at each point it is used, or arranging to expand the call into the actual data manipulation steps.

Primarily a minor optimisation to gain speed on simple functions. Main use will be for functions used to access values held in data members of objects.

Free function

A function that is not a member of a class is a "free function". Examples would include all the functions in the maths and string libraries. Generally, you should try to minimize the number of free functions that you define with external (global) scope.

Static auxiliary function

In a procedural style program, a "static auxiliary function" fulfils much the same role as a private auxiliary function in a class.

The functions exported by a function-style module (analogues of the public functions of a class) may be too complex to implement as single functions. Consequently, you will invent auxiliary functions that do part of their work. These auxiliary functions should only be used from within your principal functions, so you want to hide them.

You hide them by giving them static linkage, making them filescope rather than global scope. Their names disappear before your code reaches the linker – so code from other modules cannot possibly call them.

Local variables

Local automatic variables are defined within the body of a function or block. Their scope extends from the point of definition to the end of the block where they are defined. They are allocated on the stack. Their lifetime is the lifetime of the enclosing scope (block or function).

Some lecturers insist that all variables introduced in a block be defined at the start of that block. Others follow standard C++ practice and introduce variables at the point where they are first used. (Different preferences can easily cost you 0.5 marks in a 5 mark assignment.)

The following is standard C++ style (it may violate your lecturer's preferences):

```
int FillBuffer(istream& input, char buffer[], int maxlen, int& errno)
{
    /*
    Fill buffer with characters from input.
    Input must be in correct message form, starting with stx code then
    ...
    Return 0 if successful.
    Return 1 if error (in which case set reference variable errno to
        error number, 1 = EOF encountered,
        2 = incorrect message structure, 3 =
    ...
    */
    errno = 1; // most likely error is end of file
    int ichar; // use int for character, simplify eof check
    ichar = input.get();

    if(ichar == EOF) return 1;

    const char STX = 3;
    if(ichar != STX) { errno = 2; return 1; }
```



```
// Next two characters represent length
ichar = input.get();
if(ichar == EOF) return 1;
int highdigit = ichar;

ichar = input.get();
if(ichar == EOF) return 1;
int lowdigit = ichar;

int msglen = 256*highdigit + lowdigit;

if(msglen > maxlen) { errno = 2; return 1; }

// have to do checksum on data, initialize to 0
long checksum = 0;

// Use ptr when filling in destination array
char *ptr = buffer;
// read characters
for(int i = 0; i<msglen; i++, ptr++) {
    int nxtch = input.get();
    if(nxtch == EOF) { return 1; // end of file error }
    *ptr = nxtch;
    checksum += nxtch;
}
...
}
```

Here variables (and constants) are being introduced as needed. Scope of `ichar` (i.e. places where you can reference this variable) is almost the whole body of the function. Scope for `msglen` is from point of declaration to end of function. Scope of `nxtch` is just the little block where it is defined.

Scope of 'i' ? A problem! Most compilers don't comply with C++ standard. See comments on Limited scope variables

Limited scope variables

The proposed standard for C++ suggests:

- loop control variables, introduced in the initialization part of a for statement should have a scope that includes only the for statement and any block it controls;
- a variable can be introduced in an if statement – its scope is the code guarded by that if;

Most compilers don't yet comply; some do.

The most likely problem that you will encounter is the following:

```
for(int i=0; i < kLEN; i++) { ... DoSomething(i); ... x[i] ... }
...
for(int i=0; v[i] != 0; i++) { ... }
```

Strictly that is legal C++. Most compilers will give an error at the second for loop "double definition of variable i".

