COMPUTER ENGINEERING I

Data Structures and Algorithms through C

UCD School of Electrical, Electronic and Communications Engineering

Course Reader Developed By:

Philip Curran, Brian Murray, Daragh McDonnell and Conor Heneghan

CONTENTS

1. INTRODUCTION

1.1 Programming Overview

A computer is a machine which is capable of producing solutions to problems in a repeatable fashion. Preparing a list of instructions telling the computer how to perform its calculations is called *programming.* Many programming languages exist, however all programs *must* be written in a language that can ultimately be converted to *machine language* (ML). Every computer has its own specific ML which depends on its architecture and in particular, on the type of processor it uses. A ML program consists of a sequence of binary (1's and 0's) *operation codes* which can be interpreted by the computer's hardware, in order to carry out specific tasks. Programming in ML is extremely laborious, and highly error-prone.

Assembly language represents the first level of abstraction from ML. Using assembly language, the programmer can write instructions by means of letters, instead of binary numbers. The letters designating a given operation are arranged in a mnemonic code that conveys the "sense" of the operation. An *assembler* then translates these mnemonics into the appropriate binary operation codes. To illustrate, a fragment of an assembly language program which adds the numbers 10 and 20, and stores the result in memory at address 620, might appear as follows:

```
LDA $10          ;Load value 10 into accumulator
ADD $20          ;Add 20 to the accumulator
STA #620         ;Store accumulator in address 620
```

While programming in assembly is much more convenient than in ML, it is still tedious, and since the mnemonics employed correspond directly to the machine's specific operation codes, assembly language programs written for one type of computer will not execute on any other.

High Level Languages (HLLs) represent a further level of abstraction from the machine level. HLLs are designed to be transportable. Thus, a program written in a HLL can operate on any number of different computers, *provided* that the computer is equipped with a program capable of translating the statements of the HLL into its own specific ML. Such a translation program is termed a *compiler*. Many HLLs exist, with popular examples being C, C++, Java, Python, and Ruby. 'C' is a general purpose language noted for its concision. In addition to portability, HLL's are far less tedious to work with than assembly language, reducing program development time (and cost!).

1.2 Generating the Executable

Before a program can run on a computer, it must first be translated to the appropriate ML. The final program, which is capable of running on the machine, is termed the *executable*. The steps required to create the executable from a program written in a HLL are:

1. **Writing the source code:** A HLL is comprised of a limited syntax and set of keywords. A program written in a HLL consists of the required sequence of statements to solve the problem in hand. A text editor is generally used to compose the program, which can then be saved in a text file. The program statements contained in the file are referred to as *source code*.
2. **Compiling the source code:** A compiler is a program which translates HLL programs into ML. During compilation, the compiler alerts the programmer to any unrecognised statements or syntax encountered in the source code. In the event of such compile-time errors, the programmer must amend the source code before the compiler will generate the ML, or *object* file.
3. **Linking the object file(s):** Often libraries of useful or common functions are available to the programmer, for use in his / her programs. The linker is responsible for combining the object code corresponding to these functions with the programmer's code, in order to produce the executable.
4. **Running the program:** The fact that a program compiles and links without any errors does not necessarily imply that the executable is error-free. Errors occurring during execution are termed *run-time errors*. This arises, for instance, when a perfectly legal statement is executed using erroneous data, as in the case of division by zero.

5. **Debugging the program:** Errors in a computer program are called *bugs*. *Debugging* is the process of eliminating bugs. This is typically an iterative process, which involves repeatedly modifying the source code, compiling, linking and executing, until (hopefully) all bugs have been identified and eliminated.

Figure 1.1 illustrates the program development cycle.

1.3 The C Programming Language

C is a general-purpose programming language, noted for its economy of expression, and its rich set of operators. It was developed by Dennis Ritchie at Bell Labs in 1972, and evolved (not surprisingly) from the B programming language.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition - the ANSI standard (or "ANSI C") - was completed in late 1988. Most modern C compilers provide support for ANSI C, allowing programmers to write programs which will compile and run on a wide variety of platforms with little or no modification. It is thus good programming practice to ensure that your C programs are fully ANSI C compliant. In this course, we shall be concerned solely with the ANSI definition of C.

1.5 A First C Program

To get a feeling for some of the concepts involved in writing a C program, we will now consider listing 1.1. The purpose of this (artificially) simple program is to print the words "Goodbye World!" on the computer screen. Some points to note are:

- C is a case-sensitive language. For instance, "`printf`" in listing 1.1 would be interpreted differently to "`Printf`".
- C programs are "free format". The layout of the code is largely at the discretion of the programmer. However, there are certain conventions that are adhered to by most C programmers, which improve the legibility of the code. See listing 1.2 for an example of how NOT to lay out a C program.
- Text appearing between the *comment* delimiters (`/* . . . */`) is ignored by the compiler. It is extremely good programming practice to use comments to clearly document the operation of your program.
- A C program consists of *functions* and *variables*. A function contains *statements* that specify the computing operations to be performed, and variables store values used during the computation. You are free to give functions whatever names you wish, with one caveat: Every program *must* contain one, and only one, function called "`main`". Your program begins execution with the first statement in the function "`main`". The statements of a function (called the function *body*) are enclosed in braces { }.
- A function may call other functions. Each function can accept an argument list from the calling function, which is enclosed in parentheses ( ) immediately after the function name. The arguments comprising the list must be separated by commas. A function may return one, and only one value to the calling function. In our example, the function `main` is called with no arguments, indicated by the empty brackets, while the function `printf` is called with a single argument, namely the character string "`Goodbye World!\n`".
- In C, many useful pre-written functions are supplied in libraries. The line
  "`#include <stdio.h>`"
  instructs the compiler to include information about the standard input/output library, which is part of the ANSI C standard, and of which the function `printf` is a member.

1.6 Program Design Language (PDL)

Complex programming projects can overwhelm the programmer, unless a formal approach to program design is adopted. One such approach is *functional decomposition*, which involves formulating the solution in stages, each stage representing a refinement of the previous stage. We start with the most

general functional view of what is to be done, then proceed by breaking that function into sub-functions, and so on, until all sub-functions are simple enough to be readily understood and implemented in code.

PDL facilitates the process of functional decomposition. PDL is a combination of plain English, and special keywords. To illustrate the use of PDL, consider the following example, which might represent part of a program to print out the contents of a text file:

```
read first character
WHILE this character is not the end-of-file marker DO
      print this character
      read next character
ENDWHILE
```

We will learn more about PDL as the course proceeds.

2. TYPES, OPERATORS AND EXPRESSIONS

The basic data items manipulated in a C program are *variables* and *constants*. Operators specify what operations can be performed on the data, such as addition or multiplication. An *expression* combines data and operators to produce new values.

2.1 Data Types

There are only four basic data types in C:

- `int` Integer values
- `char` Character values
- `float` Single precision floating point values
- `double` Double precision floating point values

As we will see later, the programmer can produce an unlimited number of new data types based on these four fundamental types.

2.2 Variable Declarations

A *variable* is an item of data whose value is permitted to change during the execution of a program. Before a variable can be used, the C compiler must know its type. *Declarations* list the variables to be used, state what type they have and optionally specify their initial value. The format of a variable declaration is

```
type-specifier list_of_variables;
```

Examples of variable declarations are

```
int hours, minutes, seconds;
char key = 'x';
double pi = 3.141592654;
```

The variables `key` and `pi` are initialised at the same time as they are declared, using the "=" sign. *Be warned!* Variables that are declared, but not explicitly initialised will contain random, "garbage" values. Using such a variable before it has been assigned a valid value will lead to unpredictable results.

2.3 Type Qualifiers

The attributes of a data type may be modified using a *qualifier*. The form of the variable declaration then becomes:

```
qualifier type-specifier list_of_variables;
```

The magnitude and / or precision of values which a particular data type can represent depends on the number of bits associated with the type. For instance, if an `int` is represented using 16 bits then the range of allowable values is -32768 to 32767. This may prove insufficient for some computations. In such cases, the qualifier `long` can be used to declare an integer of extended range. For example,

```
long int big_number = 123456;
```

Similarly, for integers of restricted range, the qualifier `short` may be employed, as in

```
short int little_number = 123;
```

which *may* produce memory and /or computational savings. Be aware however, that on some machines, `short int` and `int` may have the same size, while on others `int` and `long int` may have the

same size. The only guarantee provided by ANSI C is that `short int` and `int` must be at least 16 bits, and that `long int` must be at least 32 bits.

Similarly, for floating-point calculations requiring a high degree of precision, the qualifier `long` may be applied to `double` as in

```
long double pi;
```

Once again, depending on the machine, `float`, `double` and `long double` may be represented using one, two or three sizes.

Table 2.1 details the memory requirements and range of representable values associated with the various data types and qualifiers, for a typical C compiler.

Finally, the qualifier `const` is particularly useful when we wish to use symbolic names to refer to constants, as in the delcaration

```
const double pi = 3.141592654;
```

The `const` qualifier notifies the compiler that the value of `pi` in the program is not permitted to change. Any attempt to modify `pi` will thus be flagged as an error.

2.4 Constants

A constant is just a specific data value, such as 10, 3.14, "goodbye world", etc. C recognises a variety of constant types, some of which are described next.

- Integer constants can be specified in decimal, octal or hexadecimal form. Decimal constants must begin with one of the digits 1 to 9 (e.g. `123`), octal constants must have a leading zero (`0377`), while hexadecimal constants begin with `0x` (`0x1FF`). A `long` integer constant is denoted by a terminal `L` (`123456789L`). An integer too big to fit into an `int` will also be taken as `long`.

- Floating point constants contain a decimal point (`123.4`) or an exponent (`1E-2`); they are of type `double` unless suffixed with `F` (denoting `float`) or `L` (denoting `long double`).

- A character constant consists of a single character enclosed in single quotes (`'x'`). Characters which cannot be easily typed from the keyboard can be accessed via the escape character `\` (backslash). Thus, the newline character is represented as `'\n'`, while the backslash character itself can be obtained as `'\\'`. Table 2.2 lists the complete set of escape sequences.

- A string constant is a sequence of zero or more characters surrounded by double quotes, as in `"I am a string"`, or `""` (the empty string). Internally, the compiler appends the special null character `'\0'` to a string constant, to mark the end of the string, so the physical storage required is one more character than the number of characters appearing between the quotes.

Armed with our knowledge of variables and constants, we will next discover how they may be combined with operators to produce expressions.

2.5 The Arithmetic Operators

There are five binary arithmetic operators (operators which work on two values at a time). They are `+`, `-` ,`*` , `/` and `%`, denoting addition, subtraction, multiplication, division and modulus respectively. Special attention is required with integer division. The expression `a/b` yields the integer part of `a` divided by `b`, any fractional part being discarded. The expression `a%b` ("`a` mod `b`") yields the remainder when `a` is divided by `b`. We illustrate the integer division and modulus operations by examples:

`13 % 5` evaluates to 3.

`13 / 5` evaluates to 2.

In addition to these binary operators, there is the unary negation operator − (minus), which gives the arithmetic negative of its single operand.

2.6 Type Conversion and the Cast Operator

A question arises, when an operator has operands of differing data types, as to how the types are combined to produce the result. The general rule in C is that the operand of the "lower" type is promoted to the "higher" type, the result of the operation being of the "higher" type. The "ranking" of the types, from highest to lowest is `long double`, `double`, `float`, `long int`, `int`, `short`, `char`.

Operands of type `char` are always promoted to `int` when appearing in expressions. The value of a `char` is given by the numeric value of the character in the machine's character set. For instance, in the ASCII character set, the character `'A'` is represented by the value 65. Thus, the following two expressions are equivalent:

```
char key = 'A';
char key = 65;
```

The above discussion describes instances of *automatic*, or *implicit* type conversion. You will often find it necessary to use *explicit* type conversion. Explicit type conversions can be forced by means of the *cast* operator, which simply consists of the required data type enclosed in parentheses. Thus, in the expression

```
(double) area * (double)height
```

both `area` and `height` are converted to type `double` prior to multiplication.

2.7 Increment and Decrement Operators

C provides two unary operators for incrementing and decrementing variables. The operator ++ adds the value 1 to the operand, while the operator −− subtracts 1. Both may be employed as prefix (preceeding the operand), or postfix (following the operand) operators, as in the following expressions:

`x = count++;` Increments `count` *after* its value has been assigned to the variable `x`.

`x = ++count;` Increments `count` *before* its value has been assigned to the variable `x`.

Suppose that `count` has the value 5. Then in the first expression, `x` is assigned the value 5, whereas in the second `x` is set to 6. In both cases the value of `count` becomes 6.

2.8 Assignment Operators

The assignment operator (=) is used to store values in variables. The simplest example of an expression involving the assignment operator is

```
variable = expression;
```

in which the expression is first evaluated, the resulting value being assigned to the variable (after conversion to the appropriate type, if necessary). Next consider the following expression:

```
i = i + 2;
```

In C, this can be written more succinctly by means of the *compound assignment* operator +=, i.e.

```
i += 2;
```

Similarly, we have compound assignment operators for the remaining binary arithmetic operators denoted by `-=`, `*=`, `/=` and `%=` respectively. Let `op` be a binary operator. Then, if `expr1` and `expr2` are expressions,

```
expr1 op= expr2;
```

is equivalent to

```
expr1 = (expr1) op (expr2);
```

Note carefully the parentheses around `expr2`. For example,

```
x *= y + 1;
```

means

```
x = x * (y + 1);
```

rather than

```
x = x * y + 1;
```

2.9 Precedence and Order of Evaluation

The precedence of an operator is the priority it is accorded by the compiler, when evaluating expressions. By way of example, the `*` operator has higher precedence than the `+` operator, and so the expression

```
x = 5 + 2 * 7;
```

evaluates to 19, and not 49. If we want to perform the addition prior to the multiplication, we have to make use of parentheses:

```
x = (5 + 2) * 7;
```

This time, the expression evaluates to 49. Expressions containing operators of the same precedence are evaluated according to the associativity of the operators involved. For example, the operators `*` and `/` have the same precedence, and associate left to right, so the expression

```
3 * 5 / 2
```

is evaluated by first computing `3 * 5` and then dividing the result by two to yield `7`, whereas

```
5 / 2 * 3
```

is computed by first determining `5 / 2` and then multiplying by `3` to give `6`. Obviously care should be taken when writing expressions. You are strongly adivsed to use redundant parentheses, wherever confusion might arise. Thus we may rewrite

```
3 * 5 / 2
```

as

```
(3 * 5) / 2
```

in order to clearly signal our intentions.

Table 2.3 lists the precedence and associativity of operators in ANSI C (some of which we have not yet encountered). You don't have to memorise this table, but you should know how to use it!

2.10 Bitwise Operations

C provides the ability for programmers to manipulate variables at a bit level. Bit level refers to the binary representation of a C variable. For instance, the character 'A' is 0100 0001 (decimal 65 – hexadecimal 41), the character 'B' is 0100 0010, etc. The `int 45` is represented as 00000 0000 0010 1101 etc.. An advantage of the ability to manipulate individual bit patterns is in interfacing with hardware. Typically, a physical interface with a computer system may consist of a number of wires. The voltage son the wires will represent logical 1s and 0s, and by directly reading in the voltages, the physical interface can be converted into the representations used internally in the computer. This for example is how the physical interface from the keyboard is implemented.

C provides six operators for bitwise operations. For most high level programs, it is not necessary to make use of these operations, and indeed may not be recommended due to difference between computers as to their internal representations (e.g., some computers may define a `short int` to be two bytes – others may only provide one byte). However for low level programming in so-called embedded systems (i.e., intelligent processors inside electronic equipment such as fridges, automatic brake steering, digital combination locks, etc.) then bitwise operators are wisely used. The six operators are given in Table 2.10.1.

Table 2.10.1: The bitwise operators

Operator	**Description**
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
~	Bitwise ones complement
<<	Left shift
>>	Right shift

2.10.1 The bitwise AND operator

This follows the rules of the logical AND operation. These rules are given by the following truth table:

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

A bitwise operator applied to two variables will carry out the same operation for all bits in both variables. For example, consider the following code

```
char x='A',y='B',z;
z=x&y;
```

'A' has the bit pattern 0100 0001
'B' has the bit pattern 0100 0010
'A' & 'B' has the bit pattern 0100 0000 (which is the char @).

The AND operator is useful for forcing certain bits to 0 since anything that is ANDed with 0 is automatically set to 0. For instance, if we read in 8 unknown bits (represented by X) and AND them with the pattern 0000 1111, we get the following result.
XXXX XXXX & 0000 1111 = 0000 XXXX
In other words, the four least significant bits (rightmost bits) are retained, and the top four are set to zero. This is called bit masking.

2.10.2 The bitwise OR operator

This follows the rules for the logical (inclusive) OR operation. These rules are given by the following truth table:

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

A bitwise operator applied to two variables will carry out the same operation for all bits in both variables. For example, consider the following code:

```
char x='A',y='B',z;
z=x|y;
```

'A' has the bit pattern 0100 0001
'B' has the bit pattern 0100 0010
'A' | 'B' has the bit pattern 0100 0011 (which is the char 'C').

The OR operator is useful for forcing certain bits to 1 since anything that is ORed with 1 is automatically set to 1. For instance, if we read in 8 unknown bits (represented by X) and OR them with the pattern 0000 1111, we get the following result.
XXXX XXXX & 0000 1111 = XXXX 1111
In other words, the four most significant bits (leftmost bits) are retained as they are and the bottom four are set to one.

2.10.3 The bitwise exclusive OR operator

This follows the rules for the exclusive OR operation. These rules are given by the following truth table:

Input 1	**Input 2**	**Output**
0	0	0
0	1	1
1	0	1
1	1	0

The XOR bitwise operator applied to two variables will carry out the same operation for all bits in both variables. For example, consider the following code:

```
char x='A',y='B',z;
z=x^y;
```

'A' has the bit pattern 0100 0001
'B' has the bit pattern 0100 0010
'A' ^ 'B' has the bit pattern 0000 0011 (which is the char '~').

2.10.4 The bitwise ones complement operator

This follows the rules for the logical complement operation. This is a unary operator with the following truth table:

Input	Output
0	1
	0

The ~ bitwise operator applied to two variables will carry out the same operation for all bits in both variables. For example, consider the following code:

```
char x='A',z;
z=~x;
```

'A' has the bit pattern 0100 0001
'~A' has the bit pattern 1011 1110 (which is the char '¾').

2.10.5 The bitwise shift operators

The left shift operator << causes the bits in the operand to be shifted to the left by a given amount. For example, the statement

```
op1=op1<<4;
```

causes the bits in `op1` to be shifted 4 bits to the left, filling the vacated bits with a zero bit. For example, if op1 is an `int` represented by

```
op1=1111 1100 1100 1101
```

then

```
op1<<4  is 1100 1100 1101 0000
```

If the leading bits of a number are zero, then a left shift corresponds to multiplication by 2 (in the same way that adding a zero to a decimal number corresponds to multiplication by 10). Shift operators are also used for parallel-to-serial conversion (*i.e.*, the communication to a printer is often serial in nature, *i.e.* a single bit is on the wire at a time, and is generated by shifting patterns one by one on to the wire).

The right shift operator >> causes the bits in the operand to be shifted to the right. The leftmost bits are filled in with zeros if the data type is unsigned (e.g., `char` or `unsigned int`). If the data type is signed then usually the existing sign bit is replicated (an arithmetic right shift)

```
int op1=-12;
op1=op1>>4;
```

The binary representation of –12 in signed magnitude is 1000 0000 0000 1100. After the right shift, we will have 1111 1000 0000 0000 which is –30720.

For all the bitwise operators, if the two operands have different bit widths, the smaller one is converted to the longer word by left padding with zeros, i.e.,
0011 0011 0000 1111 & 0001 0001 becomes
0011 0011 0000 1111 & 0000 0000 0001 0001
by left padding of the second operand with zeros.

3. BASIC INPUT AND OUTPUT

Before we start writing simple C programs, we need to know how to supply input data to a program, and how to display the output data it produces. The C language itself has no facilities for handling input / output (I/O) operations. However, ANSI C defines a standard library of functions for performing common tasks, including I/O. We tell the compiler that we intend to use these I/O functions by placing the directive `#include <stdio.h>` in our source code, conventionally at or near the beginning of the file. The file `stdio.h` contains all the information the compiler needs to access the standard I/O functions.

When we execute a C program, three files are automatically opened for the program's use - the *standard input*, the *standard output*, and the *standard error*. The C library provides functions for reading data from the standard input, and writing data to the standard output. Any error messages produced by the program are written to the standard error. On an interactive system, the standard input is the keyboard, while the standard output and standard error correspond to the screen.

For the moment we will look at just four of the many I/O functions provided in the standard library: `printf`, `putchar`, `scanf` and `getchar`.

3.1 The printf Function

The `printf` function writes data to the standard output, and has the following form:

```
printf(format_string, arg1, arg2, . . .)
```

The format string specifies how the list of arguments `arg1`, `arg2` etc. should be formatted. The arguments can be variables or expressions. The format string consists of ordinary text, which is written directly to the output, and one *conversion specifier* for each argument in the list. We clarify the use of `printf` by means of examples.

In the simplest case, the argument list is empty and the format string contains no conversion specifiers, allowing us to write simple text strings to the output, as in

```
printf("Goodbye World!\n");
```

Note the `\n` (newline) escape sequence which moves the cursor to the beginning of a new line.

Integer variables, or expressions can be written to the output as follows:

```
int a = 2, b = 3;
printf("The sum of %d and %d is %d.\n", a, b, a+b);
```

This would cause the following output to appear on the screen:

```
The sum of 2 and 3 is 5.
```

`%d` is an example of a conversion specifier. It informs the compiler that, during program execution, a decimal integer value obtained from the argument list is to be inserted in the output stream at the position of the conversion specifier. Each conversion specifier consumes one argument, in the order they appear in the list. Note that the number of conversion specifiers and the number of arguments *must* correspond, otherwise, upon execution, the behaviour of `printf` will be unpredictable. This will also be the case if any of the conversion specifications is malformed. A conversion specifier must begin with a `%`, and end with a valid conversion character. We may further refine the format of the output by placing special characters in-between. For instance, the minimal conversion specifier for a floating point number is `%f`, and so the statements

```
const float pi = 3.14159;
printf("[%f]\n", pi);
```

would produce the output

```
[3.141590]
```

on the screen. By default `%f` displays six decimal places, however we may choose to refine the output, as in the following examples:

```
printf("[%.4f]\n", pi);        /* Display 4 decimal places */
printf("[%7.4f]\n", pi);       /* Right justify, field width 7 */
printf("[%-7.4f]\n", pi);      /* Left justify, field width 7 */
printf("[%e]\n", pi);          /* Use scientific notation */
```

The output in each case is:

```
[3.1416]
[ 3.1416]
[3.1416 ]
[3.141590E+000]
```

We won't dwell any further on the `printf` function here. You will find a full description in the appendices of your textbook.

3.2 The scanf Function

The `scanf` function reads data from the standard input, and has the following form:

```
scanf(format_string, arg1, arg2, . . .)
```

Essentially `scanf` is `printf` in reverse: This time, conversion specifiers in the format string are used to match data from the input stream and store it in the arguments of the argument list. For example, the code to read an integer value from the standard input, and assign it to the variable `x`, is:

```
int x;
scanf("%d", &x);
```

Note *very* carefully the `&` preceding the variable name `x`. `scanf` stores the results of its conversions directly in the computer's memory. The unary `&` operator provides the memory address of its operand. Thus `&x` is the address in memory where the value of the variable `x` is stored, and not the value of `x` itself. In addition to conversion specifiers, we can use text in the format string to control the format of the input. Consider the following example, which reads in a date in the form dd/mm/yy:

```
int day, month, year;
scanf("%d/%d/%d", &day, &month, &year);
```

Here, the input must contain three integers separated by forward slashes, otherwise `scanf` will fail to make the required assignments.

`scanf` ignores blanks in its format string. Furthermore, it skips over white space (blanks, tabs, etc.) as it looks for input values. Thus, the code

```
int here, there, everywhere;
scanf("%d %d %d", &here, &there, &everywhere);
```

with the input

```
10                20
      30
```

results in the assignments `here = 10`, `there = 20`, `everywhere = 30`.

3.3 getchar and putchar

The simplest I/O task is to read or write a single character of data. Although `scanf` and `printf` can be used for this purpose, it is the equivalent of using a sledgehammer to crack a nut, primarily due to the overhead incurred in processing the format string. The standard library provides two "functions", `getchar` and `putchar` to implement this task efficiently.

`getchar` reads a single character from the standard input, as in the following example:

```
char key;
key = getchar();
```

Analagously, `putchar` writes a single character to the standard output:

```
char key = 'a';
putchar(key);
```

4. FLOW OF CONTROL

A C program consists of a collection of statements which, unless otherwise specified, are executed sequentially, from the first statement to the last. Formally, a statement is any line terminated by a semicolon (;). Often, we want to alter the sequence in which statements are executed. For instance, we may need to repeat a particular group of statements a number of times. Or, we may want to select which set of statements to execute, depending on whether or not some pre-specified condition is met. C provides two mechanisms for altering the sequential flow of control, namely *loop constructs* (allowing repetitive execution) and *decision constructs* (allowing selective execution). In either case, we must specify the conditions under which we wish the flow of control to be altered. We do this by means of relational and logical operators.

4.1 The Relational and Logical Operators

The relational operators are

```
<   <=   >   >=
```

which denote less than, less than or equal to, greater than and greater than or equal to respectively. The two equality operators are:

```
==   !=
```

denoting equal to, and not equal to respectively. In C, if a relational expression is true, it evaluates to 1, if it is false it evaluates to 0. Thus, if we have

```
int i = 5, j = 6;
```

then the expression

```
i < j
```

is true and evaluates to 1, whereas

```
i == j
```

is false, and evaluates to 0. Note carefully the difference between the relational operator (`==`) and the assignment operator (`=`). In the above expression, `i` is compared to `j` to test if their values are equal, without changing the values of either, but the line

```
i = j;
```

assigns the value of `j` to `i`.

C provides three logical operators:

```
&&   ||   !
```

denoting logical AND, logical OR and logical NOT respectively. If a logical expression is true it evaluates to 1, if it is false it evaluates to 0. The expression

```
condition1 && condtion2
```

is true if and only if `condition1` is true AND `condition2` is true. The expression

```
condition1 || condition2
```

is true if either of `condition1` OR `condition2` is true.

The NOT operator inverts the truth of an expression. Thus, if we have

```
int valid = 4;
```

Then the expression

```
!valid
```

evaluates to zero. (This illustrates an important point: in general in C, an expression is true if it evaluates to any non-zero value. Only expressions which evaluate to 0 are false).

4.2 Decisions: The Conditional Expression

The *conditional expression* provides a convenient means of choosing which of two expressions to evaluate, and has the form:

```
expr1 ? expr2 : expr3
```

`expr1` is evaluated first. If it is non-zero (true) then the expression `expr2` is evaluated and that is the value of the conditional expression, otherwise `expr3` is evaluated, and that is the value. For example, we could use a conditional expression to determine the greater of two numbers as follows:

```
z = (a > b) ? a : b;
```

4.3 Decisions: if - else

The `if-else` construct is used to make decisions, and has the general form

```
if(expression)
      statement
else if(expression)
      statement
else
      statement
```

The expressions are evaluated in order, from top to bottom. If any expression is true, the statement associated with it is evaluated, and this terminates the whole chain. Program execution resumes at the first statement following the `if-else` block. The final `else` statement handles the default case, where none of the conditions are satisfied. Note that the `else if` and `else` branches of the block are optional.

Although the C language allows us to associate only one statement with each branch of the `if-else` block, this statement may be a *compound statement*. A compound statement consists of a group of individual statements enclosed in braces, as shown below:

```
{
statement1;
statement2;
      .
      .
last_statement;
}
```

The following examples illustrate the use of `if-else`:

Ex. 1: A single `if` statement:

```
if(i % 2 == 0)
```

```
        printf("%d is an even number\n", i);
```

Ex. 2: An `if-else` block:

```
if(a > b)
      printf("%d is greater than %d\n", a, b);
else if(a < b)
      printf("%d is less than %d\n", a, b);
else
      printf("%d is equal to %d\n", a, b);
```

Ex. 3: `if-else` and compound statements:

```
if(age < 25 && sex == 'm')
      {
      car_insurance += 2000;
      printf("Your premium is EUR %d\n", car_insurance);
      }
```

Listing 4.1 provides a complete program illustrating the use of `if-else`.