Since you are likely to be trying to move code around (PC, Mac, Unix) and may be using more than one compiler on any single platform, it is probably better to play safe until the compilers are better standardised. Either define a reusable loop control variable separately from any loop:

```
int i;
for(i=0; i < kLEN; i++) { ... DoSomething(i); ... x[i] ... }
...
for(i=0; v[i] != 0; i++) { ... }
```

or use a distinct loop control variable in each loop:

```
for(int i1=0; i1 < kLEN; i1++) { ... DoSomething(i1); ... x[i1] ... }
...
```

```
for(int i2=0; v[i2] != 0; i2++) { ... }
```

The specification involving if statements should allow things like the following:

```
if((int ch = in.get()) != EOF) { ... = ch ... }  
...  
char ch;  
...
```

Best to avoid such styles until all compilers agree on the meaning.

Filescope variables

Variables declared in an implementation file, and not within the body of any function, that have a "static" qualifier can only be referenced by functions declared within the rest of that file.

Such variables are used:

- when implementing an abstract data type using simply the C base language without the C++ class extensions
- to preallocate some scratch pad work area e.g. a buffer for reading characters from an input stream or doing some formatting.

Global variables

Variables declared in implementation files, outside of any function, (and not qualified by the static linkage qualifier) possess "external" or "global" scope. Such variables can be accessed and manipulated by any of the code of a program.

Although such variables may appear to be "hidden" in an implementation file, any other part of the code can simply name them in "external" data declaration statements. The linker will then dutifully tie the reference in the "foreign" module to the variable in the module where the variable is defined. This tying together of separate modules through some global variables is something you should try to avoid.

Such ties between modules are not obvious when reading the code. But they mean that when debugging or extending the code, you can't restrict your attention to a single module – the variables that you want to work with may be being changed from any other part of the program.

You should aim to minimize the number of globals that you use.

You should group all globals in a single "globals" module with a header that names the variables in extern data declarations and an implementation file that defines the variables. (Files `globals.h`, `globals.cpp`)

Later, you will learn ways of using classes to better encapsulate and control access to globals.

Constants

Use named constants rather than "magic numbers" in your code.

Define constants at the start of a file (header or implementation *as appropriate*) after any `#include` statements.

Adopt a consistent way of naming constants. The following convention is recommend:

small letter c or k
rest of name capitalised

e.g.

```
const int kLENGTH = 56;
const char kFILENAME[] = "input.txt";

const char *kMONTHS[] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
```

What does he mean "header or implementation" as appropriate? Do your clients need to know the value of the constants? If yes put them in the header, if no put them in the implementation.

Enums

Define enums with names that make them distinctive; following illustrate preferred style:

```
enum EFontType { eTIMES, eHELVETICA, eCOURIER };
```

Main use for enums should be making function interfaces more type secure. Often you will have a function one of whose arguments is essentially an integer from a limited range, as in the following definition for a print function:

```
const int TFont = 0;
const int HFont = 1;
const int CFont = 2;

void Print(const char *str, int fonttype);
```

Operation of Print() is only defined if second argument is 0, 1, 2. But if declared like this, the compiler can't spot errors like

```
Print("hello world", 72);
```

If you changed from an integer argument to an enumerated type, the function interface would be more secure:

```
void Print2(const char* str, EFontType aFontType);
```

Compiler will complain if Print2() given anything other than eTIMES etc.

Type names

Classes and structs must be given meaningful names. An isolated code fragment given as an example can use struct thing1 or class MyClass. Your code better have struct KeyRecord and class PhoneDirectory or whatever. Typenames corresponding to structs and classes should have names that start with a capital letter; as with variables, if the name consists of multiple words you should follow a convention like capitalisation of initial letter of each word or underscore separators between words (see later in section on variable names).

You can use typedef to introduce synonyms for existing types. You will find that there are many synonyms for integers etc defined in various system header files (following examples are from Unix's stddef.h and sys/types.h):

```
typedef unsigned int    size_t;
typedef long            whcar_t;
typedef int             ptrdiff_t;
typedef unsigned char   uchar_t;
typedef long            daddr_t;
typedef char*           caddr_t;
```

Note the systematic naming conventions, these synonym types all end with _t.