4.4 Decisions: switch

The `switch` statement provides an alternative to the `if-else` chain for situations where we want to compare an integer expression to a number of constant values, and has the form

```
switch(expression){
      case value_1:
            statements
      case value_2:
            statements
            .
            .
      case value_n:
            statements
      default:
            statements
}
```

The keyword `case` is used to label individual values that are compared with `expression`. If a match is found, execution begins with the statement immediately following the match. Consider the following example:

```
switch(position){
      case 1:
            printf("You finished first.\n");
            break;
      case 2:
            printf("You finished second.\n");
            break;
      case 3:
            printf("You finished third.\n");
            break;
      default:
            printf("Better luck next time.\n");
}
```

The purpose of the `break` statement is to forcibly exit the `switch` block, and thus to prevent "fall-through". (Say, in the above example, the value of `position` is 2. Then, the statements corresponding to `case 2` will be executed. But, if we omit the `break` statement, execution "falls through" to the following cases, and all statements corresponding to these cases will also be executed. The `break` statement causes execution to resume at the first statement following the entire `switch` block, allowing us to execute only those statements corresponding to the `case` that was matched). The `default` branch comes into operation only when none of the specified `case`s are matched. Listing 4.2 provides a further illustration of the `switch` construct.

4.5 Loops: while

A `while` loop is used to repeatedly execute a statement, and has the form

```
while(expression)
      statement
```

The `expression` is evaluated. If it is non-zero (true), then `statement` is executed, and expression is re-evaluated. The cycle continues until `expression` becomes zero (false). Beware of the obvious pitfall: If `expression` fails to become zero, the cycle continues indefinitely, and we are stuck in an infinite loop. A fragment of code to print out the numbers from 1 to 20 might be as follows:

```
int number = 0;
while(number++ < 20)
      printf("%d\n", number);
```

Again, we can use a compound statement if we need to execute more than one statement per iteration. For example, suppose we want to echo characters typed on the keyboard to the screen, until the user inputs an exclamation mark:

```
char key;
key = getchar();
while(key != '!'){
      putchar(key);
      key = getchar();
}
```

As an illustration of the terseness that is possible with C, note that we could have written the above code as

```
char key;
while((key = getchar()) != '!')
      putchar(key);
```

Listing 4.3 provides a complete program illustrating the use of `while`.

4.7 Loops: do-while

The termination condition for a `while` loop appears at the top of the loop. If this condition proves false the first time it is tested, the statements in the loop will not be executed. The `do-while` construct ensures that the statements within the loop are executed at least once, by placing the termination condition at the end. `do-while` has the following form:

```
do
      statement
while (expression);
```

The use of `do-while` is illustrated in listing 4.4. In practice, it tends to be used less often than `while`.

4.7 Loops: for

The syntax of a `for` loop is

```
for(expr1; expr2; expr3)
      statement
```

The `for` loop is equivalent to:

```
expr1;
while(expr2){
      statement
      expr3;
}
```

Most commonly, `expr1` and `expr3` are assignments and `expr2` is a relational expression. To illustrate, we might use a `for` loop to print only the even numbers from 0 to 20 inclusive as follows:

```
int i;
for(i = 0; i <= 20; i += 2)
      printf("%d\n", i);
```

We execute a group of statements in our `for` loop using a compound statement. Listing 4.5 provides a further example of the use of the `for` statement.

4.8 Infinite loops, break and continue

If necessary, we can forcibly exit a loop by means of the `break` statement. For instance, we may want to carry out some operation indefinitely, until a specific "break-out" condition obtains. This can be achieved using `break` with an infinite `for` loop:

```
for(;;){
      scanf("%c", &key);
      if(key == '!') break;
      printf("%c ", key);
}
```

The above code echoes characters read from the keyboard to the screen. The loop is executed indefinitely unless an exclamation mark is typed, in which case the `break` statement forces an exit from the loop. Execution resumes at the line immediately following the loop. Note how we implement the infinite loop by using empty expressions in the `for` statement. Similarly, an infinite loop using `while` can be obtained as follows:

```
while(1){
      statement
}
```

The `continue` statement is used to interrupt the current iteration of a loop, by immediately returning control to the top of the loop. This is useful if we don't want to process all statements of a loop during every iteration. Consider the following code, which reads in an integer, and calculates its square root:

```
for(;;){
      scanf("%lf", &x);
      if(x == 0) break;
      if(x < 0) continue;
      printf("The square root of x is: %lf", sqrt(x));
}
```

Here, the `continue` statement guards against attempts to compute the square root of a negative number.

5. FUNCTIONS

A C program consists of one or more functions. Every C program must contain exactly one function called `main`. Program execution begins with the first statement in `main`. Functions are used to simplify the coding of complex programming tasks. We decompose our original task into a number of (relatively) simple tasks. We then implement each of these simple tasks using a separate function. A common task may be implemented as a re-useable function, which can then be used in any software project that requires this task. Thus, programmers can build on the work of others, without having to "re-invent the wheel".

In C, all functions have the following general form:

```
return-type function_name(formal_argument_list)
{
      declarations
      statements
}
```

Only limited communication between functions is possible. Generally, a function passes data into a function it invokes, by means of the invoked function's argument list. The invoked function passes data to the invoking function by means of a *return value*.

5.1 The Argument List

We execute the statements in a function by making a function call. When we call the function, we pass data (termed the function *arguments*) to it by means of the function's formal argument list. This list determines the type of data the function expects to receive, and assigns identifiers to it. For example, we might code a function to print the square of a number as follows:

```
void print_square(double x)
{
      printf("The square of %lf is %lf", x, x * x);

      return;
}
```

The argument list indicates that the function `print_square` expects to receive a double precision floating point number when called. This data item is identified in the function body by the name `x`. Thus, we may use `x` in our function exactly as if we had declared a variable of that name, and assigned it the value passed by the calling function. We invoke a function by typing its name, followed by a parenthesised list of arguments whose types match the corresponding entries in the function's formal argument list. For example, we might invoke `print_square` from another function as follows:

```
      double value = 2.27;
      print_square(value);
```

Here, we have supplied the variable `value` as the single argument to `print_square`. We can also use constants or expressions as function arguments, as in the following:

```
      print_square(2.27);
```

or

```
      print_square(value * 2.27);
```

We can specify as many entries in the argument list as we like, so long as we remember to invoke the function with the correct number of arguments and each argument is of the correct type, as in

```
void many_args(int x, float y, char z)
{
      .
      .
      .
}
```

We might invoke `many_args` as follows:

```
int value1;
float value2;
char value3;
      .
      .
many_args(value1, value2, value3);
```

If a function doesn't need any data to be passed to it, you should state this explicitly by replacing the argument list by the keyword `void`, as in

```
int no_args(void)
{
      .
      .
      .
}
```

5.2 Return Values

A function can return one, and only one, value to the calling function. The return-type precedes the function name, and specifies the data type of the value to be returned. For instance, say we wanted to calculate the square of a number, and use it in subsequent calculations, rather than printing its value as previously. We might write a function `square` as follows:

```
double square(double x)
{
      return (x * x);
}
```

Here, the return value is of type `double`. We inform the compiler of the value we want returned by means of the `return` statement. We highlight the value to be returned by placing it in brackets. If our function doesn't need to return a value, we should make this explicit by specifying a return type of `void`, as in

```
void print_char(char key)
{
      printf("%c\n", key);

      return;
}
```

Formally, we still use a `return` statement to indicate the exit point from the function. However, for functions which return `void`, it is permissible to omit the return statement, the function automatically exiting when it encounters the closing brace. We can use the return value of a function in an expression, as if it were a variable, as in

```
double y, z, value = 4.7;

y = square(value);
z = y + square(3.7);
```

5.3 Declaring Functions - Prototypes

Before we attempt to call a function, the compiler needs to know the arguments the function accepts, and the type it returns. We specify this information by means of a function declaration, or *prototype*. For example, the prototype for our function `square` is

```
double square(double);
```

This tells the compiler that we intend to use a function `square` which accepts a single argument of type `double` and returns a value of type `double`. There are two common choices in C programming for the location of a function prototype. First, we can prototype our function in any other function which calls our function, e.g.

```
int main(void)
{
double y;
double square(double);
      .
      .
y = square(2.5);
      .
      .
return (0);
}
```

With this approach, we need to prototype `square` in any function which uses it. Alternatively, we can make the function `square` available to *all* functions in our source file, by prototyping it outside of, and before any other function in the file, as in

```
#include <stdio.h>

double square(double);

int main(void)
{
      double y;

      y = square(2.5);
      .
      .
}
```

Note that, with either alternative, the order of functions in the source file is unimportant.

5.4 Automatic Variables

You should be aware that a variable declared inside a function comes into existence only when that function is called, and is destroyed when the function returns. For this reason, such variables are termed *automatic* or *local* variables. Furthermore, when you supply a variable as an argument to a function, you are passing a *copy* of the variable's value into the function, and not the variable itself. Consider the following example:

```
void func1(int);

int main(void)
{
      int x = 2;

      func1(x);
      printf("%d\n", x);
      .
      .
}
```

```
void func1(int x)
{
      x *= 2;
      printf("%d\n", x);
}
```

The output from `func1` is 4, and from `main` is 2. `x` in `func1` is local to `func1`, and is unrelated to `x` in `main`. In practice, we often choose names in function argument lists to match variable names in the calling function, simply because it can be difficult to think up new variable names. However, you should remember that the two variables are completely independent of each other.

Listing 5.1 illustrates all of the concepts relating to functions we have described above.

6. THE C PREPROCESSOR

The C preprocessor is a mechanism which makes modifications to your source code, before it is processed by the compiler. Three modifications are possible: Macro substitution, file inclusion and conditional compilation. An instruction to the preprocessor is called a *directive*. Every directive must begin with the hash (#) character, followed immediately (no spaces) by the directive type.

6.1 Macro Substitution

Macro substitution allows us to replace a token in our source code by a text string. Substitutions are specified using the `#define` directive. The general form of a macro is

```
#define token replacement text
```

For example, the directive

```
#define PI 3.14159
```

instructs the preprocessor that, anywhere the token `PI` is encountered in the source file, it is to be replaced with the text `3.14159`. This is useful for assigning meaningful names to constants used in your source code. The replacement text is arbitrary - you can associate any text with the token, so long as it constitutes valid C source code, as in the following example:

```
#define FOREVER for(;;)
```

Now, when we want to implement an infinite loop in our source code, we can use the token `FOREVER`, in place of the somewhat more cryptic `for(;;)`, as in

```
FOREVER{
      if((key = getchar()) == 'q') break;
      putchar(key);
}
```

We can also define macros which take arguments, thus allowing the replacement text to vary for different calls of the macro. The general form of such a macro is

```
#define macro_name(arg1, arg2, ...) replacement text
```

For example, consider the following macro, which produces source code to determine the greater of two numbers:

```
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

If, in our source code, we invoke this macro as

```
x = MAX(p + q, r + s);
```

the preprocessor will replace the formal argument `A` with the text `p + q`, and `B` with `r + s`, giving

```
x = ((p + q) > (r + s) ? (p + q) : (r + s));
```

We place each of the arguments in the macro definition in parentheses in order to eliminate any ambiguity in the order of evaluation. Consider the following, erroneous macro which is intended to produce code to calculate the square of a number:

```
#define SQUARE(x) x * x
```

Then, the code

```
y = SQUARE(x + 1);
```

will be expanded as

```
y = x + 1 * x + 1;
```

which is equivalent to $y = x + (1 * x) + 1$ instead of the intended $y = (x + 1) * (x + 1)$. The correct macro definition in this case would be

```
#define SQUARE(x)  ((x) * (x))
```

Although a macro with arguments resembles a function call, it serves a completely unrelated purpose. A macro is used *only* to modify the text of your source code, prior to compilation.

6.2 File Inclusion

We can include the contents of another file in our source file by means of the `#include` directive. Any source line of the form

```
#include "filename"
```

or

```
#include <filename>
```

will result in the contents of `filename` being inserted in our file, at the position of the `#include` directive. (In practice, you will never actually *see* the included file in your code). If `filename` appears between quotes as in the first form, then the preprocessor will search in the same directory as

the source file for the file to be included. This form is most often used to include files you have written yourself. The second form searches in a set of standard locations for the file to be included. This form is typically used to include system files. When you want to access functions in the standard library, you need to include the appropriate *header* file, as in

```
#include <stdio.h>
```

Header files contain macros and function prototypes which the compiler needs before it can access the functions in the library. By convention, the names of header files end with ".h". Table 6.1 lists the standard header files needed to access the various functions in the standard library.

6.3 Conditional Compilation

Conditional compilation allows us to selectively incorporate or omit lines of source code before compilation. Typically, we achieve this by a combination of the `#define` and `#ifdef` (if defined) directives, as in the following:

```
#define IDENTIFIER
      .
lines_1
#ifdef IDENTIFIER
      lines_2
#endif
lines_3
```

In this case, the lines of code between the `#ifdef-#endif` directives will be compiled only if the token `IDENTIFIER` has been defined earlier in the file. Otherwise, only the code corresponding to `lines_1` and `lines_3` will be compiled. This can be useful for program development, where we might want to generate extra information during the debugging phase. An example of this scenario is provided in listing 6.1. In addition to `#ifdef`, we have `#ifndef` (if not defined), which can be useful for determining whether a token has been previously defined, as in

```
#ifndef DEBUG
      #define DEBUG
#endif
```

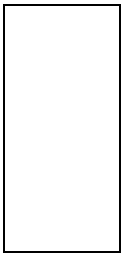
We will encounter conditional compilation again later, when we deal with program organisation.

7. ARRAYS, POINTERS AND STRINGS

An array is a collection of related data of the same type. A pointer is a variable whose value is the address in memory of another variable. In C, there is an intimate relationship between arrays and pointers but, as we shall see, the pointer is a more fundamental concept than the array. A string is special type of one one-dimensional character array.

7.1 Arrays

An array allows us to conveniently manipulate a group of related data items which have the same type. Instead of assigning each member of the group its own variable name, we assign a name to the group as a whole, and use a special *index* or *subscript* notation to access the individual members of the group. As an analogy, consider a one dimensional vector, with *n* elements, of the form



Here, the vector as a whole is identified by the name , and the individual elements are identified by the vector name, plus a subscript ranging from *1* to *n*. The general form of the declaration of a 1-D array in C is

```
type-specifier array_name[number of elements];
```

For instance, the declarations for two arrays, `x` and `y`, containing 10 integers and 15 floats respectively, would be:

```
int x[10];
float y[15];
```

The size of an array *must* be specified by a constant value: We cannot use a variable expression to set the array size. Thus, the following style of declaration is illegal:

```
int size = 5;
int array[size];
```

There is one *crucial* difference between the vector notation we saw above, and our array notation: For an array with *n* elements, the array subscripts range from zero to *n-1*, and not *1* to *n* as with the vector. In order to access an individual element of the array, we use the array name, with the index of the element enclosed in square brackets. For example, `x[4]` is the *fifth* element of the array `x` above, `x[0]` is the first element and `x[9]` is the tenth and *last* element. However, C doesn't perform bounds checking on an array, so the following statement is perfectly legal:

```
x[10] = 0;
```

In this case, memory has been set aside for 10 elements, but we attempt to assign a value to a non-existent eleventh element, for which storage has not been provided. This results in an attempt to store data in memory which has been allocated for some other purpose, which can produce highly undesirable results.

We can access all the elements of an array using a loop, as in

```
#define SIZEX 10
```

```
      .
      .
int i, x[SIZEX];
      .
for(i = 0; i < SIZEX; i++)
      printf("%d\n", x[i]);
```

7.2 Initialising 1-D Arrays

We can initialise the elements of a 1-D array when we declare it, by enclosing a comma-separated list of initialisers in braces, as in

```
int x[5] = {0, 1, 2, 3, 4};
```

Note that the intialisers must be constant expressions. For arrays declared within functions, if too few initialisers are specified, the values of the remaining array elements will be undefined, e.g.

```
int x[4] = {0, 1} /* x[2] and x[3] undefined */
```

Too many initialisers will produce a compilation error. For large arrays, it may be more convenient to perform initialisation by means of a loop, as in

```
#define SIZEX 1000
      .
      .
int i, x[SIZEX];

for(i = 0; i < SIZEX; i++)
      x[i] = 0;   /*Initialise all elements to zero */
```

7.3 Multidimensional Arrays

We are not limited to 1-D arrays. In particular, we will often need to use a 2-D, rectangular array. A 2-D array is analogous to a matrix in mathematics. The general form of a declaration for a 2-D array is

```
type-specifier array_name[rows][columns];
```

For example, the declaration of a 2-D array of floats named `A`, with 3 rows, and 3 columns, is:

```
float A[3][3];
```

We can think of `A` as being a [ ] matrix, with the following form:

```
A[0][0]     A[0][1]     A[0][2]
A[1][0]     A[1][1]     A[1][2]
A[2][0]     A[2][1]     A[2][2]
```

i.e., for an array with *m* rows and *n* columns, the row and column subscripts range from zero to *m-1*, and zero to *n-1* respectively.

We can access the all the elements of the array using nested `for` loops, as in

```
#define MAXROWS 3
#define MAXCOLS 3
      .
      .
int i, j;
float table[MAXROWS][MAXCOLS];
      .
      .
for(i = 0; i < MAXROWS; i++)
```

```
        for(j = 0; j < MAXCOLS; j++)
              printf("%f\n", table[i][j];
```

We can initialise a 2-D array by enclosing the initialisers for a given row in braces, then enclosing all the row initialisers in braces, as follows:

```
float table[3][3] = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8} }
```

7.4 Arrays as Function Arguments

A special notation is used to specify arrays as function arguments. The function prototype for a function `func1` which accepts an array of integers as input might be as follows:

```
void func1(int []);
```

The empty square brackets tell the compiler that the argument is an array. (For a 1-D array, there is no need to specify the array size). The function definition for `func1` might then be as follows:

```
void func1(int table[])
{
      int i;
      for(i = 0; i < MAXSIZE; i++)
            printf("%d\n", table[i]);
}
```

In `func1`, the array of integers passed in is identified by the name `table`. We can then access the elements of the array using the usual array notation. We could invoke `func1` from another function as follows:

```
int list[MAXSIZE];
      .
      .
func1(list);
```

i.e. in the function call, we need only specify the array name in order to pass the array to the function.

We can specify 2-D arrays as function arguments in the following manner:

Function prototype:

```
void func2(int [][COLS]);
```

Function definition:

```
void func2(int table[][COLS])
{
      statements;
}
```

Function call:

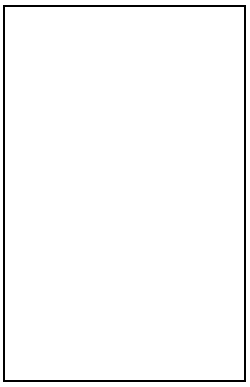
```
int matrix[ROWS][COLS];
      .
      .
func2(matrix);
```

Note that, for multidimensional arrays, we may omit the first dimension, but must specify all subsequent dimensions in the function prototype and definition.

It is important to note that, when we pass an array to a function, any modifications made to elements of the array in the called function will affect the array elements in the calling function. We will explain why this is so in our treatment of pointers.

7.5 Pointers

A pointer is a variable containing the address in memory of another variable. Typically, a computer's memory consists of a contiguous arrangement of cells. Each cell is assigned a number, called its address, which is used to access its contents. We can visualise this arrangement as shown below:



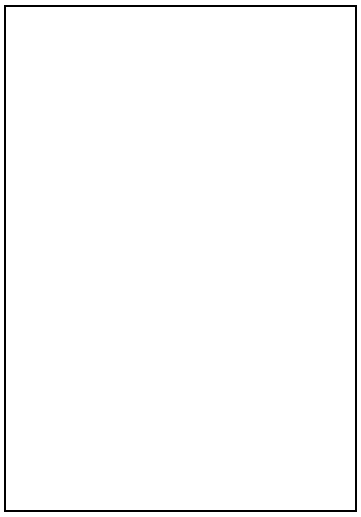
Now, suppose we declare a variable `x` as follows:

```
int x = 123;
```

If we have a pointer named `ptr`, we could use it to store the address in memory where `x` is stored as follows:

```
ptr = &x;
```

We have seen the unary address operator before: `&x` evaluates to the address in memory of `x`, which we then assign to `ptr`. Now `ptr` is said to "point" to `x`. This situation can be depicted as follows:



`x` is stored at address 1020. Since `ptr` points to `x`, it contains the address of `x`, i.e. 1020. We can use `ptr` to access the value of `x` indirectly, by means of the *indirection* or *dereferencing* operator, `*`. For example, we could assign the value of `x` to an integer variable `y` using `ptr` as follows:

```
y = *ptr;
```

The value of `ptr` is the address 1020. The `*` operator fetches the data stored at that address, (i.e. the value of `x`), so `y` is assigned the value 123.

Like any other variable, a pointer must be declared. We *must* inform the compiler of the data type to which we require it to point. The general form of a pointer declaration is

```
type_specifier *pointer_name;
```

For instance, we declare a pointer `ptr` which will hold the address in memory of an integer variable as follows:

```
int *ptr;
```

This notation is for consistency: It says that, if we access the contents of the memory address stored in `ptr`, we expect the data there to be of type `int`. The lines below illustrate the basics of pointers:

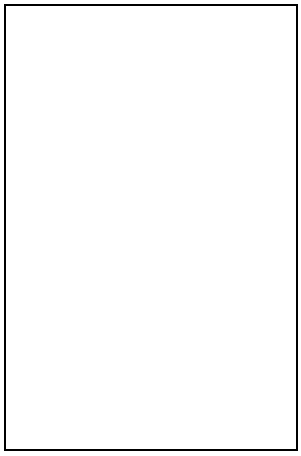
```
int x = 1, y = 2;
int *ip;    /* ip is a pointer to int */

ip = &x;    /* ip now points to x */
y = *ip;    /* y is now 1 */
*ip = 0;    /* x is now 0 */
ip = &y;    /* ip now points to y */
*ip += 2;   /* y is now 3 */
```

7.6 Pointers and Arrays

We are now ready to investigate the relationship between pointers and arrays. First, we consider how the computer represents an array in its memory. Assuming each memory cell can hold one integer value, the declaration

```
int x[5];
```

causes 5 consecutive memory cells to be set aside for the array `x`, as shown below:



Let us assign a pointer to the first element of the array `x`:

```
int x[5], *ptr, y;
ptr = &x[0];
```

Now we can use `ptr` to access the elements of the array. For instance:

```
y = *ptr;   /* y is set to x[0] */
```

To access the third element of `x`, we could use

```
y = *(ptr + 2);
```

Here, we add an *offset* of 2 to the address pointed to by `ptr`, so `*(ptr + 2)` accesses the memory address corresponding to the third element of `x` (i.e. `x[2]`). In general, different data types will require

different amounts of storage. For instance, a `float` may require 4 cells. This need not concern us, since the compiler automatically scales the any offset we add to a pointer by the number of memory cells required to store the data type corresponding to that pointer.

Our array notation is merely a programming convenience. In fact, before performing any operations on an array, the compiler converts the array notation to pointer notation. Thus, we can use the "array / index" and "pointer / offset" notations interchangeably, as illustrated in the following:

```
float x[3] = {0.1, 0.2, 0.3}, *fptr;

fptr = &x[0];
*(fptr + 1) = 0.4;       /* x[1] is now 0.4 */
fptr[1] = 0.5;           /* x[1] is now 0.5 */
x[1] = 0.6;        /* x[1] is now 0.6 */
*(x + 1) = 0.7;    /* x[1] is now 0.7 */
```

In C, the array name by itself is a synonym for the address of the first element of the array, so if we have

```
char a[SIZEA], *cptr;
```

then the following are equivalent:

```
cptr = &a[0];
cptr = a;
```

Although there is a close correspondence between arrays and pointers, they are *not* the same. The crucial difference is this: When memory is allocated to an array, the array name is a *constant* pointer to the start of the array, i.e., the address to which it points may not be changed. Hence, the following is illegal:

```
int x[SIZEX], y;
x = &y;      /*illegal */
```

However, since a pointer is a variable, we can have

```
int x[SIZEX], *ip;

ip = x; /* ip points to x[0] */
ip++; /* ip points to x[1] */;
ip[0] = 1;  /* x[1] is now 1 */
```

7.7 Pointers as Function Arguments

Previously, we noted that the only means by which a function may modify the value of a variable in the calling function is via its (single) return value. There are situations however, where it would be convenient for the called function to be able to change the value of more than one variable in the calling function. This we can achieve by using pointers. In general, we specify a pointer in an argument list as follows:

```
return_type function(type_specifier *identifier, ...)
```

To illustrate, consider the following function, which swaps the values of two variables, `x` and `y`:

```
void swap(int *x, int *y)
{
      int temp;

      temp = *y;
      *y = *x;
      *x = *temp;
```

```
}
```

The argument list indicates that `swap` expects to receive two pointers to integers, `x` and `y`. We might then invoke `swap` as

```
int a = 1, b = 2;
swap(&a, &b);
```

We supply the two pointer arguments expected by `swap` by passing in the addresses of the integer variables, `a` and `b`, as `&a` and `&b`, which are assigned the identifiers `x` and `y` respectively. In `swap`, we can alter the *contents* of these addresses (and not the addresses themselves!) by indirectly accessing them using the indirection (`*`) operator. Thus, after the call to `swap`, the values of `a` and `b` are 2, and 1 respectively. The function prototype for the function `swap` is

```
void swap(int *, int *);
```

which indicates that `swap` expects to receive two pointers to integers.

Finally, note that when we pass an array into a function, we are in fact passing a pointer to the first element of the array. Thus, the following function definitions are equivalent:

```
void init(int x[])
{
      int i;

      for(i = 0; i < SIZEX; i++)
            x[i] = 0;
}

void init(int *x)
{
      int i;

      for(i = 0; i < SIZEX; i++)
            x[i] = 0;
}
```

In either case, the changes we make to the elements of `x` in `init` will persist when the function returns, since we are modifying the contents of the memory pointed to by `x`, and not `x` itself. We can *prevent* a function from modifying the contents of an array or pointer passed into it by using the `const` qualifier, as in:

```
void print_array(const int table[])
{
/*    This function cannot modify the elements of the
      array table */
}

void print_value(const int *x)
{
/*    This function cannot modify the contents of the
      address pointed to by x */
}
```

In general, use of the `const` qualifier in an argument list is good practice, when our function doesn't need to modify the value of an argument passed into it.

Listings 7.1 and 7.2 illustrate the use of arrays and pointers.

7.8 Strings

A string is a character array, whose last element is the null character '\0'. The terminating null character enables the C compiler to identify where the string ends. A string constant is a text string enclosed by double quotes, as in

```
"I am a string";
```

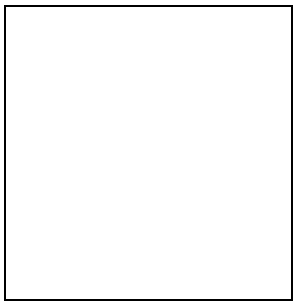
We may declare and initialise a string in either of two ways:

```
char s_array[] = "Hello";
```

or

```
char *s_ptr = "Hello";
```

When we initialise a string in this way, the compiler automatically determines the amount of storage required, and appends a '\0' to the end of the string. In each case, the string is stored in memory as shown below:



Each of the declarations allows the string to be accessed by providing a pointer to the first character of the string in memory. The array declaration provides a constant pointer, so the address to which it points cannot be changed. With the pointer declaration however, we could have the following:

```
char *s_ptr = "Hello";
printf("%s\n", s_ptr);
s_ptr++;
printf("%s\n", s_ptr);
```

The output from this code would be

```
Hello
ello
```

Note that the `printf` format conversion specifier for a string is `%s`. We can read a string from the standard input using `scanf` and `%s` as follows:

```
char txt[80];

printf("Input a string:  ");
scanf("%s", txt);
```

In this example we declare a character array, `txt`, which we will use to hold our string. We must ensure that we allocate enough storage for any string we intend to assign to our array, *including* the null terminating character. Note that here, we don't need to use the `&` operator with `scanf`, since the array name `txt` is itself a pointer to an address in memory, as required by `scanf`.

Some manipulations on strings are shown in listing 7.3.

7.9 String-Handling Functions - string.h

Since a string is just a special character array, any operations, such as copying one string into another, or concatenating two strings, must be performed on a character by character basis. Fortunately, the standard library provides convenient string-handling functions, which can be accessed via the header file `string.h`. Some of the more important functions are described below:

We may compare two strings using the function `strcmp`, whose prototype is

```
int strcmp(const char *s1, const char *s2)
```

This function compares the two strings pointed to by `s1` and `s2`, and returns an integer value less than, greater than, or equal to zero, if `s1` is respectively less than, greater than, or equal to `s2`, in the alphabetic sense.

We can determine the length of a string `s` using the `strlen` function, whose prototype is

```
int strlen(const char *s)
```

The function returns an integer, whose value is the number of characters in the string pointed to by `s`, excluding the null character.

We can copy the string pointed to by `s2` into the string pointed to by `s1` using the `strcpy` function:

```
char *strcpy(char *s1, const char *s2)
```

The entire contents of `s2` are copied into `s1`, including the null character. The function returns a pointer to the start of the first string, `s1`. You must ensure that the storage allocated to `s1` is sufficient to hold the string pointed to by `s2`.

We can concatenate (join) two strings, `s1` and `s2`, using the `strcat` function:

```
char *strcat(char *s1, const char *s2)
```

The null character, and subsequent characters of `s1` are overwritten with the characters of `s2`. Sufficient storage should be allocated to `s1` to hold the combined strings `s1` and `s2`. The function returns a pointer to the first character of the combined string, `s1`.

The use of these functions is illustrated in listing 7.4.

7.9 Other String Functions

The standard library contains three functions, `atoi`, `atol`, and `atof`, which convert an ASCII string to an `int`, `long int` or `double` respectively. They are useful when you need to extract numeric data from input read in as a string. They are accessed via `stdlib.h`. Their prototypes are

```
int atoi(const char *s)
long int atol(const char *s)
double atof(const char *s)
```

Each function takes a pointer to a string as an argument, and returns the corresponding numeric value of the string. Obviously, the string must consist of characters corresponding to the digits 0 to 9, with an optional decimal point in the case of `atof`.

The functions `sprintf` and `sscanf` are accessible via `stdio.h`, and have the prototypes

```
int sprintf(char *buffer, const char *format, arg_list)
int sscanf(char *buffer, const char *format, arg_list)
```

They operate similarly to `printf` and `scanf`, except that instead of writing to the standard output and reading from the standard input, they write to, and read from, the string pointed to by `buffer`. The use of these functions is illustrated in listing 7.5.

8. USER-DEFINED TYPES

The four fundamental data types in C are `int`, `char`, `float` and `double`. C also provides a mechanism for creating any number of new data type names. It must be emphasised that we cannot create a new data type - we may merely add a new name for an existing data type. New type names are created via the `typedef` keyword.

8.1 Using typedef

The `typedef` mechanism serves three primary purposes, allowing us to

- create convenient synonyms for complex declarations
- give existing data types more meaningful names in the context of our program
- easily change the type of all variables in our program that are used to represent a particular quantity

To illustrate, consider a route-planning program, which calculates the shortest road distance between an origin and a destination. The programmer may decide to represent all distances by variables of type `unsigned short int`. Thus, to declare a variable `dist`, which is used to store a distance, we would have

```
unsigned short int dist;
```

We could simplify this unwieldy declaration by creating a new data type, `Distance`, as follows:

```
typedef unsigned short int Distance;
```

Now, the new type name `Distance` is a synonym for `unsigned short int`, so we could rewrite our declaration of `dist` above as

```
Distance dist;
```

`Distance` is *not* a new data type: The variable `dist` exhibits the behaviour associated with variables of type `unsigned short int`. Note that, not only have we simplified our declaration - the type name is now more meaningful in the context of our program: The purpose for which variables of type `Distance` are intended will be readily apparent from inspection of our code.

Next, suppose the programmer realises that the type `unsigned short int` is inadequate for representing the longest distances arising in the route-planning program, and that all variables representing distances must be re-declared as `unsigned long int`. If `typedef` hadn't been used, s/he would have to scour the code for all variable declarations of `unsigned short int`, decide whether those variables represented distances, or some other quantity, and make any necessary replacements. A `typedef` simplifies this arduous process greatly, since all that needs to be done is to change the data type corresponding to the `typedef` statement. Thus, if we have

```
typedef unsigned short int Distance;
```

and we need to alter the type of all variables representing distances, we need only modify our `typedef` as follows:

```
typedef unsigned long int Distance;
```

8.2 Enumerated Types

When a variable can assume only a limited set of values, we can explicitly enumerate the permissible values using the `enum` declaration. For instance, a variable, `days`, used to represent the days of the week could be declared as follows:

```
enum {SUN, MON, TUE, WED, THU, FRI, SAT} days;
```

The variable `days` may only take the values of the enumeration constants (`SUN`, `MON`, etc.) appearing between the braces. Very often, `typedef` is used in conjunction with `enum` to simplify the declaration. In the above example, we could have

```
typedef enum {SUN, MON, TUE, WED, THU, FRI, SAT} Days;
```

Now we can declare the enumerated variable `days` as

```
Days days;
```

The use of enumerated types is recommended when dealing with quantities such as days of the week, or months of the year, in order to improve program clarity.

Listing 8.1 illustrates the use of the `typedef` and `enum` keywords.

9. STRUCTURES

A structure is a group of one or more variables, possibly of different types, grouped together under a single name, for convenient handling. (In Pascal, structures are called "records"). The general form of a structure declaration is

```
struct tag {
      member_declarations
};
```

e.g., we might declare a structure to represent a complex number as follows:

```
struct complex {
      double real;
      double imag;
      };
```

The above declaration reserves no storage - it merely defines a template for a new type "`complex`". We declare a variable with the type `complex` as follows:

```
struct complex z1;
```

This declaration causes storage to be set aside for a structure `z1` of type `complex`. The tag provides a convenient shorthand for declaring structured variables. It is common practice to use `typedef` to further simplify the declaration of structured variables, e.g. the statement

```
typedef struct complex Complex;
```

creates a new type name, `Complex`, which we could use to declare a variable of type `struct complex` as follows:

```
Complex z1;
```

The variables declared within the structure template are termed *members*. Thus, the structure `complex` has two members, `real` and `imag`, of type `double`. Note that the member variables of a structure may be of mixed types, as in

```
struct grades{
      char *student_name;
      int maths;
      int physics;
      float average;
};
```

We access the members of a structure using a special "dot" notation, the general form of which is

```
structure_name.member_name
```

Thus, to set the real and imaginary parts of `z1`, we might have:

```
z1.real = 4.6;
z1.imag = 2.5;
```

9.1 Pointers to Structures

As with any other type, we can declare a pointer to a structured variable. For example, to declare a pointer to a structure of type `Complex`, we could have

```
Complex *z_ptr;
```

We could set `z_ptr` to point to a variable `z`, of type `Complex`, using the address operator as follows:

```
Complex *z_ptr, z;

z_ptr = &z;
```

This causes `z_ptr` to be assigned the starting address in memory where `z` is stored. C provides a special operator, "`->`" for accessing the structure members indirectly via a pointer, the general form of which is:

```
structure_pointer->member;
```

e.g., to modify the real and imaginary parts of `z` above, via the pointer `z_ptr`, we could have

```
z_ptr->real = 3.5;
z_ptr->imag = 2.5;
```

Pointers to structures are particularly useful when we wish to specify structures as function arguments.

9.2 Structures as Function Arguments

Structured variables may be specified as function arguments and return values. We pass a structure into a function in either of two ways: First, we can pass in a copy of the structure, as illustrated below:

```
struct date{
      int day;
      int month;
      int year;
};

typedef struct date Date;

void print_date(Date);
```

```
int main(void)
{
      Date today;

      today.day = 25;
      today.month = 12;
      today.year = 96;

      print_date(today);

      return(0);
}

void print_date(Date d)
{
      printf("The date is %d/%d/%d\n",
                  d.day, d.month, d.year);
}
```

Instead of passing in a copy of the entire structure, it can be more efficient to pass in a pointer to the structure (which also allows the called function to modify the structure members in the calling function). We could add a function to the above program to set the date, as follows:

```
void set_date(Date *d)
{
      printf("Enter new date as dd/mm/yy:\n");
      scanf("%d/%d/%d", &d->day, &d->month, &d->year);
}
```

We could invoke this function from `main` as:

```
int main(void)
{
      Date today;
            .
            .
      set_date(&today);
            .
            .
      return(0);
}
```

Finally, to illustrate how we can return a structure from a function, we could have the following alternative to the function `set_date` above:

```
Date set_date(void)
{
      Date new;

      printf("Enter new date as dd/mm/yy:\n");
      scanf("%d/%d/%d", &new.day, &new.month, &new.year);
      return (new);
}
```

We might invoke this function from `main` as:

```
Date today;
      .
      .
today = set_date();
```

Listings 9.1 and 9.2 provide more ambitious examples of the use of structures.

10. FILE HANDLING

Thus far, our programs have required the user to enter all i/p data via the keyboard, and o/p data has been displayed the screen. We achieved this using two special files - the standard i/p and o/p respectively - which are automatically "connected" to the program when it is executed. In practice, this situation is unrealistic: more generally, programs work with large data sets, and are required to read i/p from, and write o/p to, files residing on a permanent storage medium, such as the computer's hard disk.

Conceptually, a file consists of basic units called records. Each record contains one, or more fields. For instance, we might define a record to hold details of a student's exam results. The fields in the record might then be the student name, and the grades obtained in each subject.

In C the data in files may be accessed sequentially, or randomly. In sequentially organised files, the records are stored in some logical order, and are intended to be accessed in that order, whereas in a randomly organised file, the records are not in any logical order, and may be accessed out of sequence by the program. We will be concerned only with sequential access.

A file may be stored in one of two ways - in either text, or binary format. A text file consists of a sequence of ASCII characters, so data is represented as it would be written on a page. A binary file consists entirely of 1's and 0's, with all data being stored in the form in which C represents them internally. We will consider text files only.

10.1 Opening and Closing Files.

File handling functions in C are accessed via `stdio.h`. Before a file can be accessed it must be opened. In C, when a file is opened, a special data structure with the type name `FILE` is created, which contains operating system specific information about the file. We need not concern ourselves with the details of this structure - we need only view it as a stream, which connects our program to the file, allowing data to flow in either direction. A file is opened using the function `fopen`, whose prototype is

```
FILE *fopen(char *filename, char *mode)
```

`filename` is a string containing the name of the file to be opened. `mode` is a string specifying how we intend to use the file. The two most common modes are `"r"` and `"w"`, denoting read-only and write-only respectively. If a file is opened as read-only, that file must already exist. If a file opened as write-only doesn't exist, it is automatically created. `fopen` returns a pointer to a structure of type `FILE`, which we must supply as an argument to any function which we use to access the file. To illustrate, we might open a file called "`input.dat`", for reading only, as follows

```
FILE *fpin;

if((fpin = fopen("input.dat", "r")) == NULL){
      printf("Cannot open file\n");
      exit(EXIT_FAILURE);
}
```

We declare a pointer, `fpin`, to a structure of type `FILE`, to store the address of the `FILE` structure returned by `fopen`. The opened file may be accessed only via `fpin`. `fopen` returns the special null pointer value `NULL` (pointer to nothing), when it is unable to open the specified file. The library function `exit` (`stdlib.h`) is used to immediately terminate the program, when some condition is encountered which renders further execution of the program meaningless. The exit code `EXIT_FAILURE` (`stdlib.h`) is used to notify the process which invoked the program that the program has terminated abnormally.

When we are finished using a file, it should be explicitly closed, since generally, the operating system limits the number of files that can be simultaneously open. This we achieve using the function `fclose`:

```
int fclose(FILE *)
```

e.g., to close the file pointed to by `fpin` above, we would have

```
fclose(fpin);
```

10.2 Character I/O

At the most basic level, we can read a single character from an open file, using the function `fgetc`:

```
int fgetc(FILE *)
```

`fgetc` fetches the next character from the file, and returns it as an `int`. If `fgetc` encounters an error, or reaches the end of the file, it returns the special value, `EOF`. When this value is returned, the programmer doesn't know whether it was due to an error, or the end of the file being reached. We may resolve this ambiguity using the function `feof`

```
int feof(FILE *)
```

which returns a non-zero value if the end of the file has been reached, and zero otherwise. Similarly, the function `fputc`

```
int fputc(int, FILE *)
```

is used to write a single character to a file, returning the value of the character just written. The use of `fgetc`, `fputc`, `EOF` and `feof` is illustrated in listing 10.1.

10.3 String I/O

The function `fgets` is used to read an entire string from a file.

```
char *fgets(char *, int, FILE *)
```

The first argument is a pointer to a character array. The second argument is the dimension of this array. `fgets` continues to read characters into the array until a newline character is encountered, an error arises, or all but the last element of the array has been written. A '\0' is then appended to the end of the characters read. If no errors are detected, `fgets` returns a pointer to the start of the array, otherwise it returns the null pointer.

The function `fputs` writes a string to a file.

```
int fputs(char *, FILE *)
```

The first argument is a pointer to an array containing the string to be written. `fputs` returns a nonnegative value if it is successful, and `EOF` on an error.

The use of `fputs` and `fgets` is illustrated in listing 10.2.

10.4 Formatted I/O

The functions `fscanf` and `fprintf` may be used to perform formatted input and output. They are in every respect identical to `scanf` and `printf`, with the exception that they operate on a stream specified by a `FILE` pointer.

```
int fscanf(FILE *, const char *format, arg_list)
int fprintf(FILE *, const char *format, arg_list)
```

`fscanf` returns the number of successful assignments made, and `EOF` if an error occurs, or the end of the file is reached. `fprintf` returns the number of characters written, or on an error, a negative value. Listing 10.3 illustrates the use of these functions.

10.5 Standard File Pointers

When a C program executes the standard i/p, o/p and error files are automatically opened. As with any other file, each of these files has an associated `FILE` pointer (`stdin`, `stdout` and `stderr` respectively) via which all communication between program and file must take place. Functions which read and write these files (`scanf`, `printf` etc.), do so via their corresponding file pointer, though, for convenience, the detail is hidden from the programmer. In fact, we could use `fprintf` or `fscanf` to access the standard o/p and i/p by specifying the corresponding file pointer, as in

```
fprintf(stdout, "Print to the standard output\n");
```

We have yet to use the standard error stream, to which any error messages the programmer wants to print out are normally sent, as in the following example:

```
fprintf(stderr, "Error: couldn't open input file\n");
```

Henceforth, we will adopt this policy.

11 ADVANCED TOPICS ON POINTERS

11.1 Dynamic Storage Allocation

We have seen previously that, when we wish to set aside storage for an array, we must specify the fixed dimensions of the array when we declare it. This can be extremely inefficient. For example, we might have a program to manipulate vectors ranging in size from 1 to 1000 elements. Instead of declaring arrays of 1000 elements, to cater for all possible vector sizes, it would be preferable if we could request just enough storage to hold the vectors we actually need to process. Dynamic memory allocation allows us to request variable amounts of storage during execution, and to access this storage indirectly via pointers.

11.1.1 The sizeof Operator

In order to set aside a sufficient number of memory cells to store the type of data we are interested in, we need to know how many cells it takes to store a variable of that type. This we can achieve using the `sizeof` operator. For instance,

```
sizeof(int)
```

evaluates to the number of cells required to store an integer. We can specify any type name as an operand to `sizeof`, including types created using `typedef`.

11.1.2 malloc and calloc

The function `malloc` is used to allocate memory during program execution. It takes one argument - the required number of memory cells - and returns a pointer to the start of the memory block allocated. `malloc` is prototyped in `stdlib.h`:

```
void *malloc( size_t number )
```

`size_t` ("size type") is a special type name for the result returned by `sizeof`. `number` is the number of cells we want to allocate. `malloc` returns a "pointer to void" (`void *`), which the programmer *must* explicitly cast to a pointer to the type of data the allocated block is intended to store. For example, to request storage for 20 integers, we might have

```
int *vect;

if((vect = (int *)malloc(20 * sizeof(int))) == NULL){
      fprintf(stderr, "\nFailed to allocate memory.");
      exit(EXIT_FAILURE);
}
```

In this case, the pointer returned by `malloc` is cast to a pointer to `int`. If `malloc` is unable to allocate the required memory, it returns the null pointer. Note how we use the `sizeof` operator to ensure that the correct number of cells for 20 integers is allocated. We can now treat the block of memory pointed to by `vect` as an array of 20 `int`s, using the familiar array notation.

The function `calloc` performs a similar role to `malloc`.

```
void *calloc( size_t, size_t )
```

`calloc` takes two arguments, of type `size_t`. The first specifies the number of elements to be allocated, the second the number of cells required to store the type. For example, to allocate storage for an array of 30 `float`s, we would have

```
float *table;
```

```
if((table = (float *)calloc(30, sizeof(float))) == NULL){
      fprintf(stderr, "\nFailed to allocate memory.");
      exit(EXIT_FAILURE);
}
```

The primary difference between `malloc` and `calloc` is that `calloc` initialises the allocated memory to zero, whereas data in memory allocated with `malloc` must be initialised by the programmer, before it is used.

11.1.3 free

When storage allocated by `malloc`, or `calloc`, is no longer required, it should be explicitly released using the function `free`:

```
void free(void *)
```

The argument is a pointer to the start of the block of memory which is to be released. For example, to release the storage allocated to the pointer `table` above, we would have

```
free(table);
```

Dynamic storage allocation is illustrated in listings 11.1 and 11.2.

11.2 Command Line Arguments

Typically, the executable file produced by the compiler from your source code can be invoked from the operating system's command prompt by typing its name. For example, in MS-DOS, a source file called `test.c` would be compiled and linked to a file called `test.exe`, which can then be run by typing the name `test.exe`. Often, it is convenient to provide data to the program on the command line, when it is run. For example, if we wanted to be able to specify the i/p and o/p files to be used by the program, then we could type

```
test.exe infile.dat outfile.dat
```

The program name, along with additional data on the command line, are termed *command line arguments (CLA's)*. In C, CLA's are passed into our program via the function `main`. If we wish to use CLA's with our program, then `main` must have the following argument list:

```
return_type main(int argc, char *argv[])
```

When the program is executed, the number of CLA's entered on the command line (including the program name itself) is passed into the variable `argc`. We use `argc` to check that the correct number of CLA's have been typed. The notation

```
char *argv[]
```

requires some explanation: `char *` indicates a pointer to a string. `argv[]` indicates that we are dealing with an array of such pointers. Thus, each element of `argv` is a pointer to a string, while the array name `argv` is a pointer to the first element of the array. The CLA's passed into the program are stored in memory as character strings. Each element of `argv[]` then points to the first character of the corresponding CLA. For example, if we typed

```
test.exe infile.dat outfile.dat
```

at the command prompt then, in our program, we would have `argc = 3`, `argv[0] = "test.exe"` (i.e. `argv[0]` contains a pointer the first character of the string "test.exe"), `argv[1] = "infile.dat"`, and `argv[2] = "outfile.dat"`.

Note that is it not compulsory to use the identifiers `argc` and `argv`, but this choice is an almost universal convention.

Listing 11.3 illustrates the use of CLA's.

11.3 Pointers to Functions

When a program is run, the executable file is first loaded from the hard disk into the computer's memory. Thus, associated with every function in the program, we have an address in memory, where the executable code corresponding to that function is located. In C, the function name by itself evaluates to the address of the function. For example, if we had

```
int square(const int);
```

then the name `square` evaluates to the address where the executable code corresponding to `square` resides. Thus, we may view `square` as a fixed pointer to a function which takes a single argument of type `const int`, and returns a result of type `int`. C allows us to declare a variable pointer, to which we can assign the value of the fixed pointer `square`. This variable pointer is then a pointer to a function. When we declare such a pointer, we must specify the return type and argument list of the function (or functions) it will be used to point to. For example, to declare a pointer `fnptr` to the function `square` above, we would have

```
int (*fnptr)(const int);

fnptr = square;
```

This complicated looking declaration notifies the compiler that `fnptr` is a pointer to some function that takes a `const int` as an argument, and has a return type of `int`.
We can now invoke `square` indirectly via `fnptr` as follows:

```
int x = 4, y;

y = fnptr(x);
```

Note that we can make `fnptr` point to *any* function whose argument list contains a single `const int` and which returns an `int`. Thus, if we had

```
int cube(const int);
```

then we could reassign `fnptr` to point to this function as follows:

```
fnptr = cube;
```

We can use `typedef` to simplify the declaration of a pointer to a function. In the above example, we could create a new type name `Pfunction`, to declare pointers to functions returning `int` and taking an argument of type `const int`, as follows:

```
typedef int (*Pfunction)(const int);
      .
      .
PFunction fnptr;
      .
fnptr = square;
```

The principal application of a pointer to a function is as an argument to another function. This is useful when we wish to write a "generic" function to process data generated by a number of different, but similar functions, as illustrated in listing 11.4.

12. PROGRAM ORGANISATION

Thus far, we have approached C programming from an implementation-oriented perspective. That is, we have considered the language syntax and constructs necessary to code simple tasks. For a software project of any significance, this is typically the last phase of the development cycle.

Software engineering is concerned with the design of large software systems. The goal is to produce software that is readily understood, reliable, easy to maintain and extensible. For large projects, this is virtually impossible to achieve, unless a systematic design philosophy is adopted. Important concepts in the design of complex systems are *data abstraction* and *data hiding*.

12.1 Abstract Data Types

An abstract data type (ADT) consists of an underlying data structure, and an *interface* to that data structure. The interface consists of a set of permissible *methods* that can be carried out on the data structure. To illustrate this concept, consider the example of a simple bank ATM machine. We identify the customer's account to be the fundamental unit of data in the problem, and create an underlying data structure called `account` to encapsulate all the relevant details of the customer's balance and transactions. Next, we define an interface to the `account` structure. The only way in which the customer can access the account is via the methods that we provide in the interface. For a simple ATM, a possible set of methods might be

```
make_withdrawal(account)
make_deposit(account)
display_balance(account)
```

From a software engineering perspective, data abstraction has a number of advantages:

- Maintainability is enhanced, since changes in the implementation of a data structure or algorithm can be localised to the region of the code that implements the ADT. So long as the interface to the data type remains unchanged, no "fall-out" effects are induced in other parts of the program, as it is only through the interface that the data structure can be manipulated.
- Reusability is enhanced, since the properties of the ADT are described by the interface methods. A programmer wishing to re-use a previously coded ADT need only understand the behaviour of the data type as specified by the interface methods, without concern for the implementation. From the programmer's viewpoint, the data type and method implementations are contained in a "black box" hidden from view.
- Extensibility is enhanced, since to extend the functionality of our ADT, we need only add a new method to the existing interface (with corresponding modifications to the underlying data structure, if necessary). Other parts of the system can then make use of this enhanced functionality merely by invoking the new method.

12.2 Data Hiding

Associated with each ADT is a boundary. Data and code residing within this boundary are *private* to the ADT, and cannot be "seen" by any part of the program residing outside the boundary. The only manifestation of the ADT outside the boundary is the interface. The interface methods represent the *public* part of the ADT, and provide the only means by which other parts of the program may manipulate its private data. Thus, in order to permit the implementation of ADT's, a programming language must provide for some form of "data hiding", which restricts the visibility of data and code to a specific part of the program.

Object-oriented languages, such as C++ or Smalltalk, were designed from the outset to encapsulate the concepts of ADT's and data hiding. Although C is not ideal in this sense, it has a number of features which allow us to implement ADT's. In order to understand how this may be achieved, we need to consider the concepts of *storage classes, scope, and linkage*.

12.3 Storage Classes

When a variable is declared in a C program, the storage allocated falls into one of two classes, namely *automatic* or *static*. The storage class determines the lifetime of the storage associated with the variable.

Unless otherwise specified, the storage class of a variable declared within a function is automatic. This means that the variable comes into existence and is appropriately initialised, only when the function is entered, and is destroyed when the function is exited.

Sometimes, the programmer may require a variable in a function to retain its value across different calls of the function. This can be achieved by declaring the variable with the storage class specifier `static`. If a static variable is initialised when it is declared, then the initialisation is performed *once only*, the first time the function is entered. Listing 12.1 illustrates the use of static variables within functions.

A variable declared outside of any function is said to be *external*, and the storage class for such a variable is `static`. This means that the variable comes into existence, and is initialised, as soon as the program begins executing, and remains in existence until it terminates.

12.4 Scope

The *scope* of a variable refers to the parts of the program to which the variable is visible.

As we have already seen, the scope of a variable declared inside a function is the function body. Such a variable can be used between the braces enclosing the function, but will not be recognised anywhere else in the program.

The scope of an external variable extends from the point of its declaration to the end of the file in which it is declared. Thus, the variable can be used by any function following the point of its declaration, but is not visible to functions defined before it. Such a variable is termed *global*. If the variable is declared at the top of the file, it is available to all functions in that file.

Although global variables can be used as a shortcut for passing information between functions (bypassing the argument list mechanism), this approach is not generally recommended, as it increases the interdependence between functions.

Since a function cannot be defined within another function, it follows that all functions are global entities.

12.5 Linkage

Each entity in a C program has an associated *linkage*, which determines whether or not that entity is available in other program units. An entity with external linkage is available in files other than the file in which it appears, whereas an entity with internal linkage is not.

The rules governing linkage are broadly as follows:

- All global variables have external linkage.
- All local variables have internal linkage.
- All functions have external linkage.

The programmer can prevent a variable or function with external linkage from being visible to other files, by prefacing the variable declaration, or function definition with the keyword `static`. This use of `static` ensures that such entities are then private to the file in which they appear, providing us with a mechanism for implementing data hiding in C.

A file which intends to make use of a function defined in another file must include an *external referencing declaration*, which consists of the usual function prototype, preceded by the keyword

`extern`. Such a declaration tells the compiler that we wish to use the function in the current file, but that the function definition is actually in another file. While it is also possible to have external references to variables declared in another file, this is regarded as poor programming practice. Instead, such variables should be manipulated only by the functions in that file. Listing 12.2 illustrates the use of external referencing declarations.

12.6 Header Files

As we have seen, in order for the functions defined in one source file to be available in a second file, the second file must contain external references to each of the required functions in the first. If many files want to use the functions in the first file, it saves on repetition (and reduces the risk of introducing an error), if we provided all the external referencing declarations for the functions in the file in a separate header file. Then, in order to use these functions, any other file would merely have to include the appropriate header file. Typically, this header file would also contain any `#defines` and `typedefs` required by the functions in the file. This idea is illustrated in listing 12.3.

12.7 Implementing an Abstract Data Type

We are now in a position to implement an abstract data type in C. The steps required are as follows:

1. Identify the underlying data structure required, and any private "house-keeping" functions needed to manipulate it.
2. Design the public interface to the private data structure.
3. Create a source file to implement the private functions required to manipulate the data structure, ensuring their privacy by preceding their definitions with the keyword `static`.
4. In the same file, implement the public functions (corresponding to the methods of the interface).
5. Place external referencing declarations to the public functions of the ADT's interface in a header file. In order to make use of the ADT via its public interface, any other file now need only include this header file.

Listing 12.4 provides an artificially simple illustration of the above procedure.

13. ELEMENTARY DATA STRUCTURES

In many applications, the choice of a suitable data structure is really the only major decision involved in the implementation. For a given data set, some structures require more space than others. For the same operations on the same data, some structures lead to more efficient algorithms than others. In this section, we will consider some simple data structures with widespread applicability.

13.1 Arrays

Perhaps the most fundamental data structure is the array. An array consists of a fixed number of data elements, stored contiguously, each element being accessed by an index into the array.

Advantages:

- Each element can be accessed in the same, constant time, provided its index is known in advance.
- Arrays support random access, since the elements can be accessed in any order, simply by specifying the appropriate indices.

Disadvantages:

- The size of an array is fixed, so the number of data items that can be stored is not permitted to grow, or shrink dynamically during program execution.
- Insertion of a new element, or deletion of an existing element are inefficient operations. To achieve the former, all elements following the insertion point must be shifted up to allow the new element to be inserted. To achieve the latter, all elements following the deleted element must be moved down, in order to fill the gap it leaves.

A simple algorithm illustrating these points appears in listing 13.1. The "Sieve of Eratosthenes" finds all the prime numbers, up to a specified limit. The program employs an array called `isprime`. The aim is to set `isprime[i]` to `TRUE` if `i` is prime, and to `FALSE` otherwise. This is achieved by marking all the multiplies of every number (except 1) as non-prime. Note that the efficiency of the algorithm hinges on having fast access to every element in the array, given its index. Also, although the size of the array `isprime` is fixed during execution, its size *can* vary from one run of the program to the next, by using dynamic memory allocation.

13.2 Linked Lists

A linked list is a one dimensional data structure. It consists of a collection of *nodes* joined together by *links*. As depicted in fig. 13.1, each node is comprised of two fields: a data field containing a single data item, and a link field, containing a pointer to the next node in the list. In C, a node can be represented using a structure, as follows:

```
typedef int Key;

struct node{
      Key key;
      Node *next;
};

typedef struct node Node;
```

The data items to be processed are commonly referred to as *keys*. By creating a new type name `Key`, we can easily modify our code to handle different data types. In the above example, the data set is comprised of integers. Links are implemented using pointers. Each structure of type `Node` contains a member variable `next`, which is a pointer to a structure of type `Node` (`Node *`). In order to link the current node to the next node in the list, we merely assign the address of the next node to this pointer. For example, to link two nodes, `node1` and `node2` together, we could have

```
Node node1, node2;

node1.next = &node2;
```

In order to simplify certain operations on the list, we create two special nodes: The *head* node resides at the start of the list, and has an empty data field. Its link field points to the first data-containing node in the list. The *terminal* node marks the end of the list, and its link field is set to point to itself. We will refer to the terminal node as the *z node*.

Linked lists allow efficient insertion of new nodes and deletion of existing nodes. Consider the example in fig. 13.2(a). In order to insert the node "I" into the list, we need only make the link field of the "L" node point to the "I" node, and the link of the "I" node point to the "S" node, i.e. insertion requires the readjustment of just two links.

To delete the "I" node again, we would make the link field of the "L" node point to the "S" node (fig 13.2(b)), i.e. deletion requires the readjustment of just a single link.

The operation of moving a node to a new position in the list is also efficient. As shown in fig 13.2(c), this is achieved by the rearrangement of three links.

A disadvantage of linked lists is that they only permit sequential access to the nodes. Thus, to access the k^{th} node in the list, we would start at the head of the list, and follow k links from one node to the next until we arrived at the k^{th} node. This is the *only* way we can access the k^{th} node. The process of navigating the list sequentially, from one node to the next, is termed list *traversal*.

Another operation which is inefficient is that of finding the node before the current node. Since each node contains a link only to the next node, this entails returning to the head, and traversing the list, until we reach the node which links to the current node. This procedure is required in order to insert a new

node *before* the current node, or to delete the current node, since both these tasks require the link in the previous node to be modified. For this reason, it is common in linked list implementations, to permit only deletion of the next node, and insertion after the current node.

In applications where it is necessary to efficiently locate the node before a given node, a doubly-linked list may be used. In this case, each node contains two link fields - the first linking to the previous node, the second linking to the next node. Such a structure is uncommon however, as it requires more storage than the singly-linked list.

Next, we turn our attention to a C implementation of a linked list (listing 13.2). Nodes are constructed using structures, as detailed above. When a new node is required, it is created using dynamic memory allocation. When a node is deleted, the storage previously allocated to it is released using the function `free`. This allows the list to grow, or shrink dynamically during execution, ensuring that there are precisely as many nodes as are there are data items.

Three functions are provided:

- `list_initialise` sets up the list, by creating new head and z nodes. The head node is linked to the z node to indicate that the list is empty. The z node is linked to itself.
- `insert_after` inserts a new node following the current node. The list can grow until all available storage is exhausted. The procedure followed for inserting the new node is precisely that detailed in fig. 13.2(a).
- `delete_next` deletes the node following the current node, in the manner depicted in figure 13.2(b). A temporary pointer is required to point to the node to be deleted, so that when we break the link to it from the previous node, we are still able to free the storage that has been allocated to it.

The above functions form the basis of the linked list. Other functions can make use of these basic operations to manipulate data stored in linked list form. As an example, consider the Josephus problem, which may be posed as follows:

N people stand in a circle, each armed with a handgun. The first person shoots the m^{th} person, who falls over. The next person in the circle, *m+1*, now shoots the m^{th} person, counting forward from himself / herself. The process continues in this fashion, until only one person remains standing. The problem is to determine the order in which the members of the circle are despatched.

The problem can be solved by means of a linked list. In order to simplify the solution, the last node in the list is linked to the first. Such a list is said to be *circular*. The solution appears in listing 13.3.

13.3 Pushdown Stacks

The data structures we have encountered so far permit access to *any* item of data stored in the structure. In certain applications, such flexibility is not required, and a more efficient implementation results from using a simpler data structure, offering only limited access to the stored data. The *pushdown stack* is a structure which severely restricts access to the data stored in it. Just two operations are permitted: a data item may be *pushed* (placed) onto the top of the stack, or a data-item may be *popped* (retrieved) from the top of the stack. Since the last item to be pushed onto the stack will always be the first item to be popped off it, the pushdown stack is referred to as a last in - first out (LIFO) structure. In C, functions to implement these operations might have the following prototypes:

```
void push(Key)
Key pop(void)
```

The function `push` takes a single data key as an argument, and places it on the top of the stack. The return value of the function `pop` is the data key retrieved from the top of the stack. These operations are illustrated in fig. 13.3.

A pushdown stack is the ideal mechanism for storing intermediate results in arithmetic calculations. To illustrate, consider the following expression:

```
5 * (((9 + 8) * (4 + 6)) + 7)
```

This expression can be computed using a stack. The sequence of operations would be

```
push(5);
push(9);
push(8);
push(pop() + pop ());
push(4);
push(6);
push(pop() + pop());
push(pop() * pop();
push(7);
push(pop() + pop();
push(pop() * pop());
printf("%d\n", pop());
```

Here, the order in which the operations are performed is dictated by the parentheses in the expression, and the convention that evaluation proceeds from left to right.

To facilitate the computation of an expression using a stack, we could convert the expression from the customary notation (which is known as *infix*) to *postfix* notation. In postfix notation, no parentheses are used. Instead, all operands precede the corresponding operator, and the operators appear in the order in which they are to be applied. Thus we may rewrite the expression above using postfix notation as follows:

```
5 9 8 + 4 6 + * 7 + *
```

Notice that there is a close correspondence between the expression in postfix form, and its evaluation using a stack.

A pushdown stack can be implemented in C by means of a linked list, or an array. The former would be used when the stack size is required to grow, or shrink dynamically during execution. Our previous implementation of the linked list would have to be modified to ensure that insertion and deletion are always performed at the beginning of the list.

If the maximum stack size can be anticipated in advance, then an array-based implementation may be easier. A basic stack implementation, using arrays, appears in listing 13.4.

13.4 Queues

Another restricted access data structure is the queue, which supports two basic operations: addition of a data item to the tail of the queue, or removal of a data item from the front of the queue. In contrast to the stack, the queue is a first in - first out (FIFO) structure, since the first item added to the queue will always be the first item to be removed. Queues are common in telephone exchange equipment, where subscriber requests are queued, and subsequently handled in the order in which they arrived.

We can modify our linked list code to implement the essentials of a queue, as shown in listing 13.5. Note that the tail of the queue, where the next item is to be added, corresponds to the head of the linked list. The front of the queue, from which the next item is to be removed, is represented by the z node. A disadvantage of this implementation is that, in order to find the item at the front of the queue, we must traverse the entire queue from the `tail` node to the `z` node. In this case, a doubly-linked, circular list would provide faster access to the front of the queue. As with the stack, a queue may be implemented using an array, provided that the maximum queue size can be anticipated in advance.

It is apparent from the foregoing, that linked lists, pushdown stacks and queues share a common structure. Each consists of a data structure, and a set of operations that can be performed on that structure. Hence, these data structures can readily be implemented as abstract data types, as described in the first part of the course.

14. TREES

The linked list encountered in the previous section is an inherently 1-D structure. We now consider a 2-D linked data structure called a *tree*.

A tree is a non-empty collection of nodes, connected by *edges*. A *path* is a list of nodes, in which successive nodes are connected by edges. A tree has one *root* node. The defining property of a tree is that there is precisely *one* path between the root node and each of the other nodes in the tree.

An example of a tree appears in fig. 14.1. The root node resides at the top of the tree. Every other node is connected to exactly one node above it, called its *parent*. The nodes below the parent, and connected to it, are termed its *children*. Nodes with no children are called *terminal* nodes.

If each node is constrained to have a certain number of children, the resulting structure is known as a *multiway* tree. In this case, special terminal nodes, called *external* nodes, containing no data and having no children, are used as dummy children for terminal nodes, which do contain data, but have no proper children. External nodes mark the end of a path through the tree.

The simplest, and most common *multiway* tree is the *binary* tree, in which all nodes (except external nodes) have two children. A sample binary tree appears in fig. 14.2. External nodes are represented by squares. In this example, M is termed the *left child* of P, while L is the *right child.*

In C, the following structure might be used to represent a binary tree node:

```
typedef int Key;
typedef struct info Info;
typedef struct node Node;

struct node{
      Key key;
      Info info;
      Node *left;
      Node *right;
};
```

The data for a node is contained in some structure of type `Info`, which we have not defined here. The purpose of the key is to act as an index for the information in the node. For example, say that each node in the tree contains information about a book. In this case, `info` might contain the title of the book, the author's name, the publisher and the ISBN code. `key`, however, might contain just the author's surname. Thus, finding all books by all authors with a particular surname, simply involves finding all nodes whose `key` contains that surname.

14.1 Ordered Binary Trees

An ordered binary tree is one where all the nodes in the left sub-tree beneath a given node satisfy some ordering relation (alphabetical, or numerical) when compared to the nodes in the right sub-tree. One way to construct an ordered tree is to take the next key in our data set and compare it that of the root node. If it is less, then we descend the tree to the left child node, otherwise we descend to the right child node. We continue according to this scheme until an external node is reached, then replace this external node with our new node. This procedure is illustrated in fig 14.3, for the data set {M, Y, S, A, M, P, L, E, T, R, E, E}. The motivation for constructing the tree in this fashion will become clear in section 14.3.

14.2 Implementation

Listing 14.1 provides a basic C implementation of a binary tree. Terminal nodes are represented by the special `z` node, whose `left` and `right` members link back to itself. Two functions are provided:

`tree_initialise` sets up an empty tree, by allocating storage to the `root` and `z` nodes. The `left` and `right` members of the `root` node are linked to the `z` node.

`insert` constructs a new node in the tree. The location of the new node is determined by the ordering relation described in section 14.1. The `left` and `right` members of the newly created node are linked to the `z` node.

14.3 Binary Tree Search

Imagine that we have constructed an ordered binary tree, as described above, and now wish to search the tree for the node corresponding to a particular data key. As a result of our ordering scheme, we know that all nodes in the left sub-tree beneath the current node contain keys that are strictly less then the key in the current node. All nodes in the right sub-tree contain keys that are greater than, or equal to the key in the current node. Thus, if we wanted to find the node containing a given key, we would start at the root node. If the key is less than that of the root, we know it must reside somewhere in the left sub-tree of the root, so we descend to the root's left child node. Otherwise we descend to the right child node. We repeat this procedure, for each node in the tree, until we arrive at the node containing the search key. Listing 14.2 details a C function to perform a binary tree search.

Now, if every node in the tree has an equal number of nodes to its left and to its right, the tree is said to be *balanced*. Then, each comparison in the searching procedure either terminates the search, or halves the number of nodes remaining to be searched. Hence, an ordered binary tree can be searched very efficiently, compared to searching a 1-D data structure. Since the same ordering scheme is used to construct the tree as to search it, insertion of new data is relatively slow. However, in many applications, such as our book database, insertion is not very common, whereas searching is. Thus the increased insertion time is a small price to pay for the reduction in searching time.

15. RECURSION

Recursion is a fundamental concept both in mathematics and computer science. Many algorithms lend themselves readily to a recursive implementation.

Stated simply, a recursive function is one which calls itself. We need to qualify this statement however, in order to prevent such a function invoking itself indefinitely. Thus, a recursive function must also include a *termination condition*. When this condition is met, the function does not call itself.

To illustrate the concept of recursion, we will now investigate Euclid's algorithm for computing the greatest common divisor (GCD) of two numbers. The method hinges on the observation that if *u* is greater than *v*, then the GCD of *u* and *v* is the same as the GCD of *v* and *u*-*v*. This claim can be trivially demonstrated by assuming *D* to be the GCD of *u* and *v*. If Δ is the GCD of $u-v$ and *v*, then

$$\frac{u-v}{\Delta} = \frac{u}{\Delta} - \frac{v}{\Delta}$$

i.e. the largest value of Δ is that which divides evenly into both *u* and *v*. By our initial assumption, this is *D*.

To couch the computation in a recursive framework, we need only to apply the same observation to $u-v$ and *v*, i.e. if *D* is the GCD of $u-v$ and *v*, and $u-v>v$ then *D* is also the GCD of $(u-v)-v$ and *v*. We apply this property repeatedly (updating *u* with the value $u-v$, and ensuring that *u* is always greater than *v* by swapping their values if necessary), until $u=0$ and *v* is the GCD of the original (and all intermediate) values of *u* and *v*. The efficiency of the recursion is improved if we note that, as long as $u>v$, we continue to subtract *v* from *u* until $u \le v$. This is equivalent to finding the remainder when *u* is divided by *v*, i.e. $u \,\%\, v$. A recursive implementation of Euclid's method appears in listing 15.1. To illustrate the operation of this code, Table 15.1 shows the values of *u* and *v* at the beginning of each call of the function `gcd`, when calculating the GCD of 461,952 and 116,298. The required answer (i.e. 18) appears in the final row of the table.

Euclid's method may also be implemented non-recursively, using a `while` loop, as in listing 15.2. Testing each implementation on the same machine, using the same compiler, reveals that the non-recursive function calculates the GCD of 461,952 and 116,298 more than five times faster than its recursive counterpart! This highlights a very important issue: great care must be taken in using recursion, as a recursive implementation is not necessarily the "best". A further (more drastic) example will serve to confirm this assertion.

The *Fibonacci numbers* are generated according to the following recurrence relation:

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n \ge 2 \text{ with } F_0 = F_1 = 1$$

This defines the sequence

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, …

Analysis reveals that F_n increases exponentially with *n*. The n^{th} Fibonacci number can be computed recursively, as in listing 3.3. The number of function calls C_n required to compute the n^{th} value is given by

$$C_n = 1 + C_{n-1} + C_{n-2} \quad \text{for } n \ge 2 ; C_0 = C_1 = 1$$

which is very similar to the recurrence relation for F_n, and it may be shown that C_n also rises exponentially with *n*. Furthermore, recall that when a function calls another function (including itself),

memory must be set aside to store the program state before the call is made. Thus, both the time and the amount of memory required to compute F_n grow exponentially with *n*. Now consider the function in listing 15.4 which computes the n^{th} Fibonacci number by means of an array. In this case, it is clear that the amount of time and memory needed increase linearly with *n*. Once again, the non-recursive implementation is preferable to its recursive counterpart.

15.1 The "Divide and Conquer" Paradigm

The gross inefficiency of the recursive Fibonacci calculation above results from repeated, unnecessary re-processing of the data. In some situations, it may be possible to (recursively) partition the input data into non-overlapping subsets, then to process each subset independently. This is the "Divide and Conquer" paradigm for algorithm design which, with large data sets often yields significant economies. To illustrate, imagine that you are Ghengis Khan, plotting to rule the world. You might start by partitioning the world into East and West, sending an army to conquer each region individually. Realising this to be a monumental task, you decide to further divide East and West into continents, and each continent into countries. Then you despatch armies to plunder each country separately. The recursive possibilities of "Divide and Conquer" are already apparent. To illustrate, we will consider a somewhat more mundane example - the Quicksort algorithm.

Quicksort is an efficient sorting algorithm, developed by C.A.R. Hoare in 1960. The sorting problem can be stated as follows: Given an array of "keys" (items to be sorted), and two indices, left and right, we wish to arrange the array elements between left and right in ascending order. Listing 15.5 details a basic implementation of Quicksort. The crux of the method is the `partition` function whose purpose is to place any given element (the sort key) in the array in its final position at index *i* in the sorted array. When `partition` returns, all elements less than the sort key are positioned to its left, while elements greater than or equal to the sort key are positioned to its right. Thus we create two subsets (the data to the left of the sort key, and the data to the right of the sort key) which we sort independently, by recursive application of the `partition` operation. Listing 3.6 provides a suitable implementation of this function.

We arbitrarily select the rightmost element to be the sort key. Scanning the array from left to right, we look for any element greater than the sort key, marking its position with a pointer. Similarly, scanning from right to left, we seek any element less than the sort key and mark its position with a second pointer. Whenever two such elements are located, they are swapped. We proceed through the array in this fashion, until the pointers meet. At this juncture, all elements less than the sort key reside on the left of the array, while those greater reside on the right. The final stage is to place the sort key in its final position, between the two subsets. The effect of a single call of the `partition` function is illustrated in Figure 15.1.

15.2 End-Recursion Removal

With functions that employ two recursive calls, the second call can be removed using a technique called end-recursion removal. The idea is based on the observation that the second recursive call merely invokes the function with a different set of arguments to the first - the statements executed during each call are identical. If we can contrive to modify the arguments after the first call, and execute the required statements *without actually making the second call*, then we have eliminated half of the recursion. This can be achieved by jumping back to the start of the function, using a `goto` statement. Listing 15.7 illustrates the technique, using Quicksort. Use of the dreaded `goto` is avoided by realising that the combination of the `if` statement and the `goto` can be replaced by an equivalent `while` loop, as in listing 15.8.

Optimal sorting efficiency is achieved when each partitioning stage divides the preceding partition precisely in two. On average, for large, random data files, this condition will obtain. The worst-case scenario arises when the *n* items in an array are already sorted. The rightmost element (i.e. the sort key), being the largest element, will not change place and the index *i* returned by the partition function will correspond to the position of the sort key itself. Thus, the next partitioning will proceed with *i-1* elements, and we have succeeded in reducing the partition size by a paltry single element. The entire process requires about *n* function calls, and each call requires memory to be set aside. The result may

be that we quickly run out of memory! We avoid this by comparing the sizes of the two sub-partitions after each partitioning stage. If we start with *n* keys, then Quicksort will be re-invoked with at most *n/2* keys, then *n/4* keys, and so on, i.e. the number of recursive calls will be logarithmic in *n*. (See Listing 15.9).

15.3 Removal of Recursion with a Pushdown Stack

When a function call is made, the compiler produces code to (a) push the values of the local variables and the address of the next instruction on a stack, (b) set the values of the parameters passed to the function and (c) jump to the start of the function. We can remove the second recursive call in Quicksort by mimicking the function call procedure using a pushdown stack, as in listing 15.10. Here, instead of re-invoking Quicksort ,we explicitly push the values for the left and right indices for the larger of the next sub-partitions onto a stack. The two calls to `pop()` at the beginning of the outer `while` loop mimic the restoration of local variables when a function returns. The principle overhead involved with this approach is the memory required to store the stack. Since the indices pushed onto the stack during each iteration correspond to the larger of the two sub-partitions, it follows from our previous discussion that at worst, the stack will grow logarithmically with *n*.

15.4 Recursive Traversal of Binary Trees

Recall from section 14, that a binary tree is a collection of nodes, connected by edges. Each node is connected to a single parent; each parent has two child nodes - a left child and a right child. Terminal nodes have no children.

We saw how to "grow" an ordered binary tree, by inserting new nodes according to a special ordering algorithm. Specifically, we compare the key of the new node to that of the root. If it is less, then we proceed to the left, otherwise we proceed to the right. We continue according to this procedure until an external node is reached, then replace this external node with our new node, as illustrated in figure 14.1. Such an ordering scheme permits efficient searching of the tree for a given key.

In addition to searching the tree for a specific key, there are other operations we may wish to carry out on the data in the tree. For instance, we might want to print out all the keys in the tree in ascending order. (If the keys consisted of a set of authors' surnames, this is equivalent to printing out a list of authors in alphabetical order). In order to achieve this, we need to visit every node in the tree in some pre-specified order, and print the key associated with each node. The process visiting each node in a tree once, and performing some operation there, is termed *traversal*.

There are numerous ways in which we can visit the nodes in a binary tree. Perhaps the most common method is in-order traversal. In-order traversal is defined by the following recursive rule: Visit the left sub-tree, visit the root, then visit the right sub-tree. Listing 15.11 details a C function to implement this strategy. We can print the whole tree by invoking the `traverse` function with a pointer to the root node, as follows:

```
traverse(head);
```

The significance of the in-order approach is that, if we store our data in the tree using the ordered insertion algorithm described above, then an in-order traversal will print a sorted list of keys, in ascending order. Figure 15.2 illustrates the sequence in which nodes are visited during in-order traversal of the tree we constructed in figure 15.1. The output of the traversal would be {A, E, E, E, L, M, M, P, R, S, T, Y}.

In the preceding discussion, we opted to print the key during our visit to each node. In practice, there are many alternative operations we could perform. In order to make our traversal algorithm independent of the operation to be performed, we can modify the `traverse` function to accept a pointer to the function ("`visit`") defining the required operation, as in listing 15.12.

We note two other schemes by which we may visit the nodes of a tree, namely pre-order and post-order traversal. The recursive rule for pre-order traversal is as follows: Visit the root, visit the left sub-tree,

then visit the right sub-tree. The analogous rule for post-order traversal is: Visit the left sub-tree, visit the right sub-tree, then visit the root. Functions to implement these rules are provided in listings 15.13 and 15.14 respectively.

15.5 Non-Recursive Traversal of Binary Trees

As previously, we seek to remove the recursion from our traversal function, in order to reduce the costly memory requirements and processing time associated with recursive functions. Once again, our strategy is to employ a push-down stack. We first consider the case of pre-order traversal, since it lends itself to a straightforward stack-based implementation. Recall that at each node, we visit that node at once, then move to the left, remembering that we must subsequently return and visit the nodes to the right. We "remember" to visit the nodes to the right by pushing the address of the right hand node onto the stack, and popping it off again later. Listing 15.15 details the implementation.

Finally, we consider a stack-based implementation of in-order traversal. Recall that for each node, we visit the left sub-tree, visit the root, then visit the right sub-tree. Once again, we remember the nodes we must return to by pushing their addresses onto the stack, as in listing 15.16. Note carefully the position of the termination condition. We cannot check for an empty stack at the top of the `while` loop, because the last action before returning to the top of the loop is to push the address of the right child node onto the stack. Instead we employ an infinite loop, and use an `if` statement to break out of the traversal, when the stack is empty.

16. MERGING

While studying Quicksort, we encountered the `partition` function, whose role was to locate an arbitrarily selected key in its final position in the sorted data. This was achieved by splitting the data into two sub-partitions, one containing all keys less than the sort key, the other containing all keys greater than the sort key. Recursive application of the partitioning operation resulted in a fully sorted set of keys. We employed the `quicksort` function to recursively "drive" the `partition` function.

Merging can be viewed as the complementary process to partitioning. Specifically, the *two-way* merging operation consists of taking two sorted sets and combining them to form a larger, sorted set. We will see how recursive application of merging can be used to sort a list of keys. As with Quicksort, we will employ a function `mergesort` to recursively drive the merging procedure.

16.1 The Merging Procedure

We start by considering how the merging operation might be implemented. A C function to merge two sorted arrays of keys, `keya[]` (containing *m* elements) and `keyb[]` (containing *n* elements) into one larger, sorted array `keyc[]` (containing *m+n* elements) appears in listing 16.1. Essentially, we have two markers, *i* and *j*, which point to the current elements we are considering in `keya[]` and `keyb[]` respectively. As long as `keya[i]` is less than or equal to `keyb[j]`, we make `keya[i]` the next element of `keyc[]`, and increment *i*. Otherwise, `keyb[j]` is less than `keya[i]`, so we make `keyb[j]` the next element of `keyc[]`, and increment *j*. This process is implemented by the combination of a `for` loop and an `if` statement. There are precautions which must be observed during the merging process: If *j* has been incremented up to *n*, then we have exhausted all the elements in `keyb[]`, so all that remains to be done is to insert the remaining keys in `keya[]` into `keyc[]`. Similarly, if *i* has been incremented to *m*, then all we need do is to insert the remaining elements of `keyb[]` into `keyc[]`. Note that the extra storage required to perform the merge is equal to the combined sizes of the two arrays to be merged.

The merging process can also be applied to data stored in the form of a linked list. Recall from section 1 that each node in a linked list contains a key, and a pointer to the next node in the list. The `head` node resides at the beginning of the list, the dummy `z` node terminates the list. Linked lists lend themselves readily to merging, because the operation of insertion is naturally supported, by simply redirecting a node's pointer to the next node, rather than "moving" the existing nodes to make space for a new node. Listing 16.2 details a function to implement merging of two (sorted) linked lists, list (a) and list (b), pointed to by `ptra` and `ptrb` respectively. `ptrc` is an auxiliary pointer which points to the most recent addition to the merged list. Initially `ptrc` points to the `z` node. The `if-else` block performs analogous operations to the `if-else` in listing 16.1: We determine when the keys in either list have been exhausted by checking if we have reached the terminal `z` node in each list. This is equivalent to testing for the array bounds *m* and *n* in listing 16.1. If `ptra->key` is less than or equal to `ptrb->key`, the next node in the merged list should come from list (a), so we direct `ptrc->next` to point to the node pointed to by `ptra`. Then, we must make `ptrc` point to this new addition to the merged list by setting `ptrc` equal to `ptra`. Finally, we move onto the next node in list (a) by setting `ptra` equal to `ptra->next`, and the process continues. If `ptrb->key` is less than `ptra->key`, we perform a similar set of operations to those just described, using list (b) instead of list (a), as indicated by the `else` block. The entire merging process is complete when `ptrc` points to the `z` node. At this point, we make `ptrc` point to `z->next` (remembering that we started our merged list with the `z` node). Then we reset the `z` node to point to itself. We return `ptrc` from our function as a pointer to the head of the merged list.

16.2 Recursive Mergesort

We can use the merging routine detailed above as the basis for a recursive sorting algorithm which we will call *mergesort*. The basic idea is this: We take our original set of keys to be sorted, and split it into two equal (or almost equal) sub-sets to be merged. We take each of these two sub-sets, and split them in two again, producing (for each sub-set) two further sub-sets to be merged. We proceed

recursively, dividing each of our sub-sets in two, until we arrive at sub-sets containing one element each. Obviously, a single element is considered to be sorted. Emerging from the deepest level of the recursion, we merge each single-element sub-set to form a sorted two-element sub-set. Coming out another level, we merge two sorted two-element subsets to form a sorted four-element sub-set. The process continues until we merge the two sorted sub-sets corresponding to the first division of the original set, to form a fully sorted set. (Note that, at each stage of the dividing procedure, if we have to divide an odd number of elements, our two sub-sets will differ in size by a single element, so we will end up merging sub-sets of unequal size). As with Quicksort, we develop a recursive Mergesort function to drive the `merge` function we described above. This function is detailed in listing 16.3, for an array of keys. Note that we must provide the `merge` function with an array `temp[]` to allow for temporary storage of intermediate merges. The operation of `mergesort` on the set of keys {A, S, O, R, T, I, N, G, E, X, A, M, P, L, E} is illustrated in figure 16.1.

Auxiliary storage requirements are virtually eliminated when we implement `mergesort` via linked lists, as in listing 16.4. Although we pass the list length *n* of the list to be sorted as a parameter to the function, this is not strictly necessary, since we could determine *n* by proceeding sequentially through the list, counting each node as we go, until we reach the `z` node. The form of the linked list `mergesort` function is similar to that of its array-based counterpart. We recursively split our original list (pointed to by `ptr`) in two, by proceeding to the node at the mid-point of the list, and making it point to the `z` node. We employ an auxiliary pointer `ptrb` to point to the first node of the second half of the list, thus creating two lists from our original list. We proceed in this fashion, until our sub-lists contain a single node. Analogously to the array-based Mergesort, we merge sorted sub-lists into larger sorted lists as we emerge from each stage of the recursion.

Recursive Mergesort is an even better example of the Divide and Conquer paradigm than Quicksort. With Quicksort, we performed a partition operation at each stage as we descended into the recursion. Mergesort, on the other hand, completely divides the input data by recursion, *before* we carry out any merging operations. It is only as we emerge from the recursion that any processing of the divided data takes place.

16.3 Stability

A property of sorting algorithms which can be of considerable importance is *stability*. We define a sorting method to be stable when the order in which keys of equal value arise in the unsorted data set is preserved in the sorted set. To illustrate why stability can be desirable, consider the following example: Suppose we have a class list, sorted alphabetically by surname. For each student, we have an associated exam grade. If we now sort the class list by grade, using a stable algorithm, then the names corresponding to each grade will still be in alphabetical order. An unstable algorithm, on the other hand, will not preserve the alphabetical order. Consider the following alphabetically sorted list of surnames and grades:

Barrichello	C
Berger	C
Brundle	C
Frentzen	B
Hakkinen	A
Hill	A
Irvine	B
Salo	D
Schumacher	A

If we were to sort this table by grade, a stable algorithm would produce the following result:

A	Hakkinen
A	Hill
A	Schumacher
B	Frentzen
B	Irvine

C Barrichello
C Berger
C Brundle
D Salo

In addition to having a list which is sorted by grade, the records corresponding to each grade are still in alphabetical order.

Mergesort is a stable sorting algorithm. To verify this assertion, we need only demonstrate that the merging operation itself is stable, since it is only during merging that the keys change place. Recall from listing 16.1 that if `keya[i]` is equal to `keyb[j]`, then `keya[i]` is the next element in the merged array. If `keya[i+1]` is equal to `keyb[j]` then `keya[i+1]` is the next element in the merged array, and so on until `keya[i+p]` is no longer equal to `keyb[j]` for some value *p*. Now `keya[i+p]` must be larger than `keyb[j]`, so the next element in the merged array is `keyb[j]`. Thus keys of equal value appear in the merged array in the same order as they appeared in each of the arrays to be merged, and so, by definition, merging is stable. (A similar argument applies to the merging of linked lists). To clarify the above, consider the following two lists, which are to be merged:

B	Frentzen	A	Hakkinen
B	Irvine	B	Coulthard
C	Berger	C	Barrichello
D	Salo	C	Brundle

After merging, we obtain

A Hakkinen
B Frentzen
B Irvine
B Coulthard
C Berger
C Barrichello
C Brundle
D Salo

Unlike merging, the partitioning operation we saw previously is not stable, so Quicksort is not a stable sorting algorithm.

17. ANALYSIS OF ALGORITHMS

Often, there are many algorithms which we may use to solve a given problem. The programmer is then faced with choosing the "best" algorithm for the task in hand. Assuming that other criteria, such as robustness and stability, have been met, the "best" algorithm is the one with the shortest running time. Unfortunately, for most non-trivial algorithms, it is impossible to precisely determine the actual running time of the algorithm. The goal of algorithm analysis is then to provide some meaningful estimate of running time, in the average, or worst-case sense. Our approach involves three distinct steps:

Step 1: Characterising the Input.
Ideally, given the probability distribution describing the occurrence of the input data, we would like to determine the corresponding probability distribution of running times. In most cases however, the probability distribution of the input is unknown to us, and we must content ourselves with placing an upper bound on performance, or estimating the average running time assuming a random input.

Step 2: Identifying Abstract Operations.
The execution time corresponding to a particular operation (for example, a comparison in a sorting algorithm), will vary widely between platforms, depending on many factors, such as the machine architecture, or (for a high level language) the ability of the compiler to produce an efficient machine language implementation of the source code. Our approach is to identify the abstract operations constituting the algorithm, rather than considering machine-specific details. This allows us to perform an initial comparison of algorithm efficiency, independent of the target architecture.

Step 3: Computing the Required Number of Abstract Operations.
Having identified the abstract operations involved, we seek to compute the required number of operations of each type, under average, and worst-case conditions. Generally, in our analysis, we will only consider the most important operations involved in the algorithm, *viz.* we are concerned with obtaining a rough estimate of performance, rather than a precise determination. We adopt this strategy in order to suppress unnecessary detail (which doesn't contribute significantly to the overall running time), thereby simplifying the analysis without adversely skewing the estimate. Furthermore, overly detailed expressions for the running time may be difficult to compare, thus defeating the original purpose of our analysis.

17.1 Classification of Algorithms.

Most algorithms have a primary parameter which is most significant in determining the running time. Typically, this is the number of data items to be processed, which we denote by *n*. For large *n*, many common algorithms have a running time approximately proportional to one of the functions listed below:

c	Running time is constant, irrespective of the number of data items to be processed. Ideally, we strive to design our algorithm to have this property.
lg n	The running time increases logarithmically with *n*, i.e. a large increase in the number of items to be processed results in a relatively small increase in the running time. In computer science, it is common to calculate logarithms in base 2, which we denote by *lg*.
n	The running time is linear in *n*. This is the optimal situation when the program must process all *n* inputs.
n lg n	The running time is said to be "*n lg n*", and typically arises in the case of algorithms based on the "Divide and Conquer" paradigm.
n^2	The running time is quadratic in *n*. Such algorithms are practical, only when the number of data items is small.
2^n	The running time is exponential in *n*. Practically, such an algorithm is likely to be inappropriate, but may be used when a "brute-force" solution is the only alternative.

The running time of a particular program is likely to be a weighted sum of terms corresponding to the functions listed above. As *n* increases, one of these terms will dominate. We can use this fact to classify an algorithm's performance according to its dominant term.

17.2 O Notation and Asymptotic Behaviour

As mentioned previously, we generally seek to calculate an estimate of an algorithm's running time, rather than making a precise determination. O notation allows us to classify algorithms, based on an estimate of their worst-case performance. The idea is to formulate the estimate as a function of the number of inputs, ignoring constant terms. A function $f(n)$ is said to be $O[g(n)]$ if and only if

$$\exists\, c_0, n_0 \text{ s.t. } f(n) < c_0 g(n) \quad \forall n > n_0$$

Thus, if the running time of the algorithm is given by the (unknown) function $f(n)$, we can place a worst-case bound on the running time by finding a function $g(n)$ for which the above condition holds. Then, we can state that the algorithm is $O[g(n)]$, i.e. the running time is at worst proportional to $g(n)$.

While O notation allows us to classify algorithms on the basis of worst-case performance, it is important to be aware of the limitations of the approach:

- The upper bound may not be the best upper bound - the least upper bound may actually be much lower.
- The input which produces the worst case may occur very infrequently in practice, and thus the upper bound may not be representative.
- The constant c_0 is unknown, and need not be small. If n_0 is small, then a large value of c_0 can dominate in the expression for running time.
- The constant n_0 is unknown, and need not be small. O notation tells us nothing about the algorithm's performance when $n < n_0$.

The first limitation above is removed, if we can place a least upper bound on the running time. Formally, this amounts to finding a function $g(n)$ such that the algorithm is $O[g(n)]$, subject to the condition that

$$\lim_{n\to\infty} \frac{h(n)}{g(n)} = 0$$

where $h(n)$ is any function such that the running time is $O[h(n)]$.

Now, suppose we analyse an algorithm, and arrive at the following expression for the running time:

$$time = a_0 n \lg n + a_1 n + a_2$$

for some constants a_0, a_1 and a_2. But, it is also true to state that

$$time = a_0 n \lg n + O(n)$$

Thus, if we are interested only in an approximate determination, then for large *n*, we may not need to find the values of a_1 and a_2. Furthermore, the asymptotic behaviour of the function is determined by the leading term, so that, as $n \to \infty$, we might choose to ignore the quantities represented by O-notation, and say that the algorithm is "about" $a_0 n \lg n$.

17.3 Basic Recurrences

Many algorithms are based on the principle of recursively decomposing a large problem into smaller ones, and using the solutions to the smaller problems to solve the original problem. The running time then depends on the size, and number, of sub-problems, along with the cost of the decomposition. We

will now analyse some typical cases that arise in practice. In each of the cases below, we will use the letter *c* to denote some cost associated with the operations involved.

CASE 1:
A recursive function loops through it's input, and eliminates one item. It then reinvokes itself to process the remaining *n-1* items. This process is described by the following recurrence relation:

$$c_n = c_{n-1} + n \quad for \;\; n \geq 2,\; c_1 = 0$$

Analysis:

$$\begin{aligned} c_n &= c_{n-1} + n \\ c_{n-1} &= c_{n-2} + (n-1) \\ \therefore c_n &= c_{n-2} + (n-1) + n \\ &= c_{n-3} + (n-2) + (n-1) + n \\ &\quad \vdots \\ &= 2 + 3 + \ldots + (n-1) + n \end{aligned}$$

But,

$$\begin{aligned} c_n &= n + (n-1) + \ldots + 3 + 2 \\ c_n &= 2 + 3 + \ldots + (n-2) + (n-1) + n \\ 2c_n &= n + (n+1) + \ldots + (n+1) + n \\ &= (n+1) + (n+1) \ldots + (n+1) - 2 \\ c_n &= \frac{n(n+1)}{2} - 1 \end{aligned}$$

CASE 2:
A recursive function halves its input data, and then reinvokes itself to divide the remaining $n/2$ items. (Note: This recursive division generates only a single subset, rather than left and right subsets, as we encountered with Mergesort).

$$c_n = c_{n/2} + 1 \quad for \;\; n \geq 2,\; c_1 = 0$$

where we assume that $n = 2^m$.

Analysis:

$$\begin{aligned} c_{2^m} &= c_{2^m/2} + 1 = c_{2^{m-1}} + 1 \\ &= (c_{2^{m-2}} + 1) + 1 \\ &\quad \vdots \\ &= c_{2^0} + m = c_1 + m \\ &= m \end{aligned}$$

But,

$$\begin{aligned} & n = 2^m \\ \Rightarrow\; & m = \lg n \\ \therefore\; & c_n = \lg n \end{aligned}$$

CASE 3:
A recursive function loops through its i/p data, halves it, then reinvokes itself to process the remaining $n/2$ items.

$$c_n = c_{n/2} + n \ \ for \ \ n \geq 2,\ c_1 = 0$$

assuming $n = 2^m$.

Analysis:

$$\begin{aligned} c_{2^m} &= c_{2^{m-1}} + 2^m \\ &= c_{2^{m-2}} + 2^{m-1} + 2^m \\ &\quad . \\ &\quad . \\ &= c_{2^0} + 2 + \ldots + 2^{m-1} + 2^m \end{aligned}$$

But,

$$2c_{2^m} = 4 + 8 + \ldots + 2^{m-1} + 2^m + 2^{m+1}$$

$$2c_{2^m} - c_{2^m} = 2^{m+1} - 2$$

$$c_{2^m} = 2(2^m - 1)$$

$$c_n = 2(n-1)$$

CASE 4:

A recursive function splits its i/p data in half, then reinvokes itself to split each of the two halves in half. (This is recursive division involving left and right subsets, as in Mergesort).

$$c_n = 2c_{n/2} + 1 \ \ for \ \ n \geq 2,\ c_1 = 0$$

assuming $n = 2^m$.

Analysis:

$$\begin{aligned} c_{2^m} &= 2c_{2^{m-1}} + 1 \\ &= 2(2c_{2^{m-2}} + 1) + 1 \\ &= 2^2 c_{2^{m-2}} + 2 + 1 \\ &= 2^3 c_{2^{m-3}} + 2^2 + 2 + 1 \\ &\quad . \\ &\quad . \\ &= 2^{m-1} c_2 + \ldots + 2 + 1 \\ &= 2^{m-1} + \ldots + 2 + 1 \end{aligned}$$

$$2c_{2^m} = 2^m + \ldots + 4 + 2$$

$$2c_{2^m} - c_{2^m} = 2^m - 1$$

$$\therefore c_n = n - 1$$

CASE 5:

A recursive function loops through its i/p data, splits it in two, then reinvokes itself to process the two halves.

$$c_n = 2c_{n/2} + n \ \ for \ \ n \geq 2,\ c_1 = 0$$

assuming $n = 2^m$.

Analysis:

$$
\begin{aligned}
c_{2^m} &= 2c_{2^{m-1}} + 2^m \\
&= 2(2c_{2^{m-2}} + 2^{m-1}) + 2^m \\
&= 2^2 c_{2^{m-2}} + 2.2^m \\
&= 2^3 c_{2^{m-3}} + 3.2^m \\
&\quad . \\
&\quad . \\
&= m.2^m \\
\therefore c_n &= n \lg n
\end{aligned}
$$

17.4 Analysis of Quicksort

By way of illustration, we shall now apply the foregoing to the analysis of Quicksort.

Step 1:

We will estimate the average performance, rather than the worst-case performance. Hence, we will assume that the keys are in completely random order at the outset. We will further assume that that all of the keys are different, though in practice, this need not necessarily be the case.

Step 2:

We identify the abstract operations involved: In the case of sorting, the most important are compare and swap. For simplicity, we will restrict our analysis to the number of comparison operations required in the average case.

Step 3:

Denote the number of comparisons required to sort *n* keys by c_n. Then, the following recurrence relation describes the number of comparisons required:

$$c_n = (n+1) + \tfrac{1}{n}\sum_{k=1}^{n}(c_{k-1} + c_{n-k}) \quad for\ n \geq 2,\ c_1 = c_0 = 0$$

The term $(n+1)$ covers the covers the cost of comparing the sort key with each of the others, in the partitioning function. The sort key is compared to the remaining $n-1$ keys, but we must allow for the case when the scanning pointers cross over, resulting in two keys being compared twice. The summation is a probabilistic term. It says that each element *k* is equally likely to be the one where the partition is made. (As we emerge from the recursion, the two sub-partitions corresponding to the current partition contain $k-1$ and $n-k$ keys, each incurring a cost of c_{k-1} and c_{n-k} comparisons respectively).

By expanding the summation term, we find that

$$c_n = (n+1) + \tfrac{2}{n}\sum_{k=1}^{n-1} c_k$$

$$nc_n = n(n+1) + 2\sum_{k=1}^{n-1} c_k$$

But,

$$(n-1)c_{n-1} = n(n-1) + 2\sum_{k=1}^{n-2} c_k$$

$$\therefore\ nc_n - (n-1)c_{n-1} = 2n + 2c_{n-1}$$

$$c_n = \left(\frac{n+1}{n}\right)c_{n-1} + 2$$

Now,

$$c_{n-1} = \left(\frac{n}{n-1}\right)c_{n-2} + 2$$

$$\therefore\ c_n = \left(\frac{n+1}{n}\right)\left[\left(\frac{n}{n-1}\right)c_{n-2} + 2\right] + 2$$

$$= \left(\frac{n+1}{n-1}\right)c_{n-2} + 2\left(\frac{n+1}{n}\right) + 2$$

$$= \left(\frac{n+1}{n-2}\right)c_{n-3} + 2\left(\frac{n+1}{n-1}\right) + 2\left(\frac{n+1}{n}\right) + 2$$

.

.

$$= 2(n+1)[\tfrac{1}{3} + \tfrac{1}{4} + \tfrac{1}{5} + \ldots + \tfrac{1}{n}] + 2$$

We can approximate the above summation by an integral, i.e.

$$\sum_{i=3}^{n} \frac{1}{i} \approx \int_{3}^{n} 1/x\ dx = ln(n/3)$$

Hence

$$c_n \approx 2 + 2(n+1)ln(n/3)$$

For large *n*, the dominant term in the above expression is $2n.ln(n)$, so we may say that, on average, Quicksort requires $2n.ln(n)$ comparisons.

We can use O notation to place an upper bound on the performance of Quicksort. In the worst case, the data is already sorted, so that *n* comparisons are required to perform the partitioning operation, and the size of the next subset to be partitioned is reduced to *n-1* keys. Thus,

$$c_n = n + c_{n-1} \quad for\ \ n \geq 2\ and\ c_1 = c_0 = 0$$

$$= \frac{n(n+1)}{2} - 1$$

from Case 1 above. Thus, for large *n*, we have $c_n \approx n^2/2$, which demonstrates that Quicksort is $O[n^2]$.

17.5 The Concept of NP-Hard Problems

In previous examples, our *O*(*n*) analysis has resulted in computational complexity being typically *n*, *n*log*n* or n^2. In general, these can all be though of as finite polynomial expressions in *n*. However, it turns out that there are significant numbers of problems for which the computational complexity cannot

be expressed as a finite polynomial in *n* (where *n* is a number dimensioning the input). Such problems are referred to as NP-hard, where NP stands for non-deterministic polynomial.

Example 1: A scheduling problem

You have a factory with three (*m* in general) production lines, and a series of *n* jobs to do. Job *i* takes a time t_i to complete (including all overhead and set up time, *etc*.). A job can only be completed in one session on a production line, and a production line can only do one job at a time. As an example, consider the case where you have 7 jobs, which take the following times to complete {2,14,4,16,6,5,3} minutes. Your task is to allocate the jobs to the production lines in such a way as to minimise the overall time to completion. For example, here is one possible allocation:

Job Schedule 1

Line 1	2	14		
Line 2	4	16		6
Line 3	5	3		

It is apparent that Job Schedule 1 is not an optimal arrangement, as the overall time to completion is 26 minutes, but for much of that time Line 1 and Line 3 are idle. In fact an optimal schedule might look like:

Optimal Job Schedule

Line 1	16			
Line 2	5	4	2	6
Line 3	14			3

In the optimal job schedule, the total time to completion is 17 minutes, which is much better than Job Schedule 1. The problem is that no-one has discovered an algorithm which is guaranteed to provide the optimal schedule in a time limited by $O(m^k n^p)$ for any constants *k* and *p*, where *m* is the number of Production Lines and *n* is the number of jobs to be allocated. Mathematically, it remains an open problem to prove that no such algorithm exists, but there is empirical evidence that such algorithms do not exist.

17.6 Greedy Algorithms

So what should be done for problems which are NP-hard? In general, people will settle for sub-optimal solutions which run in a shorter time, and which give acceptable performance. Algorithms which provide sub-optimal solutions could be divided into various classes. One commonly encountered class of algorithms are "greedy algorithms". These are algorithms which act "locally" in that they attempt to do what appears to be the best thing in the short term, even when this impacts adversely on the overall solution. For example, in our scheduling problem, we might do the following scheduling algorithm:

Arrange the jobs in reverse processing time, and then assign the longest job to the first available processor line, i.e., order the jobs as {16,14,6,5,4,3,2}. Assign the job of length 16 to Line 1, job of length 14 to Line 2, job of length 6 to Line 3. Then, since Line 3 becomes available next, assign job of length 5 to it. Then Line 3 becomes available again, so assign job of length 4. Line 2 is then next to become free, so assign job of length 3. Then Line 3 becomes available next, so assign it job of length 2. This results in the following schedule.

Line 1	16			
Line 2	14		3	
Line 3	6	5	4	2

It turns out that in this case, the greedy algorithm does a good job, and arrives at an optimal scheduling algorithm. In general, however, this is not true, though a well designed greedy algorithm will usually arrive at a sub-optimal solution which is "close" to that desired.

Example 2: The knapsack problem

Another classic problem is that of deciding what objects to put in a knapsack, where different objects have different weights, and different values to you. (For example, carrying a knife on a camping trip might contribute little weight, but have high value, whereas bringing your kitchen sink would have high weight, but little value). Expressed mathematically, you have n objects, which each have a weight w_i, and a value v_i. You have a maximum weight which you can carry W. You would like to reach the maximum value V in your knapsack, while obeying the weight constraint. We introduce the variable x_i, which is 0 if object is not in the knapsack, and 1 if it is in the knapsack. Therefore, we have the problem, choose the values x_i such that we maximise V defined below, and the weight constraint is satisfied:

$$V = \sum_{i=1}^{n} x_i v_i \text{ such that } \sum_{i=1}^{n} x_i w_i \leq W$$

In general, this is a NP-hard problem. A greedy algorithm to solve this might be simply, add the objects with highest value first until you exceed the weight criterion, and then remove last object added. For example, if you had the objects {A,B,C,D,E} with values {10,5,4,3,2} and weights {2,10,1,4,1}, with a total weight constraint of 14, then you would add objects A, B, C and D, which gives a weight of 17, which exceeds 14, then simply remove D to satisfy the weight constraint, so that our final choice of objects would be A, B, and C for a weight of 13. However, it's easy to see that in fact we could also squeeze in object E, without violating our weight constraint. So in general, a greedy algorithm will not provide the optimal solution, but will provide a good sub-optimal solution.

18. INTRODUCTION TO OBJECT-ORIENTED DESIGN

So far we have considered data structures and algorithms in the context of the C programming language. The C language was first proposed in the early 1970s, and has become a *de facto* standard for many engineering applications. In fact, many new programming languages have retained the flavour of C programming by using conventions such as a semi-colon to mark the end of a statement, comments delimited by /* */, *etc*. C is sometimes referred to as a "middle-level" language, somewhere between a true high-level language (like BASIC) and low-level language (such as the assembly language for the x86 microprocessors). C has the "high-level" features of data abstraction, intelligibility, and portability, but also provides some low-level operations such as bit-level operators, and memory access.

An equally important distinction is that C is a procedurally-oriented language. In writing a C program, there is usually a "beginning", "middle" and "end" in mind. For example, a program might read in a data file (the beginning), sort the data (the middle), and then print out the sorted list (the end). This is a powerful programming *paradigm* (way of doing things), as is testified by the longevity of the C language and other procedural languages.

However, some tasks in life do not fit naturally into a procedural scheme. A popular arcade game of the early 1980s was "Space Invaders", in which the player controlled a set of guns at the bottom of the screen while "blobs" floated in from above in various formations. To the outside observer, all the elements in the game moved independently. A range of events could happen at any time (*e.g*., the gun could fire, a space ship could move left, the player could be destroyed), and each realisation of the game would be unique. Contrast this with the C data-sorting program described above. While the data sets might change, the overall flow of the program would be identical from execution to execution.

A further programming challenge was the advent of graphical computer environments such as the Mac OS, Windows, and X11 with screens capable of displaying multiple windows. In such an environment, each window can be considered as its own object, capable of operating independently. For example, a window might be closed, have its size changed, or be minimised.

In response to these complex tasks, a new programming paradigm emerged called "object-oriented design" (OOD). This software design philosophy was aimed at simplifying the programming for tasks such as the "Space Invaders" game and the graphical user interface mentioned above. Several computer languages were developed to implement the new OOD philosophy, i.e. C++, Eiffel, Java, *etc*. These new languages provided varying degrees of purity in their implementation of the OOD philosophy, as

most of them retained features from earlier procedural languages. For example it is perfectly possible (and frequently done) to write routines in C++ which could equally well have been written in C.

These introductory notes are not intended as course in C++ programming, though we will use C++ as an illustration of object-oriented ideas. The important thing to grasp from these notes is the fundamental features of object –oriented design which differ from procedural design. At the end of this section, you should be familiar and understand the following terms: *class, object, method, private vs. public, encapsulation, inheritance, and polymorphism*. We will use trivial examples, but the points they illustrate can be applied to far more complex programming tasks.

18.1 Classes and Objects

To understand the concept of a class and its role in OOD, it is instructive to take a step back and reconsider how procedural (C-like) programming is carried out.

An early definition of programming was "*Data+Algorithms=Programs*", and this is very much how C works. For example, if the task was to find the sum of a set of 50 numbers in a table, a pseudocode program might look like:

Define array to hold 50 numbers **DATA DEFINITION**
Define an overall sum and set to 0

Do the following 50 times:
Read in a number and add to the sum. **ALGORITHM DEFINITION**

Print out the sum

There is a clear division between data structure definition and the actions of the program. This is a common feature of procedural programs. Algorithms and data structures are the central components of procedural programs.

The key concept in object-oriented programming is that the division between data structure definition and actions is removed. A new concept called a *class* is introduced. The most significant feature of classes is that a class contains *both data and functionality* in the same item.

This is easier to see if we consider concrete examples. Let us assume we are designing some graphics package for drawing diagrams. One of the things that we like to draw with are rectangles, so we will define a class called Rectangle. The **Rectangle** class will have data:

- Co-ordinates of corners
- How thick the border line is
- The colour of the interior

but it will also have functionality, such as

- Draw yourself
- Change colour of interior
- Change the thickness of the surrounding line
- Disappear

As a different example, imagine we are writing a program for the Space Invaders example mentioned above. We would like to define a class called **Player**, which will represent the player's gun. The **Player** class will have data members

- Current position
- How long you have survived
- Number of missiles left

and also incorporate functionality, such as

- Move left
- Move right
- Fire a missile
- *etc.*

To begin, we will however choose a very simple example. Assume we are writing a program to handle payroll and personnel records. A fundamental unit of information will be a staff member. Let us say that the only things we are concerned with are (a) the person's staff number, (b) the person's age, (c) the person's salary, (d) the ability to change the salary to a new number (e.g., annual payrise), and (e) the ability to view the record. Note that the program specification mixes together data and actions – this is often typical of the way humans think.

Listing 6.1 gives a definition of a simple class that provides us with these features. There is a lot of new material in this snippet of code, so we will analyse it in some detail to extract the important concepts.

Let us focus first of all on the definition of the class called **StaffMember**. Conceptually, from the program specifications listed above, we know that such a class will have the ability to store the person's age, their staff number, and their current salary. This are all data items, which we are familiar with handling using `int`, `float` *etc.* type variables. The new thinking is involved with including actions for the class. Specifically we are requested to provide two possible actions: (1) display current values of age, staff number and salary, and (2) provide the ability to change salary. So the class must include two *methods* (the OOD jargon for functionality) which can carry out these tasks. The first new concept encountered is this combination of data and functions under one umbrella. This concept is called *encapsulation*, since the data and methods are tightly bound (encapsulated) together.

The next new concept is *private versus public* visibility. In fact, this is not so different from the concept of scope that we encountered already. However, the programmer must now make explicit decisions about every single item of data and every method in his/her program. This allows us to bring the "black-box" approach to software design. A "black box" in engineering is a box with an input wire and an output wire, and whose internal workings are unknown. All we know is that if we connect up the input wire, the behaviour of the output will obey the input-output characteristic for the box. So the input and output wires are *public* (openly accessible), but the internal box wiring is *private*. For our software example, the only public access to the class `StaffMember` is (a) when defining, (b) when viewing the current values, and (c) when changing the value of salary. There should be no other way to interfere with the internal workings of the `StaffMember` class.

This is achieved by using `Private` and `Public` visibility. The keyword `private` means that things (data or methods) defined as `private` can only be seen or used from inside the class. Things that are `public`, however, can be accessed by the outside world.

With the two new concepts of encapsulation and private/public visibility, let us analyse the first part of the `StaffMember` class, which is the class definition:

```
class StaffMember
{
      private:
            int age;
            long int staff_number;
            float salary;
      public:
            StaffMember(int,long int, float);
            void showStaffMember(void);
            void change_salary(float);

};
```

Do not be too concerned with the syntax at this stage, instead concentrate on the two important concepts described above.

Firstly the class has both data items (`age, salary, staff_number`) and methods (`showStaffMember, change_salary`). This is encapsulation. Secondly, all the internal data variables are `private`. This means that nothing outside the class can access these values or alter them by mistake. This provides "black-box" properties for the class.

Let us now consider some of the actual C++ details. Data variables are constructed from the C fundamental types, so there is no new syntax to learn. Method definitions are very similar to the function prototypes used in C, in which a list of variable types (and optional dummy variables) is provided.

Now that we have defined a class, how do we actually use it? The first thing is that we need to create an *instance* of the particular class in question: `StaffMember`. This is somewhat analogous to the process of type definition in C. For example in C, we could first define a new type e.g.,
`typedef int Age`
and then use that type definition to create an instance of a variable of type Age using
`Age a;`
In the same way, having defined a generic template for a `StaffMember`, we can now go ahead and create an instance of a particular StaffMember using
`StaffMember secretary(34,9008010025L,15450.0);`

This creates an instance (or object) of class `StaffMember` called `secretary`. Moreover, we have also given initial values to this object using a special method called the *constructor:*

```
StaffMember::StaffMember(int a, long int n, float s)
// Constructor method
{
      age=a;
      staff_number=n;
      salary=s;
}
```

Every class must have at least one constructor method (we will consider the possibility later of classes with more than one constructor). For the time being, consider the constructor method as a means of initialising the class (*i.e.*, setting up its internal variables). The specific C++ syntax for a constructor requires you to give the classname, followed by a double colon, followed by the constructor name plus arguments. The arguments can then be assigned to the private and public variables of the class.

Having created an instance of the class `StaffMember` called `secretary`, what can we do with it? In this particular contrived example, there are only two possibilities. One is to view the contents of the object secretary, and the second is to change the salary field. The first task is done using the publicly available method called `showStaffMember`. Generically this is often referred to as an *accessor* method, since it allows you access to viewing the internal data values.