Such synonyms have a limited use; they help to clarify role of arguments to a function, e.g. void WriteTo(daddr_t position, caddr_t data, long len) is clearer than WriteTo(long, char*, long).

However they are not new types. The compiler won't object if you add a `wchar_t` variable to a `daddr_t` variable.

Function names

Functions that you define should have names that explain their role in the program (or, if member functions, the behaviour that they define for the class); generally, the names should start with capital letter and follow the convention of capitalising embedded words.

Choosing a good function name?

- A few are conventional. In a program that accepts commands from a user you should expect a central "HandleCommand()" or "MenuSelect()" function; the functions that it calls to process the inputs will have names that correspond to the user commands, such as: `DoAddNewRecord()`, `DoFindRecord()`, `DoSave()`, etc. A class for an abstract data type will have names that relate to the services it provides; so for example, a class `Stack` will have member functions `Push()` and `Pop()`. Use consistent names when implementing classes – so in all your classes where the instances can read and write their data you should have members `ReadFrom()` and `WriteTo()`.
- A function (member, static auxiliary, or global free function) that returns a value should be named for the value it produces – `Identifier()`, `Name()`, `Salary()`. Some people like the convention that functions that return boolean values all have names that end with `_p` (for predicate) – so they would prefer `Empty_p()` to `Empty()` etc; if you adopt such a convention, stick to it throughout all the code of a program (don't have some functions with `_p` and others without).
- Names that combine a verb and a noun are often the best for conveying the role of a procedure (void function).
- A suffix or prefix `Aux` can be useful in identifying (private) auxiliary helper functions for a principal function (public member of class or exported function of function style module) – e.g. `AuxAdd()` for an `Add()` function.

Variable names

Conventions:

First letter(s)	Variable type
g	Global
s	Static filescope variable
f	(Instance) data member of class
s or sc	static data member of class
a or the	arguments
p or ptr	pointers
other lower case	local variables of functions

Local variables and arguments generally have names that start with a lower case letter. For arguments, it is common to have names starting with 'a' or 'the' e.g. `void DoUpdate(SalesRec& theSalesRec, const char *aName, int theCount)`. If the variable is a pointer, it is wise to adopt a style where you use a consistent prefix (or suffix) such as `p`, `p_`, `ptr_` that flags this fact.

Some programmers like a convention that is really more the style of the Ada language than C++. In this convention, if the value of the argument is used but not changed by a function, then the argument name starts with `in_` (C++ pass by value or const reference). If the function ignores any initial value, but passes a result back (implies a C++ reference argument) then the argument's name should start with `out_`. If the function reads and changes the value of an argument then its name should start `inout_` (again, pass by reference).

After the first letter (or multi-letter prefix) the rest of name made up with words, if more than one word, capitalise first letter of each word – `gRecordCount`, `sTreeRoot`, `fXCoord`, (You can use underscore as an alternative to capitalization, e.g. `g_record_count`; but stick to one style in a program.)

Names should clarify role of variable in program. It is easier to understand code that involves manipulations of "gRecordCount" than code that manipulates "x".

There are some commonly used names: i, j, k, ndx are typically used as for-loop control variables; ch is often used as name of char variable whose scope is the current block or function. A variable called 'temp' had better be defined in a block with a scope of at most half a dozen lines!

Function Interface

Miscellaneous points:

- How many arguments?
Not many! Functions with multiple arguments (>3) are indicative of bad design.
In old style C programs, and procedural C++ programs, you may have to pass around several arguments because they represent related data. With C++ classes, a lot of related data are packaged together as the data members of an instance of the class. The member functions of that class don't have to have these passed as arguments – these data are already directly accessible!
- Default values specified in declarations?
Don't over do this; they can cause more trouble than benefit.
- Pass by value, pass by reference, or pass by pointer?
Value – for things like simple integers, doubles, or anything else ≤ 8 bytes.
const reference for things that you would have passed by value but which were larger than the 8 byte limit.
reference or pointer ? Can't be completely definitive, but here are some suggestions.

Remember a pointer and a reference are to all intents and purposes the same thing – the address of the argument. In most circumstances you could have either type as an argument. There is one difference, you can pass a NULL pointer, you can't pass a NULL reference. So, if a NULL argument is meaningful, the function must use pointer arguments.

If arguments are passed by reference, the code of the function is easier to read (you don't have all those pointer dereferencing steps):

Reference

```
void Swap(double& d1, double& d2)
{
    double temp = d1;
    d1 = d2;
    d2 = temp;
}
```

Pointer

```
void Swap(double* pd1, double *pd2)
{
    double temp = *d1;
    *d1 = *d2;
    *d2 = temp;
}
```

Most programmers find the reference style much more intelligible.

Those who favour a pointer style argue from the perspective of the calling code. They claim that the pointer style helps flag calls where arguments can be changed. The call will involve specifically passing an address rather than a value. So, if you see the address of a variable being passed to a function, you should suspect that it could get changed.

I generally favour use of pass by reference (the function prototype should use const qualifier to identify cases where the argument will not be changed).

Note. If your function is defined to take a reference, and you find you need to call a function that requires a pointer, that is easy – use the "address of" operator

```
void OtherFunction(Record* aRec);

void MyFunction(Record& theRecord)
{
    ...
    OtherFunction(&theRecord);
}
```

```
}
```

If your function is defined to take a pointer, and you find you need to call a function that requires a reference, that is easy – "dereference" the pointer:

```
void ThatFunction(Record& dataRec);

void FirstFunction(Record *p_Record)
{
    ...
    ThatFunction(*p_Record);
}
```

Functions that return error codes

There are at least four problems here – a) passing responsibility back to a calling function that can do nothing about the error, b) consistency of style, c) an unfortunate habit of programmers of returning something that could either be a result or an error flag, d) there is no way that you can make sure that someone who calls your function is going to check the error code you return.

Why are you returning an error code? You encountered some problem – have you any reason to expect that your client can do something about it? If the problem is essentially a fatal error (e.g. you tried to read data from a file and got garbage) the program will have to stop. So you might as well stop it with code that prints a message to cerr and calls exit().

Other examples of where you should just stop the program with an error exit would be a sort function being asked to sort -5 items, an empty stack being asked to Pop(), a list being asked for the 13th entry when it only has 12 items; obviously in all these cases the program has got its data corrupted (or is in some other way just plain buggy) and there is no point in continuing.

You may find it useful to define a function like:

```
void DieOnError(const char* msg) { cerr << msg << endl; exit(1); };
```

you set a breakpoint in this, then you can use the debugger to work back up the stack looking at calling sequence and data while you try to sort out why things went wrong.

If you really do encounter an error that the calling code could do something about, then it is appropriate to return an error indicator (e.g. you have been asked to load a particular record from a file and find that the record specified doesn't exist – most likely the user typed a wrong number and should be allowed another go).

There are several conventions:

- return 0 if all went well, return 1 if failed – possibly setting some global "errno" variable to a value that explains the specific failure mode (lots of system functions do this)
- return 0 if all went well, return a non-zero error number if function failed
- return 0 if all went well, return 1 if failure with an extra error variable (passed by reference) being set to a value that identifies the specific failure mode
- return 1 (true) if succeeded, 0 (false) if failed (opposite to first convention)

OK, you can't avoid these inconsistent styles. Just make sure that all the functions you write for a program adopt the same style.

Mixing error flags and results is a common practice (all those system functions that return NULL pointers or valid addresses are examples of practice). Again, try to minimize the extent that you do this. It is better to have a function that returns a success or failure indicator and sets a reference parameter to hold the result.