```
void StaffMember::showStaffMember(void)
// Display values of StaffMember data
{
      cout << "Age is " << age << "\tStaff Number is " <<
staff_number << "\tSalary is £" << salary << "\n";
}
```

Specific points to note about the C++ implementation of this are the use of a new function called `cout`, which performs the same role as `printf` in C (note that to use `cout`, you must include `iostream.h`). The syntax is simply to set up a concatenated string using the << operator, which will then be fed to the standard output device (screen). In general accessor methods will take no arguments, and return no arguments so `void showXXX(void)` is typical.

The only other thing we can do with an object of class `StaffMember` is to change the value of the `salary` variable. The only way to do this from outside the class is through the `change_salary` method. The line

```
      secretary.change_salary(16250.0);
```

calls this method for the object secretary, and supplies a new value for the salary variable. The syntax shown is quite typical for OOD languages [*object_name.method_name*(*arguments*)]. When you view the contents of `secretary`, the value of salary has been updated to this new value. Methods that allow you to change the values of internal variables in a class are called *mutator* methods (because they mutate values).

18.2 Constructor Methods

For the example given above, we indicated that a constructor method was used to initialise a particular object. In OOD jargon, we created an instantiation (or instance) of class `StaffMember` called `secretary`. The constructor method that we used allowed us to provide initial values for the three internal variables of the class. However, we may wish to create an instance of `StaffMember` without knowing these values, or knowing only some of the values. The OOD paradigm allows you to provide several different types of constructor for each class.

In general a constructor function is any function that has the same name as its class. It will have no return type, as all it does is create an instance. A special type of constructor function is a default constructor; it will be called when you create an instance of a class without supplying any arguments. For our example above, a useful default constructor might be:

```
StaffMember::StaffMember(void)
// Constructor method
{
      age=0;
      staff_number=99999999;
      salary=1.0;
}
```

If you create an instance of type `StaffMember` with no arguments, the values 0 ,99999999, and 1.0 will be automatically assigned to the `age`, `staff_number`, and `salary` variables of the object. You must also add the "method prototype" to the list of public methods for this class, i.e.,

```
      public:
            StaffMember(void);
```

To use this default constructor, simply create an instance of class `StaffMember` with no input arguments:

```
      StaffMember vicepresident;
```

Then `vicepresident` will have the values `age`=0 , `staff_number`=99999999, and `salary`=1.0. (Note that the statement `StaffMember vicepresident( );` does not have the same effect, and will not create an instance). The choice of which constructor to use is determined automatically by the number of input arguments. Finally, if you wish to define a constructor method which takes values for `age` and `staff_number`, but which provides a default value of 1.0 for salary, we can use

```
StaffMember::StaffMember(int a, long int s)
// Constructor method
{
      age=a;
      staff_number=s;
      salary=1.0;
}
```

Listing 6.2 gives an example of how different constructors might be used in the same program.

18.3 Inheritance

It is often the case in programming that we wish to reuse objects, but with some slight variations. For example, in writing our personnel program, we might realise that we have certain special categories of `StaffMember`, who have extra attributes and capabilities. For example, we might want to create a class called `Supervisor`, which has the same basic variables and methods as a `StaffMember`, but for whom we want to add a field which will represent the number of employees they supervise. We also want to be able to look at this value.

One way to do this is to create a totally new class from scratch, but one of the strengths of the OOD paradigm is that we can modify an existing class to add extra features. This concept is called *inheritance*, since the new class `Supervisor` will *inherit* most of its features from an existing class `StaffMember`. Inheritance is the capability of deriving one class from another class. The initial class used as the basis for the derived class is referred to as the *base*, *parent*, or *superclass*. The derived class is referred to as either the *derived*, *child*, or *subclass*. Listing 18.3 gives the code needed for creation of a new class called `Supervisor` from the base-class `StaffMember`. We will examine the definition of the `Supervisor` class in some detail in order to establish the concept of inheritance.

```
class Supervisor: public StaffMember   // Supervisor is derived from
StaffMember
{
      private:
            int number_of_employees;
      public:
            Supervisor(int a,long int s,float salary, int n);
            void showEmployeeNumber();

};
```

The first thing that is different is that we have

```
class Supervisor: public StaffMember
```

at the top of the class definition. This line says that we are creating a new class called `Supervisor`, which will inherit its attributes from the publicly available class called `StaffMember`. That means that a `Supervisor` object will automatically have the following variables: `age`, `salary`, `staff_number` and methods: *constructor*, and `showStaffMember`. That is the concept of reuse. However, for a `Supervisor` object we now want to add in the special feature of a `number_of_employees` field, plus the ability to view that number. Accordingly, we add in an extra private variable

```
int number_of_employees,
```

plus a new public method,

```
void showEmployeeNumber( );
```

These additions extend the capability of the class. It is also interesting to consider the constructor method for this new class. Intuitively, we would like to be able to use the same constructor as for `StaffMember` but with the extra feature of defining a value for `number_of_employees`, which is the new private variable. The constructor

```
Supervisor::Supervisor(int  a,  long  int  s,  float  salary,  int
n):StaffMember(a,s,salary)
// Constructor method
{
      number_of_employees=n;
}
```

achieves this aim by passing in four values to the constructor, but the calling the `StaffMember` constructor to assign the first three variables.

Finally, using methods for the new class is no different from using methods for the parent class, i.e., `linemanager.showStaffMember();` will display the values `age`, `salary`, `staff_number` for the object `linemanager` which is an instance of class `Supervisor`.

There is one final important point to note. In the definition of the class we replace the keyword `private` with the keyword `protected`. This was done for the following reason. Private variables or methods cannot be inherited by children classes (otherwise we could simply circumvent the privacy of a class by deriving our own subclass). If the author of the original class intends that inheritance be used, he/she sets the variables up as protected. That means that the variables are "protected" from external actions, but can be inherited by new classes.

18.4 Polymorphism

The final concept we will introduce is called polymorphism. Polymorphism allows a superclass and a subclass to operate in different ways for the same method call. The advantage of this is it allows subclasses to have more specific behaviour.

As a concrete example, let us try to make our method `showStaffMember` operate differently depending upon whether we are accessing a `StaffMember` or a `Supervisor`. For the `StaffMember` we simply want to view the three variables `age`, `salary`, and `staff_number`, whereas for the `Supervisor` class, we would like to also view the `number_of_employees` variable. In other words, what happens when we use `x.showStaffMember()` will depend upon what exactly `x` is. This is polymorphism. Listing 18.4 gives the implementation of how we achieve this.

The important point to note is the use of the virtual keyword in defining the method `showStaffMember` in class `StaffMember`. A virtual function specification tells the compiler to create a pointer to a function, but not fill in the value of the pointer until the function is actually called. Then, at run time, and based on the object making the call, the appropriate function address is used (you may remember we looked briefly at pointers to functions in our C programming). You will also hear the word *dynamic binding* used to describe this effect. That means that the implementing function is determined dynamically. This is analogous to dynamic allocation of memory, where memory is not actually reserved until run-time.

APPENDIX A: DIAGRAMS AND PROGRAM LISTINGS

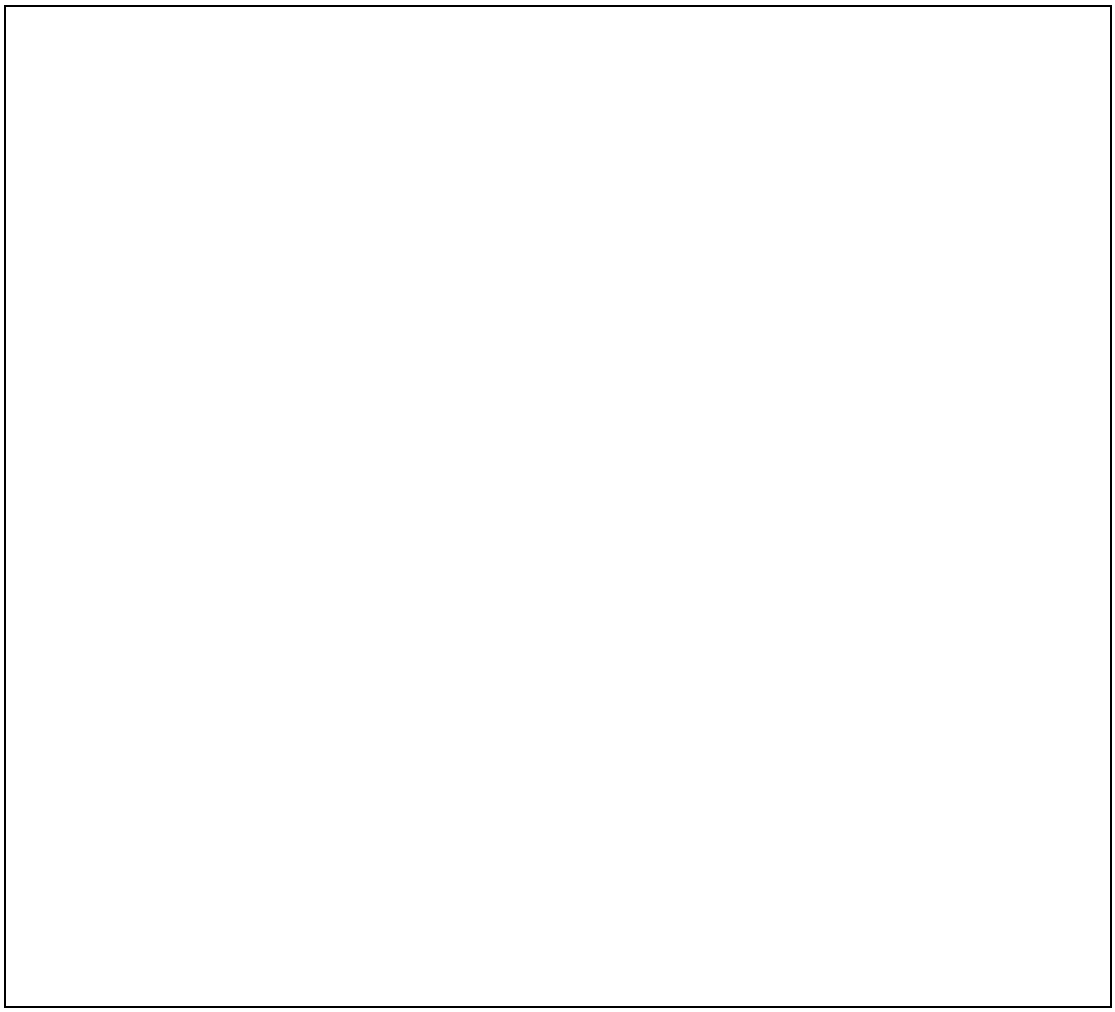


Figure 1.1: The Program Development Cycle.

LISTING 1.1

```
#include <stdio.h>

/* Program to print the words "Goodbye World!" to the screen */

int main(void)
{
      printf("Goodbye World!\n");

      return(0);
}
```

LISTING 1.2

```
#include <stdio.h> /* Program to print the words "Goodbye World!" to
the screen */ int main(void){printf("Goodbye World!\n");return(0);}
```

Type Name	**Bytes**	**Other Names**	**Range of Values**
char	1	signed char	–128 to 127
unsigned char	1	none	0 to 255
int	4	signed, signed int	–2,147,483,648 to 2,147,483,647
unsigned int	4	unsigned	0 to 4294967295
short	2	short int, signed short int	–32,768 to 32,767
unsigned short	2	unsigned short int	0 to 65,535
long	4	long int, signed long int	–2,147,483,648 to 2,147,483,647
unsigned long	4	unsigned long int	0 to 4,294,967,295
float	4	none	3.4E +/- 38 (7 digits)
double	8	none	1.7E +/- 308 (15 digits)
long double	10	none	1.2E +/- 4932 (19 digits)

Table 2.1: Allowable Ranges for Data Types, as implemented in Microsoft Visual C/C++ V4.0 for IBM PC Compatible Machines.

\a	alert (bell) character	\\	backslash
\b	backspace	\?	Question mark
\f	formfeed	\’	single quote
\n	newline	\”	double quote
\r	carriage return	*ooo*	octal number
\t	horizontal tab	\x*hh*	hexadecimal number
\v	vertical tab		

Table 2.2: Escape Sequences.

Operators	Associativity
`( )  [ ]  ->  .`	Left to right
`! ~ ++ -- + - * & (type) sizeof`	Right to left
`*  /  %`	Left to right
`+  -`	Left to right
`<<  >>`	Left to right
`<  <=  >  >=`	Left to right
`==  !=`	Left to right
`&`	Left to right
`^`	Left to right
`|`	Left to right
`&&`	Left to right
`||`	Left to right
`?:`	Right to left
`= += *= /= %= &= ^= |= <<= >>=`	Right to left
`,`	Left to right

Table 2.3: Associativity and precedence of operators.

Operators with the highest precedence appear in row 1 of the table. Operators in subsequent rows have decreasing precedence.

Character	Type	Output Format
c	int	Single-byte character.
d	int	Signed decimal integer.
i	int	Signed decimal integer.
o	int	Unsigned octal integer.
u	int	Unsigned decimal integer.
x	int	Unsigned hexadecimal integer, using "abcdef."
X	int	Unsigned hexadecimal integer, using "ABCDEF."
e	double	Signed value having the form [–]*d.dddd* e [*sign*]*ddd* where *d* is a single decimal digit, *dddd* is one or more decimal digits, *ddd* is exactly three decimal digits, and *sign* is + or –.
E	double	Identical to the e format except that E rather than e introduces the exponent.
f	double	Signed value having the form [–]*dddd.dddd*, where *dddd* is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
g	double	Signed value printed in f or e format, whichever is more compact for the given value and precision. The e format is used only when the exponent of the value is less than -4 or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	double	Identical to the g format, except that E, rather than e, introduces the exponent (where appropriate).
n	Pointer to int	Number of characters successfully written so far to the stream or buffer; this value is stored in the integer whose address is given as the argument.
p	Pointer to void	Prints the address pointed to by the argument in the form *xxxx*:*yyyy* where *xxxx* is the segment and *yyyy* is the offset, and the digits *x* and *y* are uppercase hexadecimal digits.
s	String	Character string. Characters are printed up to the first null character or until the *precision* value is reached.

Table 3.1: printf format conversion specifiers.

Character	Type of Input Expected	Type of Argument
c	Single-byte character. White-space characters that are ordinarily skipped are read when c is specified. To read next non-white-space character, use %1s.	Pointer to char.
d	Decimal integer.	Pointer to int.
i	Decimal, hexadecimal, or octal integer.	Pointer to int.
o	Octal integer.	Pointer to int.
u	Unsigned decimal integer.	Pointer to unsigned int.
x	Hexadecimal integer.	Pointer to int.
e, E, f, g, G	Floating-point value consisting of optional sign (+ or –), series of one or more decimal digits containing decimal point, and optional exponent ("e" or "E") followed by an optionally signed integer value.	Pointer to float.
n	No input read from stream or buffer.	Pointer to int, into which is stored number of characters successfully read from stream or buffer up to that point in current call to scanf.
s	String, up to first white-space character (space, tab or newline).	Character array. The array must be large enough for input field plus terminating null character, which is automatically appended.

Table 3.2: scanf format conversion specifiers.

LISTING 4.1: IF-ELSE

Program to read in a character, and classify it as either “lowercase alphabetical”, “digit”, or “other”.

PDL Description:

read in character
IF character is alphabetic and lowercase
THEN
	print “Lower case letter”
ELSE IF character is numeric
	print “Digit”
ELSE
	print “Other symbol”
ENDIF

```
/* Program to classify the character which was typed */

#include <stdio.h>

int main(void)
{
	char key;

	printf("\nEnter a character: ");
	scanf("%c", &key);

	if(key >= 'a' && key <= 'z')
		printf("\nLower case letter\n");
	else if(key >= '0' && key <= '9')
			printf("\nDigit\n");
	else
		printf("\nOther symbol\n");

	return (0);
}
```

LISTING 4.2: SWITCH

Program to determine tomorrow's date, given the date for today.

PDL Description:

read in day and month
SWITCH month TO
	WHEN Apr
	WHEN Jun
	WHEN Sep
	WHEN Nov
		number of days in month is 30
	WHEN Feb
		number of days in month is 28
	OTHERWISE
		number of days in month is 31
ENDSWITCH

```
IF day is last day of month
THEN
        tomorrow is first day of month
        IF month is December
        THEN
                new month is January
        ELSE
                increment month
        ENDIF
ELSE
        increment day
ENDIF
print day and month
```

```
/* Program to determine tomorrow's date, given the date for today. */

#include <stdio.h>

int main(void)
{
      int day, month, days_in_month;

      printf("Enter today's date: ");
      scanf("%d %d", &day, &month);

      switch(month) {
            case 4:           /*Apr, Jun, Sep, Nov */
            case 6:
            case 9:
            case 11:
                  days_in_month = 30;
                  break;
            case 2:           /* Feb */
                  days_in_month = 28;
                  break;
            default:
                  days_in_month = 31;
      }

      if(day == days_in_month){       /* end of month? */
            day = 1;                  /* yes, first day */
            if(month == 12)           /* end of year? */
                  month = 1;          /* yes, January */
            else
                  month++;
      }
      else
            day++;

      printf("Tomorrow's date is %d %d\n", day, month);

      return (0);
}
```

LISTING 4.3: WHILE

Program to read in a sequence of positive floating point numbers and form their sum. The process terminates when the user enters a negative number.

PDL Description:

initialise the sum and data to zero
WHILE data is not negative DO
    add data to the sum
    read next data item
ENDWHILE
print the sum

```
/* Read a sequence of positive floating point numbers and form their
sum */

#include <stdio.h>

int main(void)
{
      float data = 0.0, sum = 0.0;

      while(data >= 0.0){
            sum += data;
            printf("\nEnter next number:  ");
            scanf("%f", &data);
      }

      printf("The sum is %f\n", sum);

      return (0);
}
```

LISTING 4.4: DO-WHILE

Program to read a single, positive integer value and output the digits of that number in reverse sequence, e.g., 1234 becomes 4321.

PDL Description:

read the number
REPEAT
    obtain the rightmost digit
    print the digit
    remove the rightmost digit from number
WHILE number is non-zero

```
/* Read a single, positive integer value and output the digits of
that number in reverse sequence.  */

#include <stdio.h>

int main(void)
{
      int number, digit;

      printf("Enter number:  ");
      scanf("%d", &number);          /* Input data item */

      do {
            digit = number % 10;     /* Get rightmost digit */
            printf("%1d", digit);
            number /= 10;            /* Reduce number */
      } while(number != 0);          /* Repeat until nothing left */

      printf("\n");
      return (0);
}
```

LISTING 4.5: FOR

Program to input 10 floating point values, summing only those values which are positive.

PDL Description:

initialise the running total
FOR 10 times DO
    read the next data value

    IF data item is positive
    THEN
        add data value to running total
    ENDIF

ENDFOR
print total

```
/* Input 10 floating point values, summing only those values which
are positive. */

#include <stdio.h>

int main(void)
{
      float data, sum = 0.0;
      int k;

      for(k = 0; k < 10; k++){
            printf("Enter next number:  ");
            scanf("%f", &data);
            if(data > 0.0)
                  sum += data;
      }

      printf("Sum of positive values is %f\n", sum);

      return (0);
}
```

LISTING 5.1: USING FUNCTIONS

```
/*    Read a time measured in hours, minutes and seconds
      and convert it to the equivalent number of seconds */

#include <stdio.h>

/* function prototype */
int hms_to_seconds(int, int, int);

int main(void)
{
      int hours, minutes, seconds;  /* data values */
      int time;                           /* computed value */

      printf("Enter the time: ");
      scanf("%d %d %d", &hours, &minutes, &seconds);

      /* function call */
      time = hms_to_seconds(hours, minutes, seconds);

      printf("Converted time is %d seconds\n", time);

      return (0);
}


/* Function Definition */
int hms_to_seconds(int hours, int minutes, int seconds)
{
      int t;

      t = (60 * hours + minutes) * 60 + seconds;

      return (t);
}
```

Header File	Types of Operation
assert.h	Diagnostic functions
ctype.h	Functions for testing character data
float.h	Defines constants relating to floating point arithmetic
limits.h	Defines constants for the sizes of integral data types
locale.h	Facilities for localising to a particular country
math.h	Mathematical functions
setjmp.h	Non-local jumps
signal.h	Facilities for handling exceptional conditions arising during execution
stdarg.h	Functions for implementing variable argument lists
stdio.h	File operations, formatted i/o, character i/o
stdlib.h	Various utility functions
string.h	String processing functions
time.h	Functions relating to date and time

Table 6.1: Header Files for Accessing the Standard C Library

LISTING 6.1: C PREPROCESSOR EXAMPLE

```
/* Program to convert a distance in yards, feet
   and inches into the equivalent distance in inches */

#include <stdio.h>

#define DEBUG

#define YARDS_TO_FEET 3
#define FEET_TO_INCHES 12


int distance(const int, const int, const int);

int main(void)
{
      int yards, feet, inches;

      printf("Enter a distance in yards, feet and inches: ");
      scanf("%d %d %d", &yards, &feet, &inches);
      printf("\nThis distance in inches is %d",
      distance(yards, feet, inches) );

      return (0);
}


int distance(const int yards, const int feet, const int inches)
{

#ifdef DEBUG      /* Check if the arguments passed are correct */
      printf("Function 'distance':\n");
      printf("Arguments:  %d yards, %d feet, %d inches\n",
      yards, feet, inches);
#endif

      return((YARDS_TO_FEET * yards + feet) * FEET_TO_INCHES +
      inches);
}
```

LISTING 7.1: USING ARRAYS

```
/* Program to read in two four-element, real vectors,
   and add them together, using array notation only. */

#include <stdio.h>

#define VECSIZE 4       /* Dimension of each vector */

void Read_Vector(float []);
void Add_Vectors(const float [], const float [], float[]);
void Print_Vector(const float []);

int main(void)
{
      float v1[VECSIZE], v2[VECSIZE], v3[VECSIZE];

      printf("\nProgram to add two %d-element vectors.\n",
      (int)VECSIZE);

      printf("\nInput first vector:");
      Read_Vector(v1); /* pass in pointer to 1st element of v1 */
      printf("\nInput second vector:");
      Read_Vector(v2); /*pass in pointer to 1st element of v2 */

      Add_Vectors(v1, v2, v3);

      printf("\nThe sum of these vectors is:");
      Print_Vector(v3);

      return(0);
}

/* Read_Vector:  Reads in a vector with VECSIZE
   elements from the standard input */

void Read_Vector(float v[])
{
      int i;

      for(i = 0; i < VECSIZE; i++)
      {
            printf("\nEnter element %d: ", i+1);
            /* Using array notation, we must provide scanf with
               the address of each element of the array v */
            scanf("%f", &v[i]);
      }
}

/*    Add_Vector: Adds two vectors in the arrays v1 and v2, and
stores the result in a third array, v3. We specify v1 and v2 as
"const float []", because the function to perform addition shouldn't
be allowed to modify the values of the two vectors to be added. */

void Add_Vectors(const float v1[], const float v2[], float v3[])
{
      int i;

      for(i = 0; i < VECSIZE; i++)
            v3[i] = v1[i] + v2[i];
}

/* Print_Vector: Prints out the elements of the vector
   in the array v. */
```

```
void Print_Vector(const float v[])
{
      int i;

      for(i = 0; i < VECSIZE; i++)
            printf("\n%f", v[i]);
}
```

LISTING 7.2: USING POINTERS

```
/* Program to read in two four-element, real vectors,
   and add them together, using array and pointer notation. */

#include <stdio.h>

#define VECSIZE 4       /* Dimension of each vector */

void Read_Vector(float *);
void Add_Vectors(const float *, const float *, float *);
void Print_Vector(const float *);

int main(void)
{
      float v1[VECSIZE], v2[VECSIZE], v3[VECSIZE];

      printf("\nProgram to add two %d-element vectors.\n",
      (int)VECSIZE);

      printf("\nInput first vector:");
      Read_Vector(v1); /* pass in pointer to 1st element of v1 */

      printf("\nInput second vector:");
      Read_Vector(v2); /* pass in pointer to 1st element of v2 */

      Add_Vectors(v1, v2, v3);

      printf("\nThe sum of these vectors is:");
      Print_Vector(v3);

      return (0);
}

/* Read_Vector:  Reads in a vector with VECSIZE
   elements from the standard input */

void Read_Vector(float *v)
{
      int i;

      for(i = 0; i < VECSIZE; i++)
      {
            printf("\nEnter element %d: ", i+1);
            /* Here, we don't need the & with scanf, since the
            pointer v itself stores the address required by scanf */
            scanf("%f", v++);
      }
}

/* Add_Vector: Adds two vectors pointed to by v1 and v2, and
   stores the result in a third vector, pointed to by v3.
   We specify v1 and v2 as "const float *", because the function to
   perform addition shouldn't be allowed to modify the values of the
   two vectors to be added. */
```

```
void Add_Vectors(const float *v1, const float *v2, float *v3)
{
      int i;

      for(i = 0; i < VECSIZE; i++)
            *v3++ = *v1++ + *v2++;
}

/* Print_Vector: Prints out the elements of the vector
   pointed to by v */

void Print_Vector(const float *v)
{
      int i;

      for(i = 0; i < VECSIZE; i++)
            printf("\n%f", *v++);
}
```

LISTING 7.3: BASIC STRING MANIPULATION

```
#include <stdio.h>

int main(void)
{
      char *str = "Hello", *pstr;

      /* print out str */
      printf("%s\n", str);

      /* print out the third character of str */
      printf("%c\n", str[2]);

      /* set fourth character of str to 'w' */
      str[3] = 'w';
      printf("%s\n", str);

      /* Set pstr to point to the same string as str */
      pstr = str;
      printf("%s\n", pstr);

      /* Make pstr point to second character of str */
      pstr++;
      printf("%s\n", pstr);

      /* Make pstr point to the fourth character of str */
      pstr += 2;
      printf("%s\n", pstr);

      return(0);
}
```

The output from the above program would be as follows:

Hello
l
Helwo
elwo
wo

LISTING 7.4: STRING-HANDLING FUNCTIONS - STRING.H

```
#include <string.h>
#include <stdio.h>

#define STRINGSIZE 30

int main(void)
{
      char string1[STRINGSIZE];
      char *string2 = "spider";
      char *string3 = "man";
      char *string4;
      int length, compare;

      /* Find length of string2 */
      length = strlen(string2);
      printf("The length of string2 is %d\n", length);

      /* Compare string2 to string3 */
      if( (compare = strcmp(string2, string3)) < 0)
            printf("string2 is less than string3\n");
      else if(compare == 0)
            printf("string2 and string3 are the same\n");
      else
            printf("string2 is greater than string3\n");

      /* Copy string2 into string1.  Note that although strcpy
         returns a pointer to string1, we don't have to make use
         of it, if we don't need to */
      strcpy(string1, string2);
      printf("string1 is now %s\n", string1);

      /* Concatenate string1 and string3. Assign result to string4 */

      string4 = strcat(string1, string3);
      printf("string4 is %s\n", string4);
      printf("string1 is %s\n", string1);

      return(0);
}
```

The output of this program is:

The length of string2 is 6
string2 is greater than string3
string1 is now spider
string4 is spiderman
string1 is spiderman

LISTING 7.5: SPRINTF AND SSCANF

```
#include <stdio.h>

#define STRINGSIZE 10

int main(void)
{
      char *text = "4 by 4.0";
      char string[STRINGSIZE];
      int num1;
      float num2;