If you believe that the code that calls your function really should pay attention when you report an error, you had better learn how to use exceptions. (Put that off till next year.)

bool type

C++ has a boolean type `bool` with constants `true` and `false`. Most current compilers don't support this. Endless mess as you will find headers with `#define FALSE 0`, `#define TRUE ~0` and `typedef int boolean` (and all kinds of variations on this theme).

Unfortunately, for now you will just have to use `int` instead of `boolean`, following the convention that 0 is false, any non-zero value is true.

Assertions

Use assertions to validate data passed to principal functions.

Public member functions of a class, and any "exported" functions from a function style module, should check their data. Private member functions, and static functions within modules shouldn't need to repeat the checks (because you will have had to reach them via a public function that has done the checks) but you might have them while still debugging your own class.

You have to `#include <assert.h>`. Then you can have checks like the following:

```
void Sort(int data[], int low, int high)
{
    assert(low >= 0); // no negative subscripts please
    assert(high >= low); // I don't know how to sort upside down
    ...
}
```

NEVER include any effective code from a function within an assert statement. The following is wrong:

```
void Something(char *datastring)
{
    char buff[kMAX];
    int len;
    assert( (len = strlen(datastring)) < kMAX);
    ...
    for(int i=0; i < len; i++) ...
}
```

A compiler can be told to omit the assert statements! If there is effective code in an assert, the resulting optimized code is incorrect. In this example, variable `len` would not be initialized. The correct (and more natural) way of coding this example is:

```
int len = strlen(datastring);
assert( len < kMAX);
```

Functional decomposition

If you can point to a few lines of code and state its purpose, e.g. "this computes ...", "this opens the file if it exists already, otherwise it creates it", you've probably found a new function. Cut out that code, promote it to an auxiliary function (private member function if a class, static auxiliary function if a function style module).

Functions should be small.

Functions should do one thing.

Functions that take an argument and use it to select a subset of statements really should be a set of separate functions.

Be aggressive at decomposing functions into smaller auxiliary functions.

Case statements

Make sure that you have:

- constructed each clause correctly, with a "break" to prevent "fall through" into next clause (if you mean fall through to occur, this should be clearly commented in the code)
- have considered a default clause (possibly even put one in with no statements but simply a comment explaining why no default action needed).

If statements

Always double check that you don't need an else clause for an if.

That expression being tested in the if() – should it be promoted to a function? If the expression involves several && and || operators you really should be thinking about a boolean function.

If you have several clauses combined, make sure you are using the boolean && and || operators not the bitwise & and | operators.

Expressions

Use parentheses even if they are redundant. Most of us can't remember the rules of precedence of the operators in C++.

Don't combine autoincrement (autodecrement) operators into expressions (sure such usage is legal, sure all the great C hackers do it, it is also very error prone). A simple example:

```
while(sum < 50) { sum += data[i]; i++; }
```

rather than

```
while(sum < 50) { sum += data[i++]; }
```

Don't be smart alecky and mix arithmetic and boolean operations; only nerds enjoy code like

```
LongStringCount += strlen(stringdata[i++]) > 15;
```

Pointers

A pointer variable holds the address of another variable. In C++, almost always a pointer should hold the address of a dynamically allocated data structure that you created with the new operator. The only time that you should be using the & "address of" operator to get the address of something is when supplying an argument to the low-level read/write binary data transfer functions associated with input/output streams or, occasionally, when supplying an argument to one of the old C library functions that gets used from C++.

Pointers are a major cause of problems.

Pointers that point somewhere they shouldn't:

These result from careless coding. You have used an uninitialized pointer, or you've used a pointer despite the fact that its value is NULL (you should have checked), or you've used a pointer to a data structure that you once created with new but which you have since deleted.

On Macs and PCs, such misuse of pointers may pass unnoticed (your program's output is quite probably wrong, but you never check it that carefully do you). On Unix, the misuse is picked up at run time and your program is killed. The plaintive excuse "Well, it worked on my PC", is often heard by markers (who just laugh). Your program has a bug. Unix was nice enough to "point" this out.

Memory leaks:

You create a structure with `new` and, in an assignment statement, store its address in a pointer. You use it for a while. Then you forget about it. Your pointer goes out of scope or is set to point somewhere else. The data structure you created remains sitting in the heap like a kind of zombie (dead but still hanging around).

You have a memory leak. You also have a mark leak.

Pointer sillies

Markers often see code like the following:

```
char *ptr_char;
ptr_char = new char;
...
*ptr_char = 'a';
...
... = *ptr_char ...
...
delete ptr_char;
```

or worse:

```
char *ptr_char;
ptr_char = new char;
cin >> ptr_char; // read name
...
```

The first is stooooopid, but not incorrect. The programmer wanted a single character variable. Instead of just using `char ch` (and getting an automatic), this programmer arranges to have a pointer variable, sets this pointer to a newly created dynamic data structure intended to hold one character, and then laboriously works by dereferencing the pointer to get at the data. The overhead is immense. Operators `new` and `delete` aren't cheap. There is a storage allocation overhead; the data structure allocated in the heap is probably 12 or 16 bytes in size (not one). All the pointer dereferencing makes the code difficult to read.

The second example is wrong. It says create space for one character. Now read in an arbitrary length string. The characters in the string will destroy the "housekeeping information" associated with the dynamically created data structure (and other data structures nearby in the heap). The program will die in some horrid way shortly afterwards. The marker will wish a similar fate befall the programmer.

standard libraries

There are about 14 standard C libraries that are also used from C++. C++ also adds its own libraries. You need to know about a few of these. If you have a module that uses any of the functions from these libraries, you must `#include` the header file describing the corresponding library. (Caution. The Integrated Development Environments used on Macs and PCs often "help" you by effectively `#including` these headers without your having to ask. Unfortunately, your program then gets compilation errors when you transfer it to Unix.)

The "C" libraries have all been updated so that they work for both C and C++. You will probably use the libraries described in the following header files:

- `assert.h`
The `assert` "macro" that you use to check arguments.
- `ctype.h`
The "functions" like `isalpha(char)`, `isdigit(char)` used to check characters.
- `limits.h`
Mainly a set of constants defining things like the size of the largest short integer, long integer etc.
- `math.h`

Prototypes for standard mathematical functions for sine, cosine, logarithm etc. (Some versions also define lots of useful constants, e.g. PI; unfortunately, this isn't standardized.)

- `stddef.h`
 - `stdlib.h`
 - `string.h`
 - `time.h`
- Mainly typedefs. (Including a spare definition of NULL just in case you didn't include `stdlib.h`.)
- Always `#include` this. Odds and ends you need all the time (like the definition of NULL for work with pointers). Look at its contents sometime --- lots of useful functions like `atoi()` which converts a string of digits into an integer value, `rand()` which gives you a pseudo random number, `exit()` which allows your program to stop when all is lost, ...
- The functions like `strcpy(char*, const char*)` (copying a string), `strlen(const char*)`.
- Functions for measuring time, and functions for converting times and dates into strings.

You will probably only use a few of the C++ specific libraries. The ones that you use will be those providing input/output functions:

- `iostream.h`
The main streams classes, `cin`, `cout`, `cerr`.
- `fstream.h`
The extensions that allow you to read and write files.
- `strstream.h`
An extension that allows you to use an array of characters as if it were an input or output stream (sometimes useful for formatting messages).
- `iomanip.h`
A few add ons for the `iostream.h` stuff, provides the "manipulators" that help do formatting.

code layout

The integrated development environment that you use on a Mac or PC will attempt to arrange your code in a sensible manner.

Read your 111 notes or the textbooks for recommendations on layouts. There are different styles for indentation and placement of { "begin" and } "end" brackets. Just adopt one style and stick to it consistently in your code.

Functions

There are a number of function libraries that you will use; some of the more commonly used functions and their libraries are identified here.