      /* Parse text, looking for an integer, followed by
         a string followed by a float */
      sscanf(text, "%d %s %f", &num1, string, &num2);

      /* Check for the correct assignments */
      printf("num1 = %d\n", num1);
      printf("num2 = %f\n", num2);
      printf("string = %s\n", string);

      num1 = 10;
      string[0]='t'; string[1]='e'; string[2]='n'; string[3]='\0';
      num2 = 20.0F;

      /* Prepare an output string, using text */
      sprintf(text, "%d %s %5.1f", num1, string, num2);
      printf("%s\n", text);

      return(0);
}
```

The output of the above program is:
num1 = 4
num2 = 4.000000
string = by
10 ten 20.0

LISTING 7.6: ATOI EXAMPLE

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

#define STRINGSIZE 10

int main(void)
{
      char *text = "Joe 90";
      char dummy[STRINGSIZE], digits[STRINGSIZE];
      int number;

      /* Use sscanf to extract digits */
      sscanf(text, "%s %s", dummy, digits);
      /* convert the string pointed to by digits to an integer */
      number = atoi(digits);
      printf("The value of number is: %d\n", number);

      return(0);
}
```

The output of this program is:
The value of number is: 90

LISTING 8.1: USING TYPEDEF AND ENUM

```
/* Compute the week's pay for an hourly paid employee.
   Overtime is applied for weekend working */

#include <stdio.h>

#define SATURDAYADJUST 1.5F
#define SUNDAYADJUST 2.0F

typedef enum {SUN, MON, TUE, WED, THU, FRI, SAT} Weekday;
typedef int Hours;
typedef float Rates;
typedef float Wages;

Weekday tomorrow(const Weekday);

int main(void)
{
      Hours hours;
      Rates baserate, rate;
      Wages wages;
      Weekday day;

      printf("Enter the basic hourly rate: ");
      scanf("%f", &baserate);
      wages = 0.0F;
      printf("Enter the hours worked\n(Monday through Sunday):\n");

      day = SUN;
      do{
            day = tomorrow(day);
            scanf("%d", &hours);

            switch(day){
            case MON: case TUE: case WED: case THU: case FRI:
                  rate = baserate; break;
            case SAT:
                  rate = SATURDAYADJUST * baserate; break;
            case SUN:
                  rate = SUNDAYADJUST * baserate; break;
            }
            wages += rate * hours;
      } while(day != SUN);

      printf("Total wages for the week: %8.2f\n", wages);

      return(0);
}

Weekday tomorrow(const Weekday day)
{
      switch(day){
      case SUN: return MON;
      case MON: return TUE;
      case TUE: return WED;
      case WED: return THU;
      case THU: return FRI;
      case FRI: return SAT;
      case SAT: return SUN;
      }
}
```

LISTING 9.1: USING STRUCTURES TO MANIPULATE COMPLEX NUMBERS.

```
/* Program to manipulate complex numbers, using structures.  The
   structured variables are passed into the functions by value. */

#include <stdio.h>

/* Some sample data*/
#define REAL1 2.0
#define IMAG1 4.0
#define REAL2 1.5
#define IMAG2 -3.5

/* Set up familiar complex number notation */
#define RE(z) ( (z).real )
#define IM(z) ( (z).imag )

/* A structure to represent complex numbers */
struct complex {
      double real;
      double imag;
};

/* Create a new type name, for declaring complex variables */
typedef struct complex Complex;

Complex add(const Complex, const Complex);
Complex multiply(const Complex, const Complex);
Complex conjugate(const Complex);

int main(void)
{
      Complex z1, z2, z3;

      /* Initialise z1 and z2 with sample data */
      RE(z1) = REAL1; IM(z1) = IMAG1;
      RE(z2) = REAL2; IM(z2) = IMAG2;

      /* The '+' in the conversion spec means a sign is always
      printed */

      printf("z1 = %.2f%+.2fi\n", RE(z1), IM(z1));
      printf("z2 = %.2f%+.2fi\n", RE(z2), IM(z2));

      /* Add z1 and z2 */
      z3 = add(z1, z2);
      printf("z1 + z2 = %.2f%+.2fi\n", RE(z3), IM(z3));

      /* Multiply z1 and z2 */
      z3 = multiply(z1, z2);
      printf("z1 * z2 = %.2f%+.2fi\n", RE(z3), IM(z3));

      /* Form complex conjugate of z1 */
      z3 = conjugate(z1);
      printf("z1* = %.2f%+.2fi\n", RE(z3), IM(z3));

      return(0);
}

Complex add(const Complex z1, const Complex z2)
{
      Complex z3;

      RE(z3) = RE(z1) + RE(z2);
      IM(z3) = IM(z1) + IM(z2);
```

```
      return (z3);
}

Complex multiply(const Complex z1, const Complex z2)
{
      Complex z3;

      RE(z3) = RE(z1) * RE(z2) - IM(z1) * IM(z2);
      IM(z3) = RE(z1) * IM(z2) + IM(z1) * RE(z2);

      return (z3);
}

Complex conjugate(const Complex z1)
{
      Complex z2;

      RE(z2) = RE(z1);
      IM(z2) = -IM(z1);

      return (z2);
}
```

The output from this program is:

z1 = 2.00+4.00i
z2 = 1.50-3.50i
z1 + z2 = 3.50+0.50i
z1 * z2 = 17.00 - 1.00i
z1* = 2.00 - 4.00i

LISTING 9.2: USE OF POINTERS TO STRUCTURES

```
/* Program to illustrate use of pointers to structures.  This code
   might represent part of a graphics package.  The user is asked to
   enter the coordinates of a viewport (a rectangular window onto
   some portion of the computer screen).  The user can then input
   points, and the program will determine whether a point lies in the
   viewport or not. */

#include <stdio.h>

/* Structure to represent a point. x and y are the (x,y) coordinates
   of the point */

struct point{
      int x;
      int y;
};

/* Create a new type name for a point */
typedef struct point Point;

/* Structure to represent a rectangle.  The rectangle is fully
   defined by two structures of type Point, corresponding to the top
   left, and bottom right corners respectively */
struct rect{
      Point tl;
      Point br;
};

/* Create a new type name for a rectangle */
typedef struct rect Rect;
```

```
typedef enum {FALSE, TRUE} Boolean;

void makepoint(Point *const);
Boolean validrect(Rect *const);
Boolean isinrect(Rect *const, Point *const);

int main(void)
{
      Point pt;
      Rect viewport;

      /*    Obtain the coordinates of the viewport rectangle,
            and check that the rectangle specified is valid */
      do{
            printf("\nEnter the top left viewport coordinates:  \n");

            /* &viewport.tl is a pointer to the member variable tl of
               viewport. tl is itself a structure of type Point, so
               &viewport.tl is a pointer to a structure of type Point
            */
            makepoint(&viewport.tl);
            printf("\nEnter the bottom right viewport coordinates:
\n");
            makepoint(&viewport.br);
      } while(!validrect(&viewport));


      printf("\nEnter a point:  \n");

      /* Test whether a point lies within the viewport */
      do{
            /* &pt is a pointer to a structure of type Point */
            makepoint(&pt);

            if( !isinrect(&viewport, &pt) )
                  printf("This point lies outside the viewable
                  area.\n");
            else
                  printf("This point lies within the viewable
                  area.\n");

            printf("\nEnter another point? ('y' for yes)\n");

            getchar(); /* This is a dummy read to flush the '\n'
                          remaining from the previous scanf from the
                          buffer */
      } while(getchar() == 'y');

      return(0);
}

/* makepoint reads in the (x,y) coords of a point from the keyboard.
   The argument 'Point *const p' prevents the function from
   accidentally modifying the address to which the pointer p points.
*/

void makepoint(Point *const p)
{
      printf("x-coordinate:  ");

      /* p->x accesses the member variable x of the structure
         pointed to by p. &p->x is the address of this variable */
      scanf("%d", &p->x);

      printf("y-coordinate:  ");
```

```
      scanf("%d", &p->y);
}

/* validrect checks that a valid rectangle has been entered, i.e.
   that the top left coords are less than the bottom left coords. It
   returns TRUE if the rectangle is valid, otherwise FALSE. */

Boolean validrect(Rect *const r)
{
      /* r->tl accesses the member variable tl of the structure
         pointed to by r.  Since tl is itself a structure, r->tl.x
         accesses the member variable x of the structure tl,
         contained in the structure r. */
      if( r->tl.x >= r->br.x || r->tl.y >= r->br.y ){
            printf("\nInvalid rectangle entered.\n");
            return (FALSE);
      }

      return (TRUE);
}

/* isinrect determines whether the point p lies inside or outside
   the rectangle r. */

Boolean isinrect(Rect *const r, Point *const p)
{
      if( p->x > r->tl.x && p->x < r->br.x
              && p->y > r->tl.y && p->y < r->br.y )
            return (TRUE);

      return(FALSE);
}
```

LISTING 10.1: FGETC, FPUTC, EOF AND FEOF

```
/* Copy the contents of a source file to a destination file, using
   fgetc and fputc. */

#include <stdio.h>
#include <stdlib.h>

#define READONLY "r"
#define WRITEONLY "w"

/* We read and write characters as int's, because of the need to
accommodate the special value EOF */
typedef int Character;

int main(void)
{
      FILE *ip_file, *op_file;
      /* FILENAME_MAX, defined in stdio.h, is the maximum file name
         length permitted by the operating system */
      char filename[FILENAME_MAX + 1];
      Character c;

      printf("\nEnter input file name: ");
      scanf("%s", filename);

      /* Open the input file for reading only */
      if( (ip_file = fopen(filename, READONLY)) == NULL){
            printf("\nmain: failed to open file %s.", filename);
            exit(EXIT_FAILURE);
      }

      printf("\nEnter output file name: ");
      scanf("%s", filename);

      /* Open the output file for writing only */
      if( (op_file = fopen(filename, WRITEONLY)) == NULL){
            printf("\nmain: failed to open file %s.", filename);
            exit(EXIT_FAILURE);
      }

      /* Read data, one character at a time, from the i/p file, until
         EOF is encountered */
      while( (c = fgetc(ip_file)) != EOF ){
      /* Write data to o/p file.  If EOF arises during the write, an
         error has occurred */
            if(fputc(c, op_file) == EOF){
                  printf("\nmain: error writing to output file");
                  break;
            }
      }

      /* If EOF was encountered in the i/p file before the end of the
         file was reached, an error has occurred */
      if( !feof(ip_file) )
            printf("\nmain:  Error reading from input file");

      /* Close the i/p and o/p files before exiting */
      fclose(ip_file);
      fclose(op_file);
      return(0);
}
```

LISTING 10.2: USING FGETS AND FPUTS

```
/* Program to compare two files, line by line, print the line number
   in the files where they first differ, and write the corresponding
   line in each file to an output file */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define READONLY "r"
#define WRITEONLY "w"

#define MAXLINE 256 /* Maximum line length to be read */
#define IDENTICAL 0
#define FIRSTLINE 1

void comparefiles(FILE *, FILE *, FILE *);

int main(void)
{
      FILE *ip1, *ip2, *op;
      char filename[FILENAME_MAX + 1];

      printf("\nEnter first input file name: ");
      scanf("%s", filename);

      /* Open the first input file for reading only */
      if( (ip1 = fopen(filename, READONLY)) == NULL){
            printf("\nmain: failed to open file %s.", filename);
            exit(EXIT_FAILURE);
      }

      printf("\nEnter second input file name: ");
      scanf("%s", filename);

      /* Open the second input file for reading only */
      if( (ip2 = fopen(filename, READONLY)) == NULL){
            printf("\nmain: failed to open file %s.", filename);
            exit(EXIT_FAILURE);
      }

      printf("\nEnter output file name: ");
      scanf("%s", filename);

      /* Open the output file for writing only */
      if( (op = fopen(filename, WRITEONLY)) == NULL){
            printf("\nmain: failed to open file %s.", filename);
            exit(EXIT_FAILURE);
      }

      comparefiles(ip1, ip2, op);

      fclose(ip1); fclose(ip2); fclose(op);

      return(0);
}

void comparefiles(FILE *ip1, FILE *ip2, FILE *op)
{
      char line1[MAXLINE], line2[MAXLINE];
      unsigned int line_num = FIRSTLINE;

      /* Read a line of data from each input file */
      while(fgets(line1, (int)MAXLINE, ip1) != NULL &&
```

```
            fgets(line2, (int)MAXLINE, ip2) != NULL)
      {
            /* if lines are identical, increment line count and read
               next lines */
            if(strcmp(line1, line2) == IDENTICAL)
                  line_num++;
            /* otherwise, print the line number where they first
               differ, and write the corresponding lines to the
               output file */
            else{
                  printf("\nFirst difference found at line %u",
                  line_num);
                  if(fputs(line1, op) == EOF || fputs(line2, op) ==
                  EOF){
                        printf("\ncomparefiles: error writing to o/p
                        file.");
                        exit(EXIT_FAILURE);
                  }
                  return;
            }
      }
      /* Print warning if one file has fewer lines than the other */
      printf("\nRan out of input at line %u.", line_num);
}
```

LISTING 10.3: FORMATTED I/O WITH FSCANF AND FPRINTF

```
/* Program to copy a series of integers from a source file
   to a destination file.  The copy process terminates when
   EOF is encountered in the source file */

#include <stdio.h>
#include <stdlib.h>

#define READONLY "r"
#define WRITEONLY "w"

typedef int Character;

int main(void)
{
      FILE *ip_file, *op_file;
      char filename[FILENAME_MAX + 1];
      int data, status;

      printf("\nEnter input file name: ");
      scanf("%s", filename);

      /* Open the input file for reading only */
      if( (ip_file = fopen(filename, READONLY)) == NULL){
            printf("\nmain: failed to open file %s.", filename);
            exit(EXIT_FAILURE);
      }

      printf("\nEnter output file name: ");
      scanf("%s", filename);

      /* Open the output file for writing only */
      if( (op_file = fopen(filename, WRITEONLY)) == NULL){
            printf("\nmain: failed to open file %s.", filename);
            exit(EXIT_FAILURE);
      }

      /* fscanf will attempt to assign data read from the input
         file to an integer */
```

```
	while((status = fscanf(ip_file, "%d", &data)) != EOF){
	/* If the number of assignments isn't 1, an error has occurred
	*/
		if(status != 1){
			printf("\nmain: source file read failure.");
			exit(EXIT_FAILURE);
		}
	/* If fprintf fails, it returns a negative value */
		if(fprintf(op_file, "%d\n", data) < 0 ){
			printf("\nmain: destination file write failure.");
			exit(EXIT_FAILURE);
		}
	}

	fclose(ip_file);
	fclose(op_file);

	return(0);
}
```

LISTING 11.1: DYNAMIC MEMORY ALLOCATION

```
/* Program to read in two vectors from a file, and calculate their
   dot product, displaying the result on the screen. In the input
   file, the vectors MUST be preceded by the number of elements in
   each vector, so that we can can set aside the correct amount of
   storage */

#include <stdlib.h>
#include <stdio.h>

#define READONLY "r"
#define MAX_ELEMENTS 200

double dot_product(const double [], const double [], const int);

int main(void)
{
      FILE *ipfile;
      char filename[FILENAME_MAX + 1];
      double *vect1, *vect2, product;
      int elements, i;

      /* Obtain the input filename from the user */
      printf("Enter the name of the input file:\n");
      scanf("%s", filename);

      /* Attempt to open the specified file */
      if((ipfile = fopen(filename, READONLY)) == NULL){
            fprintf(stderr, "Couldn't open %s.\n", filename);
            exit(EXIT_FAILURE);
      }

      /* Attempt to read the number of elements in the vectors (must
         be the same for each vector). */
      if(fscanf(ipfile, "%d", &elements) != 1){
            fprintf(stderr, "Couldn't read vector sizes from %s.\n",
            filename);
            exit(EXIT_FAILURE);
      }
      else if(elements > MAX_ELEMENTS){
            fprintf(stderr, "Vector size exceeds max. no. of
            elements.\n");
            exit(EXIT_FAILURE);
      }

      /* Attempt to dynamically allocate arrays with the required
         number of elements to the pointers vect1 and vect2 */
      if((vect1 = (double *)malloc(elements * sizeof(double))) ==
      NULL || (vect2 = (double *)malloc(elements * sizeof(double)))
      == NULL){
            fprintf(stderr, "Failed to allocate memory.\n");
            exit(EXIT_FAILURE);
      }

      /* Attempt to read the elements of the first vector from the
         file */
      for(i = 0; i < elements; i++){
            if(!fscanf(ipfile, "%lf", &vect1[i])){
                  fprintf(stderr, "Error reading element %d of vector
                  1\n", i+1);
                  exit(EXIT_FAILURE);
            }
      }
```

```
      /* Attempt to read the elements of the second vector from the
         file */
      for(i = 0; i < elements; i++){
            if(!fscanf(ipfile, "%lf", &vect2[i])){
                  fprintf(stderr, "Error reading element %d of vector
                  2\n", i+1);
                  exit(EXIT_FAILURE);
            }
      }


      product = dot_product(vect1, vect2, elements);
      printf("The dot product of the input vectors is: %14.6f\n",
      product);

      fclose(ipfile);

      /* Release the storage allocated to vect1 and vect2 */
      free(vect1);
      free(vect2);

      return(EXIT_SUCCESS);
}


/* dot_product calculates the scalar product of the two vectors
   stored in the arrays v1 and v2 */
double dot_product(const double v1[], const double v2[], int
elements)
{
      double sum = 0.0;
      int i;

      for(i = 0; i < elements; i++)
            sum += v1[i] * v2[i];

      return (sum);
}
```

LISTING 11.2: ADVANCED EXAMPLE OF DYNAMIC MEMORY ALLOCATION

In this example, we show how to dynamically allocate memory for a 2-D array, using `malloc`. First, we need to understand how a 2-D array is stored in the computer's memory, and how 2-D array notation is handled by the compiler. Consider the following declaration of a 3x3 array of integers called `table`:

```
int table[3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

The compiler first sets aside storage for an 1-D array, `table[3]`. Each element of this array is a pointer to a second 1-D array, whose elements constitute a single row of the 2-D array. The situation is depicted in figure 11.1 below:

Figure 11.1: Storage of a 2-D Array.

Since the array name `table` is (by definition) a pointer to the start of the first array, and this array consists of a set of pointers to the actual row data, `table` is correctly termed a "pointer to a pointer".

Now, say we want to allocate storage to a 2-D array of integers, whose size is to be determined during program execution. The first thing we need to do is to set up a "pointer to a pointer", using the following notation:

```
int **table;
```

Now, say that during execution, the user enters the number of rows and columns required, which are stored in the integer variables `nrows` and `ncols` respectively. The next step is to allocate sufficient space for an array of pointers of dimension `nrows`, each element of which will point to a row of `table`:

```
table = (int **)malloc(nrows * sizeof(int *));
```

Finally, to each of these row pointers, we need to allocate an array of integers of dimension `ncols`, to store the actual row data of the 2-D array. This we do by looping through the row pointers allocated above, using `malloc` to allocate the required storage to each one:

```
for(i = 0; i < nrows; i++)
      table[i] = (int *)malloc(ncols * sizeof(int));
```

Now we can access the elements of the 2-D array pointed to by `table` using the familiar array notation - e.g. `table[2][1]` accesses the element in the third row, and second column of the array. The program below illustrates how the preceding would be implemented in practice.

```
/* Program illustrating how to use malloc to set aside storage for a
   2-D array of integers read in from a file.  The first two data
   items in the file MUST give the number of rows and columns in the
   array. */

#include <stdlib.h>
#include <stdio.h>

#define READONLY "r"

void read_table(FILE *, int **const, const int, const int);
void print_table(int **const, const int, const int);

int main(void)
{
      FILE *ipfile;
      char filename[FILENAME_MAX + 1];
      int **table;      /* table is a pointer to the 2-D array */
      int nrows, ncols; /* no. of rows and cols in the array */
      int i;

      /* Ask user for input file name */
      printf("Enter input file name:\n");
      scanf("%s", filename);

      /* Attempt to open specified input file */
      if((ipfile = fopen(filename, READONLY)) == NULL){
            fprintf(stderr, "Couldn't open file %s\n", filename);
            exit(EXIT_FAILURE);
      }

      /* Read the array dimensions from the file */
      if(fscanf(ipfile, "%d %d", &nrows, &ncols) != 2){
            fprintf(stderr, "Couldn't read array dimensions from
            %s\n", filename);
            exit(EXIT_FAILURE);
      }

/* Dynamically allocate storage for an array of pointers to the table
   rows. table will point to the start of this array of pointers */
      if((table = (int **)malloc(nrows * sizeof(int *))) == NULL){
            fprintf(stderr, "Failed to allocate memory\n");
            exit(EXIT_FAILURE);
      }

/* Cycle through the array of row pointers.  Assign to each row
   pointer storage for a single row of integers.  (There are ncols
   integers in each row) */
      for(i = 0; i < nrows; i++){
            if((table[i] = (int *)malloc(ncols * sizeof(int))) ==
            NULL){
                  fprintf(stderr, "Failed to allocate memory\n");
                  exit(EXIT_FAILURE);
            }
      }

      read_table(ipfile, table, nrows, ncols);
      print_table(table, nrows, ncols);

      /* Free the memory allocated to each row pointer */
      for(i = 0; i < nrows; i++)
            free(table[i]);

      /* Free the memory allocated to store the row pointers */
      free(table);
```

```
      fclose(ipfile);

      return(EXIT_SUCCESS);
}

/* print_table prints out the elements of the 2-D array.  We pass in
   the start of the array as a pointer to a pointer.  This allows us
   to use array notation within the function body. */
void print_table(int **const table, const int nrows, const int ncols)
{
      int i, j;

      for(i = 0; i < nrows; i++){
            for(j = 0; j < ncols; j++){
                  printf("%5d ", table[i][j]);
            }
            printf("\n");
      }
}

/* read_table reads in the elements of the matrix from the input file
   and stores them in the memory allocated to table. */
void read_table(FILE *ipfile, int **const table, const int nrows,
const int ncols)
{
      int i, j;

      for(i = 0; i < nrows; i++){
            for(j = 0; j < ncols; j++){
                  if(!fscanf(ipfile, "%d", &table[i][j])){
                        fprintf(stderr, "Failed to read element (%d,
                        %d) of table\n", i+1, j+1);
                        exit(EXIT_FAILURE);
                  }
            }
      }

}
```

LISTING 11.3: COMMAND LINE ARGUMENTS

```
/* sum.c - Program to add two numbers specified as command line
   arguments, and to store the result in a file, also
   specified on the command line */

#include <stdlib.h>
#include <stdio.h>

#define WRITEONLY "w"

int main(int argc, const char *const argv[])
{
      long int first, second;
      FILE *opfile;
      char *check;

      /* Check that the correct number of CLA's have been typed */
      if(argc != 4){
            fprintf(stderr, "Usage:  %s integer1 integer2 outfile\n",
            argv[0]);
            exit(EXIT_FAILURE);
```

```
        }

        /* Attempt to assign argv[1] to an integer, using scanf. We use
           the char 'check' to ensure that an erroneous character
           wasn't typed on the command line, between the two integer
           values */

        if(sscanf(argv[1], "%ld%c", &first, &check) != 1){
              fprintf(stderr, "Invalid parameter: %s\n", argv[1]);
              exit(EXIT_FAILURE);
        }

        if(sscanf(argv[2], "%ld%c", &second, &check) != 1){
              fprintf(stderr, "Invalid parameter: %s\n", argv[2]);
              exit(EXIT_FAILURE);
        }

        /* Open the output file specified by argv[3] */
        if((opfile = fopen(argv[3], WRITEONLY)) == NULL){
              fprintf(stderr, "Failed to open output file %s\n",
              argv[3]);
              exit(EXIT_FAILURE);
        }

        fprintf(opfile, "%ld + %ld = %ld\n", first, second, first +
        second);

        fclose(opfile);

        /* Indicate that the program has terminated successfully*/
        return(EXIT_SUCCESS);
}
```

Sample (erroneous) command lines:

c:\progs> sum.exe 123 456
Usage: sum.exe integer1 integer2 outfile

c:\progs> sum.exe 123x 456 out.dat
Invalid parameter: 123x

LISTING 11.4: POINTERS TO FUNCTIONS

```
/* Program to demonstrate the use of pointers to functions. A
   series of values are tabulated for a given function. The
   tabulation function receives a pointer to this function as one
   of its arguments */

#include <stdio.h>
#include <stdlib.h>

/* Create new type name 'PFunction' for a pointer to a function
   taking a single argument of type const double, and with a
   return value of type double */
typedef double (*PFunction)(const double);

double square(const double);
double cube(const double);
void tabulate(PFunction, const double, const double, const double,
char *);

int main(void)
{
      PFunction fnptr;  /* Declare a pointer of type PFunction */
      double lower, upper, inc; /* lower & upper bounds, increment */

      /* fflush (stdio.h) empties the file stream specified as its
         argument.  This prevents scanf from re-reading the previous
         value entered */
      printf("Enter the lower bound for the table: ");
      while(!scanf("%lf", &lower)){
            fflush(stdin);
            printf("Invalid value.  Please re-enter.\n");
      }

      printf("Enter the upper bound for the table: ");
      while(!scanf("%lf", &upper) && upper < lower){
            fflush(stdin);
            printf("Invalid value.  Please re-enter.\n");
      }

      printf("Enter the increment for the table: ");
      while(!scanf("%lf", &inc)){
            fflush(stdin);
            printf("Invalid value.  Please re-enter.\n");
      }

      fnptr = square; /* fnptr points to the function 'square' */
      tabulate(fnptr, lower, upper, inc, "square");
      fnptr = cube; /* fnptr points to the function 'cube' */
      tabulate(fnptr, lower, upper, inc,  "cube");

      return (EXIT_SUCCESS);
}


/* tabulate prints a table of values, calculated by the function
   pointed to by the argument 'f'. */
void tabulate(PFunction f, const double lower, const double upper,
const double inc, char *operation)
{
      double x;

      printf("The %s of x, from x = %.2f to x = %.2f "
               "in steps of %.2f\n\n", operation, lower, upper, inc);
```

```
	for(x = lower; x <= upper; x += inc)
		printf("%10.2f %10.2f\n", x, f(x));

	printf("\n");
}

double square(const double x)
{
	return (x * x);
}

double cube(const double x)
{
	return (x * x * x);
}
```

LISTING 12.1: STATIC VARIABLES

```
/* Static variables declared within functions are intialised once at
   compile time, and retain their value across function calls.
   Automatic variables, on the other hand, are initialised each time
   the function is entered */

#include <stdio.h>

void function(void);

int main(void)
{
      int i;    /* loop index */

      for(i = 0; i < 5; i++)
            function();

      return(0);
}

void function(void)
{
      static int static_var = 1; /* initialisation performed once */
      int auto_var = 1;        /* performed for each function call */

      printf("automatic variable = %d, static variable = %d\n",
      auto_var, static_var);

      auto_var++; /* redundant operation */
      static_var++; /* carried forward to next function call */
}
```

The output of this program is:

automatic variable = 1, static variable = 1
automatic variable = 1, static variable = 2
automatic variable = 1, static variable = 3
automatic variable = 1, static variable = 4
automatic variable = 1, static variable = 5

LISTING 12.2: EXTERNAL REFERENCING DECLARATIONS

This program calculates the difference between two times expressed in 24 - hour format. The program is implemented in two files. The “front-end”, which reads input data, and writes output data, is contained in the file **timediff.c**. The functions to manipulate the times are contained in the file **time.c**. We make the functions in **time.c** available in **timediff.c** by using external referencing declarations.

File “timediff.c”

```
/* File: timediff.c

   Determine the difference between two 24-hour clock times. The
   input data and the output are all expressed in the same 24-hour
   format */

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

```
/* External referencing declarations.  These allow the functions
   contained in the file time.c to be called from this file */
extern long int hms_to_time(const int, const int, const int);
extern void time_to_hms(const long int, int *const, int *const,
                                int *const);

int main(void)
{
      int hours1, mins1, secs1;     /* 1st time */
      int hours2, mins2, secs2;     /* 2nd time */
      int hours, mins, secs;        /* the result */
      long int time1, time2; /* conversions */

      printf("Enter the first time (hrs mins secs): ");
      scanf("%d %d %d", &hours1, &mins1, &secs1);

      printf("Enter the second time (hrs mins secs): ");
      scanf("%d %d %d", &hours2, &mins2, &secs2);

      time1 = hms_to_time(hours1, mins1, secs1);
      time2 = hms_to_time(hours2, mins2, secs2);

      /* The function 'labs' calculates the absolute value of a long
         integer, and is accessed via stdlib.h */
      time_to_hms(labs(time1 - time2), &hours, &mins, &secs);
      printf("The time difference is %d %d %d\n", hours, mins, secs);

      return(EXIT_SUCCESS);
}
```

File "time.c"

```
/* File: time.c

   This file contains two functions to convert between a time
   measured in a total number of seconds and a time expressed in
   hours, minutes and seconds.  Each function is the complement
   of the other.
*/

#include <stdio.h>

#define SECS_IN_MIN 60
#define MINS_IN_HOUR 60

void time_to_hms(const long int t, int *const h, int *const m,
                 int *const s)
{
      int mins;

      mins = (int)(t / SECS_IN_MIN);
      *s = (int)(t % SECS_IN_MIN);
      *m = mins % MINS_IN_HOUR;
      *h = mins / MINS_IN_HOUR;
}

long int hms_to_time(const int h, const int m, const int s)
{
      return ( ((long)h * MINS_IN_HOUR + m) * SECS_IN_MIN + s );
}
```

LISTING 12.3: HEADER FILES

This program is the same as that in the previous listing, except that external referencing declarations are contained in a header file, **time.h**. Note that, in a project with several source files, the same header file should only be included once. We use preprocessor directives to ensure that this is the case.