Finding out more about a function (arguments, result type etc):

On Macintosh:

Use the "Think Reference Program". This has a set of "databases", select SLR (standard library reference) or IOStream (input output) ("Databases" button will show a selection menu). Program displays list of all functions, select the one you require.

On PC:

Use Help system, entering name of function. Hypertext style cross references can help you find useful related functions whose names you don't know.

On Unix:

Use the man (manual) command with name of function (bit patchy with iostreams, as although has details on all the functions only a few are listed in the man pages index).

Selection of more commonly used functions:

Function name	Library header	Use
abs	stdlib	absolute value (see also labs, fabs)
assert	assert	test value? terminate program
atoi	stdlib	convert string digits to integer
clock	time	time since program started
cos	math	cosine
ctime	time	time to date - time string
exit	stdlib	terminate program
fabs	math	absolute value of float
isalnum	ctype	is character letter or digit?
isalpha	ctype	is character letter?
isdigit	ctype	is character digit?
islower	ctype	is character lowercase?
isupper	ctype	is character uppercase?
memset	string	initialize block of memory (array)
memcpy	string	copy array
rand	stdlib	random number
sin	math	sine
sleep	"unix"	delay (<i>not standard, library varies</i>)
srand	stdlib	initialize random number generator
strcat	string	append string
strcmp	string	compare strings
strcpy	string	copy one string into space of other
strlen	string	length of string
strncpy	string	copy part of string
strtok	string	break string into "tokens"
time	time	get time
toupper	ctype	convert character to upper case
tolower	ctype	convert character to lower case

Stream i/o

OK, this does get moderately complicated with a complete hierarchy of related classes. The following descriptions are simplified, focussing just on those aspects that you are most likely to use.

There is a further problem. The standardisation process for C++ is still not finalized. Some changes relating to the streams library are proposed. If these are adopted, they will be introduced into compilers at different rates. Consequently, there will be inconsistencies between systems.

iostream

The `iostream` header defines the basic input stream, `istream`, and output stream, `ostream`, functionality and also the specialized standard streams `cin`, `cout`, and `cerr`.

An `istream` object can:

- "give to" any of the standard builtin data types:

```
in >> x; // 'x' can be int, long, double, character, or character string
          // or an instance of a class that you defined and for which you
          // have provided an associated istream& operator>>(...) function
```
- report its status
`in.eof()`, `in.good()`, `in.bad()`, `in.fail()`
- input a complete line (defined by maximum number of characters, or end of line marker)
`in.getline(...)`;
- skip over characters (limit number or end marker)
`in.ignore(...)`;
- read a character (unlike the `>>` operator), this version does not skip whitespace
`in.get(ch)`;
- read a block of bytes (use this only for record structured files)
`in.read(...)`;
- set or report byte position (only useful for record structured files)
`in.seekg(...)`;
`in.tellg()`

Similarly, an `ostream` object can:

- "take from" any of the standard builtin data types:

```
out << x; // 'x' can be int, long, double, character, or character string
          // or an instance of a class that you defined and for which you
          // have provided an associated ostream& operator<<(...) function
```
- report its status
`out.good()`, etc
- write a block of bytes (use this only for record structured files)
`out.write(...)`;
- set or report byte position (only useful for record structured files)
`out.seekp(...)`;
`out.tellp()`

An ifstream is an istream that you can attach to a file on which you can then do operations like open(), or close().

Similarly, an ofstream is an ostream that you can attach to a file.

An fstream is for files used for both input and output. It can do anything an ifstream can do as well as anything an ofstream can do.

A function that specifies that it wants to be passed an istream can be given an ifstream (or an fstream) once the file opening has been done.

Similarly, a function that uses an ostream can be passed an ofstream (or an fstream).

stringstream

The stringstream group allows you to use a character array in memory as a "stream". Most frequently used to convert a value to a string that you want to pass around as a string.

iomanip

The iomanip part of the library defines a few extra "manipulators" that can simplify formatting.