File "time.h"

```
/* File: time.h

  Header file providing prototypes for the two time conversion
  functions, along with macros.
*/


/* In order to prevent the header file from being included more
   than once by different files in the project, we use the #ifndef
   - #define - #endif preprocessor directives */

#ifndef _TIME_H  /* Has the token _TIME_H already been defined? */

/* If not, this is the first time this file has been included, so
   define it now. Now, the function prototypes and macros which
   follow will not be included a second time. */
#define _TIME_H

#define SECS_IN_MIN 60
#define MINS_IN_HOUR 60

extern long int hms_to_time(const int, const int, const int);
extern void time_to_hms(const long int, int *const, int *const,
                                int *const);

#endif
```

File "timediff.c"

```
/* File: timediff.c

   Determine the difference between two 24-hour clock times. The
   input data and the output are all expressed in the same 24-hour
   format */

#include <stdio.h>
#include <stdlib.h>

#include "time.h"

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

File "time.c"

```
/* File: time.c

   This file contains two functions to convert between a time
   measured in a total number of seconds and a time expressed in
   hours, minutes and seconds.  Each function is the complement
   of the other.
```

```
*/

#include <stdio.h>
#include "time.h"

void time_to_hms(const long int t, int *const h, int *const m,
                                                    int *const s)
{
      int mins;
            .
            .
            .
```

LISTING 12.4: ABSTRACT DATA TYPES

This program implements a very basic ATM machine. A customer can display their account balance, or make a withdrawal. The program consists of 3 files: **account.c** implements an ADT to store details of a customer's account. The ADT is based on a data structure called `account`. Private and public functions to manipulate the `account` structure are implemented. **account.h** is a header file which specifies the public interface to the `account` structure. It is only via the functions specified in this file that any other file can access the account structure. **ATM.c** implements the user front-end to the ATM machine. It accesses the `account` data structure by including the file **account.h**.

The program asks the customer to input a PIN number. It then searches for the customer's PIN in a database file. If the PIN number is found, then the account details for that customer are read from the file into the `account` data structure. A sample input file is shown below:

Sample Account Database File

```
(Pin)   (A/C No.)    (Balance)
1000 383211     840.00
1001 441922     445.00
1002 515737     220.00
1003 651423      70.00
```

account.h - specifies the public interface to the ADT

```
/* File:  account.h
   This header file contains the public interface to the 'account'
   abstract data type. Other files can manipulate the account data
   structure ONLY via the functions provided below.
*/

#ifndef _ACCOUNT_H
      #define _ACCOUNT_H

      /* Indicators of the machine's status */
      typedef enum {FAULT, BADPIN, OK} Status;

      /* The public functions of the ADT */
      extern Status account_open_database(void);
      extern void account_close_database(void);
      extern Status account_make_withdrawal(void);
      extern void account_display_balance(void);
      extern Status account_next_customer(void);

#endif
```

account.c - implementation of the ADT

```
/* File:  account.c

This file implements an abtract data type called 'account',
as part of a simple ATM system.

The customer's account details are read from a database file called
'accounts.dat'.  A customer record consists of a PIN number, account
number and account balance.
*/

#include <stdio.h>
#include <stdlib.h>

/* 'account.h' specifies the interface to the account ADT */
#include "account.h"

#define MAXLINE 80 /* Maximum line length to be read from file */
#define LIMIT 200 /* Maximum single withdrawal from account */

/* Create new type name for PIN numbers */
typedef unsigned short int Pin;

/* Create new type name for account numbers */
typedef  unsigned long int AC_Num;

/* The data structure account is used to store a customer's account
   details */
struct account{
      FILE *ac_file;      /* File pointer to account database */
      fpos_t file_pos;    /* Records current position in database
                             file. fpos_t ('file position type') is
                             defined in stdio.h */
      Pin pin;            /* Customer's PIN number */
      AC_Num ac_num;      /* Customer's account number */
      float balance;      /* Customer's account balance */
};

/* Create new type name for an account structure */
typedef struct account Account;

/* Private functions to manipulate the account data structure */
static Status account_update_database(void);
static Status account_enter_pin(void);

/* Declare a global account structure, using the storage class
   specifier static to ensure its privacy */
static Account your_ac;

/* ------------------------------------------------------------
      Implementation of public functions, in the ADT's interface.
   ------------------------------------------------------------*/

/* account_open_database sets up the FILE pointer to the accounts
   database file. The access mode "r+" specifies that the file is
   opened for reading and writing. */

Status account_open_database(void)
{
      if((your_ac.ac_file = fopen("accounts.dat", "r+")) == NULL)
            return (FAULT);

      return (OK);
}
```

```
/* account_close_database:  Close the file stream */

void account_close_database(void)
{
      fclose(your_ac.ac_file);
}

/* account_next_customer:  Get ready for the next customer. */

Status account_next_customer(void)
{
      Status status;

      printf("\nWelcome to GreedyBank Ltd.\n");
      /* rewind [stdio.h] resets the file pointer to the start of the
         specified file */
      rewind(your_ac.ac_file);

      /* Verify the customer's PIN number */
      status = account_enter_pin();

      return (status);
}

/* account_make_withdrawal:  Allow the customer to make
   a withdrawal from his / her account */

Status account_make_withdrawal(void)
{
      unsigned short int amount;
      Status status;

      do{
            do{
                  printf("\nEnter amount to withdraw, in multiples of
                  $5:\n");
                  while(!scanf("%u", &amount)){
                        fprintf(stderr, "Invalid entry: please re-
                        enter\n");
                        fflush(stdin);
                  }
            } while(amount % 5 != 0); /* Check for multiple of 5 */

            /* Is the amount over the limit? */
            if(amount > LIMIT){
                  printf("Maximum withdrawal $200\n");
                  continue; /* Go back to the top of the loop */
            }

            /* Has the customer enough money? */
            else if(your_ac.balance - (float)amount < 0.0){
                  printf("Insufficient funds\n");
                  continue; /* Go back to the top of the loop */
            }

            /* Otherwise, the amount is OK, break out of loop */
            else break;

      } while(OK); /* Infinite loop */

      your_ac.balance -= (float) amount; /* adjust the balance */

      /* update the customer's details in the database file */
      status = account_update_database();
      return (status);
}
```

```
/* account_display_balance: display the customer's balance
   on the screen */

void account_display_balance(void)
{
      printf("\nAccount Number: %u\n", your_ac.ac_num);
      printf("Balance: $%-10.2f\n", your_ac.balance);
}

/* ------------------------------------------------------------
   Functions private to the ADT. Privacy in ensured using the
   keyword 'static'.
   ------------------------------------------------------------ */

/* account_enter_pin: validate the customer's pin number */

static Status account_enter_pin(void)
{
      Pin your_pin;
      char ac_info[MAXLINE];

      printf("Please enter your PIN: ");
      /* PIN typed in wrong? */
      while(!scanf("%u", &your_pin)){
            fprintf(stderr, "Invalid PIN: please re-enter\n");
            fflush(stdin);
      }

      /* Read the file, a line at a time, to see if the database
         contains the PIN number entered */
      do{
      /* fgetpos [stdio.h] gets the current position of the file
         pointer in the file, and stores its value in
         'your_ac.file_pos'. When we find a customer's record, we
         need to remember where it was found in the file, so we can
         update it later, if necessary */
            fgetpos(your_ac.ac_file, &your_ac.file_pos);

            /* Read a record (single line) from the file, and store
               in the character array ac_info */
            if(fgets(ac_info, (int)MAXLINE, your_ac.ac_file) ==
            NULL){

                  /* PIN not found and end of file reached */
                  if(feof(your_ac.ac_file)){
                        fprintf(stderr, "Unrecognised PIN\n");
                        return (BADPIN);
                  }
                  /* End of file not reached, but file read failed -
                     the database is corrupt */
                  else return (FAULT);
            }

            /* Use sscanf to extract the PIN number from each record.
               If the assignment can't be made, the database is
               corrupt */
            if(!sscanf(ac_info, "%u", &your_ac.pin)) return(FAULT);

            /* Loop terminates when the PIN typed in matches a PIN in
               the database */
      } while(your_ac.pin != your_pin);

      /* The record corresponding to the PIN has been read in.  Now
         store the customer's PIN, account number and balance in the
         account data structure */
      if(sscanf(ac_info, "%u %ld %f", &your_pin, &your_ac.ac_num,
```

```
      &your_ac.balance) != 3)
            return(FAULT);
      return (OK);
}


/* account_update_database:  if a customer's account details have
changed, we need to update the account database */

static Status account_update_database(void)
{

/* The variable 'your_ac.file_pos' contains the position in the file
of the customer's record.  Its value has been set previously by
account_enter_pin. We use fsetpos [stdio.h] to move the file pointer
to the beginning of the customer's record in the file */
      fsetpos(your_ac.ac_file, &your_ac.file_pos);

/* Write the updated account details to the file. If we are
unsuccessful, the database is corrupt */
      if(fprintf(your_ac.ac_file, "%u %ld %9.2f\n", your_ac.pin,
      your_ac.ac_num, your_ac.balance) < 0)
            return (FAULT);

      return (OK);
}
```

ATM.c - the user front-end to the ATM program

```
/* File: ATM.c

   This file implements the user front-end to the ATM program.
   Note that the only way this file can modify a customer's account
   details is via the public interface functions specified in
   'account.h'
*/

#include <stdio.h>
#include <stdlib.h>

/* 'account.h' specifies the public interface to the account ADT */
#include "account.h"

int main(void)
{
      char service; /* What service does the customer require? */
      Status status; /* Has a fault occurred ? */

      status = account_open_database(); /* Access the account
                                           database */

      /* While the machine is working properly ... */
      while(status != FAULT){

      /* Get ready for the next customer.  Provide service only if a
         valid PIN is entered and no faults are encountered */
            if((status = account_next_customer()) != FAULT && status
            != BADPIN){

      /* Customers can specify services, so long as there are no
         faults, and they don't request their cards back */
                  do{
                        printf("\nSelect service:\n");
                        printf("(1) to make withdrawal\n");
                        printf("(2) to display balance\n");
```

```
                        printf("'e' to exit\n");

                        fflush(stdin);
                        scanf("%c", &service);

                        switch (service){
                              case '1':
                                  status = account_make_withdrawal();
                                  break;
                              case '2':
                                  account_display_balance();
                                  break;
                              default:
                                  break;
                        }
                  } while(service != 'e' && status != FAULT);
            }
/* Machine is working properly - return to top of outer loop, in
readiness for next customer. */
      }

/* If the program makes it this far, it is because a fault has arisen
*/
      fprintf(stderr, "Internal error: Machine out of service\n");

/* Close the account database file */
      account_close_database();

      return(0);
}
```

LISTING 13.1: THE SIEVE OF ERATOSTHENES

```
/* Program to find all prime numbers up to a specified limit,
   max, using the Sieve of Eratosthenes.

   An array with max+1 elements is set up.  We test the array
   indices to see if they are prime.  If they are not, then the
   corresponding array element is set to FALSE, otherwise it is
   set to TRUE.
*/

#include <stdio.h>
#include <stdlib.h>

typedef enum {FALSE, TRUE} Boolean;

int main(void)
{
      long int i, j, max;

      /* pointer to array containing elements of type Boolean */
      Boolean *isprime;

      printf("Enter upper limit: ");
      scanf("%ld", &max);

      if((isprime = (Boolean *)calloc(max + 1, sizeof(Boolean)))
      == NULL){
            fprintf(stderr, "Failed to allocate memory\n");
            exit(EXIT_FAILURE);
      }

/* Assume initially that all indices are prime */
      for(i = 2; i <= max; i++)
            isprime[i] = TRUE;

/* If an index can be generated by multiplying two numbers,
   then it can't be prime */
      for(i = 2; i <= max / 2; i++)
            for(j = 2; j <= max / i; j++)
                  isprime[i * j] = FALSE;

 /* If an array element has been set to TRUE, then the
    corresponding index is prime */
      for(i = 2; i <= max; i++)
            if(isprime[i]) printf("%d\n", i);

      return(0);
}
```

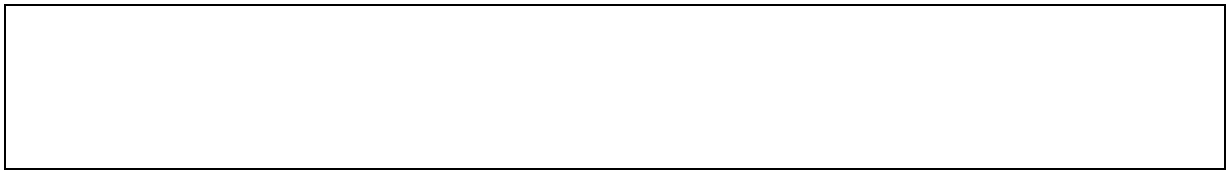
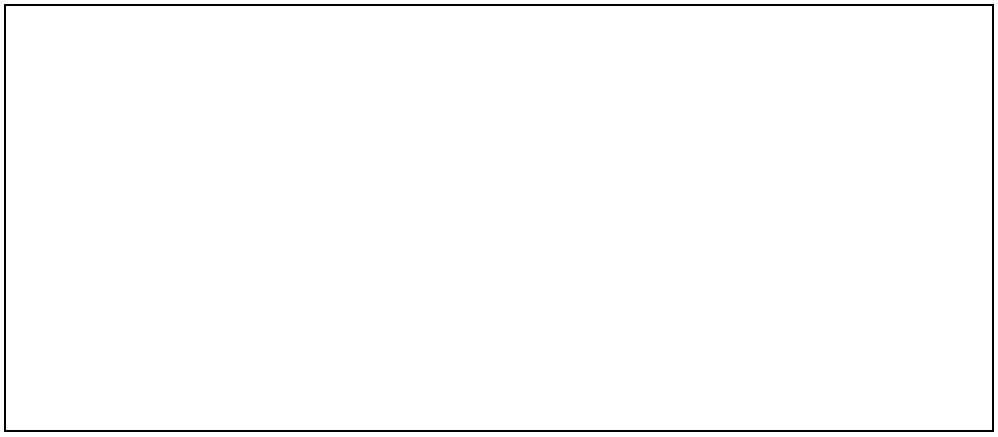
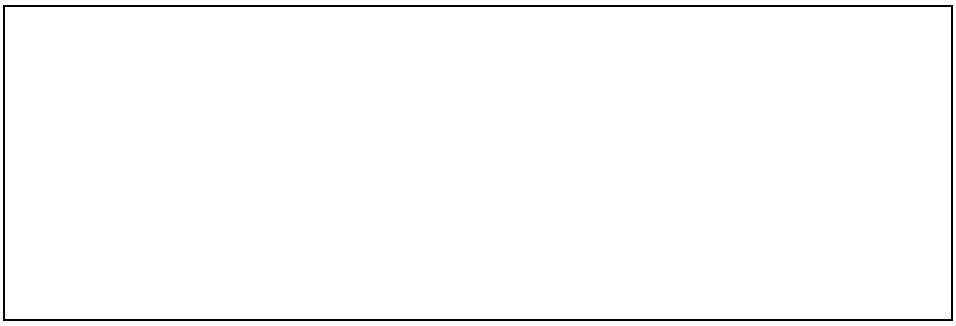


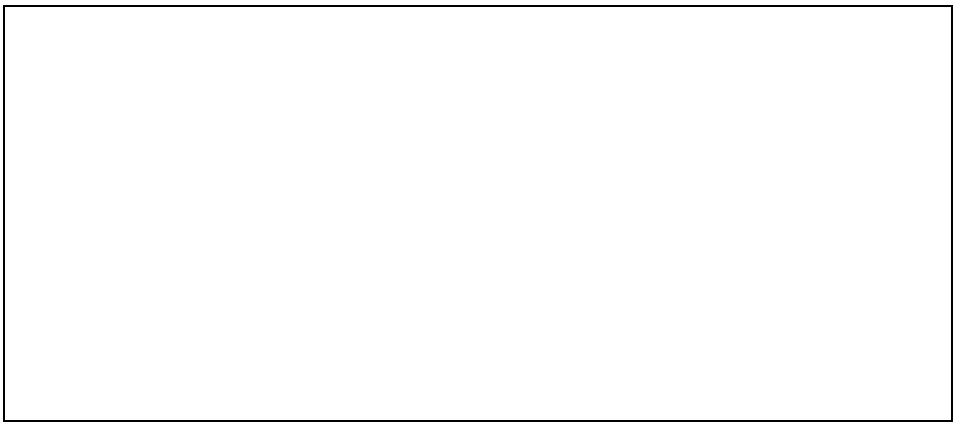
Figure 13.1: Linked List Representation.



(a) Insertion of a new node.



(b) Deletion of an existing node



(c) Moving an existing node

Figure 13.2: Operations on Linked Lists.

LISTING 13.2 LINKED LIST IMPLEMENTATION

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

typedef int Key;
typedef struct node Node;

struct node{
      Key key;
      Node *next;
};

/* For simplicity, declare head, and z as global variables */
Node *head, *z;

void list_initialise(void);
Node *insert_after(Node *, Key);
Node *delete_next(Node *);

void list_initialise()
{
      if((head = (Node *)malloc(sizeof(Node))) == NULL ||
             (z = (Node *)malloc(sizeof(Node))) == NULL){
            fprintf(stderr, "Failed to initialise list\n");
            exit(EXIT_FAILURE);
      }

      head->next = z; /* Link the head node to the z node */
      z->next = z; /*Link the z node to itself */
}

Node *insert_after(Node *current, Key data)
{
      Node *newnode;

/* Return the NULL pointer if space for new node can't be allocated
*/
      if((newnode = (Node *)malloc(sizeof(Node))) == NULL)
            return (NULL);

/* Link the new node to the node following the current node */
      newnode->next = current->next;

/* Link the current node to the new node */
      current->next = newnode;

/* Initialise the data field of the new node */
      newnode->key = data;

      return (current);
}

Node *delete_next(Node *current)
{
      Node *exnode;

/* Return the NULL pointer if we attempt to delete the z node */
      if((exnode = current->next) == z) return (NULL);

/* Link the current node to the node following the node to be deleted
*/
      current->next = exnode->next;

/* Free the memory allocated to the deleted node */
```

```
	free(exnode);

	return (current);
}
```

LISTING 13.3: THE JOSEPHUS PROBLEM

```
int main(void)
{
	int n, m, i;
	Node *current;

	list_initialise(); /* Set up the list */

	/* Get the values of n and m */
	printf("Enter the values of n and m (separated by a
	space):\n");
	scanf("%d %d", &n, &m);

	/* Set the current node to the head of the list */
	current = head;

	/* Create a list with n nodes.  Each person in the circle is
	identified by a number from 1 to n, which is stored in the data
	field of the corresponding node */
	for(i = 1; i <= n; i++){
		if(insert_after(current, i) == NULL){
			fprintf(stderr, "Error: Memory exhausted\n");
			exit(EXIT_FAILURE);
		}

		current = current->next;
	}

	/* List constructed, now link the last node to the first */
	current->next = head->next;

	/* When current = current->next, there is only one node left */
	while(current != current->next){
		/* Traverse the list to find the m'th node */
		for(i = 1; i < m; i++)
			current = current->next;

		/* Print the number of the person who has been expunged
		*/
		printf("%d\n", current->next->key);
		/* Delete the corresponding node */
		delete_next(current);
	}

	/* Print the last person standing */
	printf("%d\n", current->key);

	return(0);
}
```

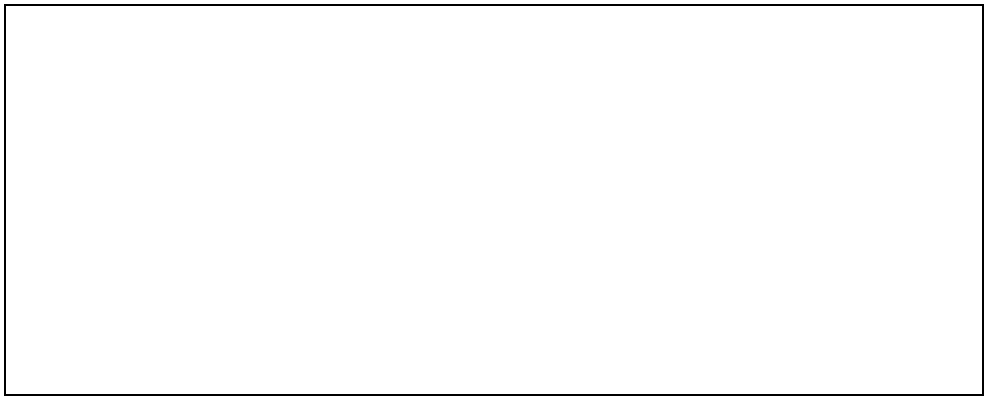


Figure 13.3: The Pushdown Stack.

LISTING 13.4: ARRAY-BASED STACK IMPLEMENTATION

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

#define STACK_MAX 1024 /* Maximum stack size */

typedef int Key; /* The data set consists of integers */

/* Set up the stack */
static Key stack[STACK_MAX];
/* 'top' is the array index of the top-most stack element */
static int top;

void stack_initialise(void)
{
      top = 0; /* Initially, the stack is empty */
}

void push(Key value)
{
/* If the stack is full, there's no room to push an extra data
   item */
      if(top == STACK_MAX){
            fprintf(stderr, "Error:  Stack overflow\n");
            exit(EXIT_FAILURE);
      }

/* Place the data on the top of the stack.  Increment top in
   readiness for the next data item */
      stack[top++] = value;
}

Key pop(void)
{
/* If the stack is empty, there's no data left to pop */
      if(top == 0){
            fprintf(stderr, "Error:  Stack underflow\n");
            exit(EXIT_FAILURE);
      }

/* The push operation sets the index top to the next available
   array element so, in order to pop a data item, we first have to
   decrement top */
      return(stack[--top]);
}
```

```
int isempty(void)
{
      /* Test for an empty stack */
      return (top == 0);
}
```

LISTING 13.5: LIST-BASED QUEUE IMPLEMENTATION

```
typedef int Key;
typedef struct node Node;

struct node{
      Key key;
      Node *next;
};

/* The tail of the queue, where new data items are added,
   corresponds to the head of the linked list. */
static Node *tail, *z;

void queue_initialise()
{
      if((tail = (Node *)malloc(sizeof(Node))) == NULL ||
             (z = (Node *)malloc(sizeof(Node))) == NULL){
            fprintf(stderr, "Failed to initialise queue\n");
            exit(EXIT_FAILURE);
      }

      tail->next = z; /* Link tail of the queue to the z node */
      z->next = z; /*Link the z node to itself */
}

void queue_add(Key data)
{
      Node *newnode;

      if((newnode = (Node *)malloc(sizeof(Node))) == NULL){
            fprintf(stderr, "Error:  Queue overflow\n");
            exit(EXIT_FAILURE);
      }

/* Add the new data item to the tail of the queue */
      newnode->next = tail->next;
      tail->next = newnode;
      newnode->key = data;
}

Key queue_get(void)
{
      Node *exnode, *prevnode;
      Key data;

/* If the z node follows the tail node, then the queue is empty */
      if(queue_isempty()){
            fprintf(stderr, "Error:  Queue underflow\n");
            exit(EXIT_FAILURE);
      }

/* In order to process the item at the front of the queue, we need
   to work our way from the tail of the queue to the front of the
   queue, which is represented by the z node. */
      exnode = tail;

      do{
```

```
            /* Keep track of the node prior to the node to be removed
            from the queue */
            prevnode = exnode;

            /* "Hop" to the next node in the queue... */
            exnode = exnode->next;

            /* ...until we reach the node before the z node
            (i.e. the front of the queue) */

      }     while(exnode->next != z);

      /* Get the data item at the front of the queue */
      data = exnode->key;

      /* Remove the node from the front of the queue */
      free(exnode);

      /* Make the previous node the new front of the queue */
      prevnode->next = z;
      return(data);
}

int queue_isempty(void)
{
/* Queue is empty if tail is linked to z node */
      return(tail->next == z);
}
```

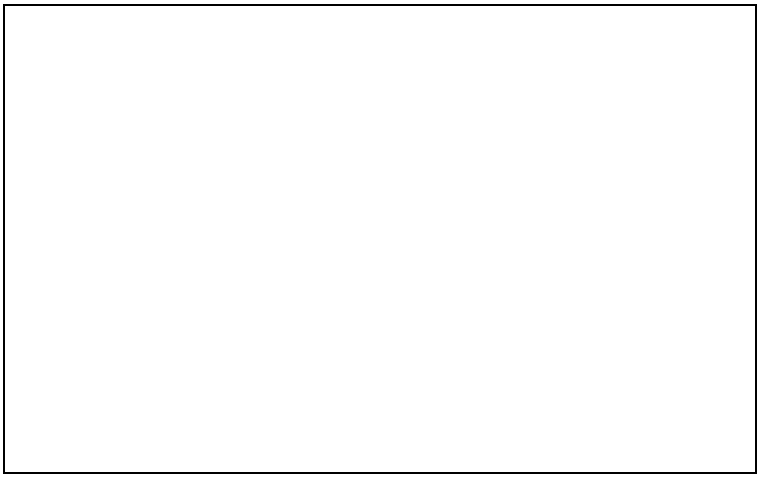


Figure 14.1: A Sample Tree.

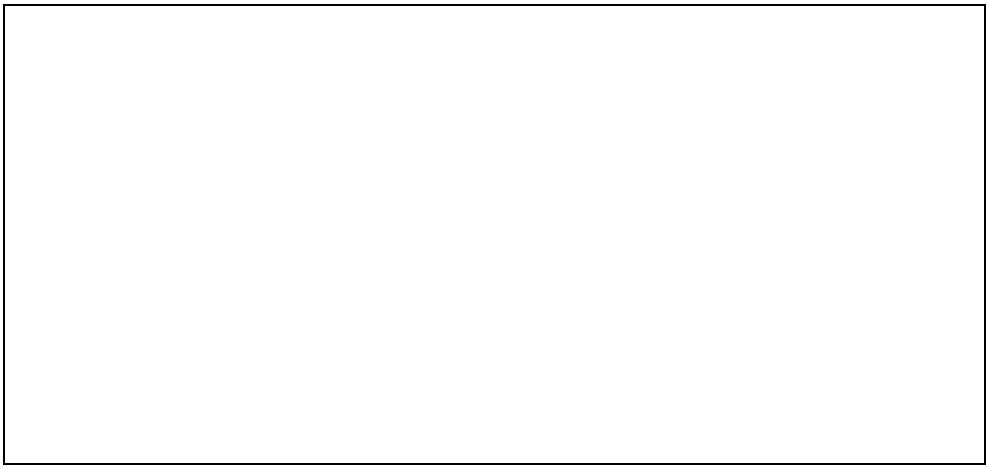


Figure 14.2: A Sample Binary Tree.

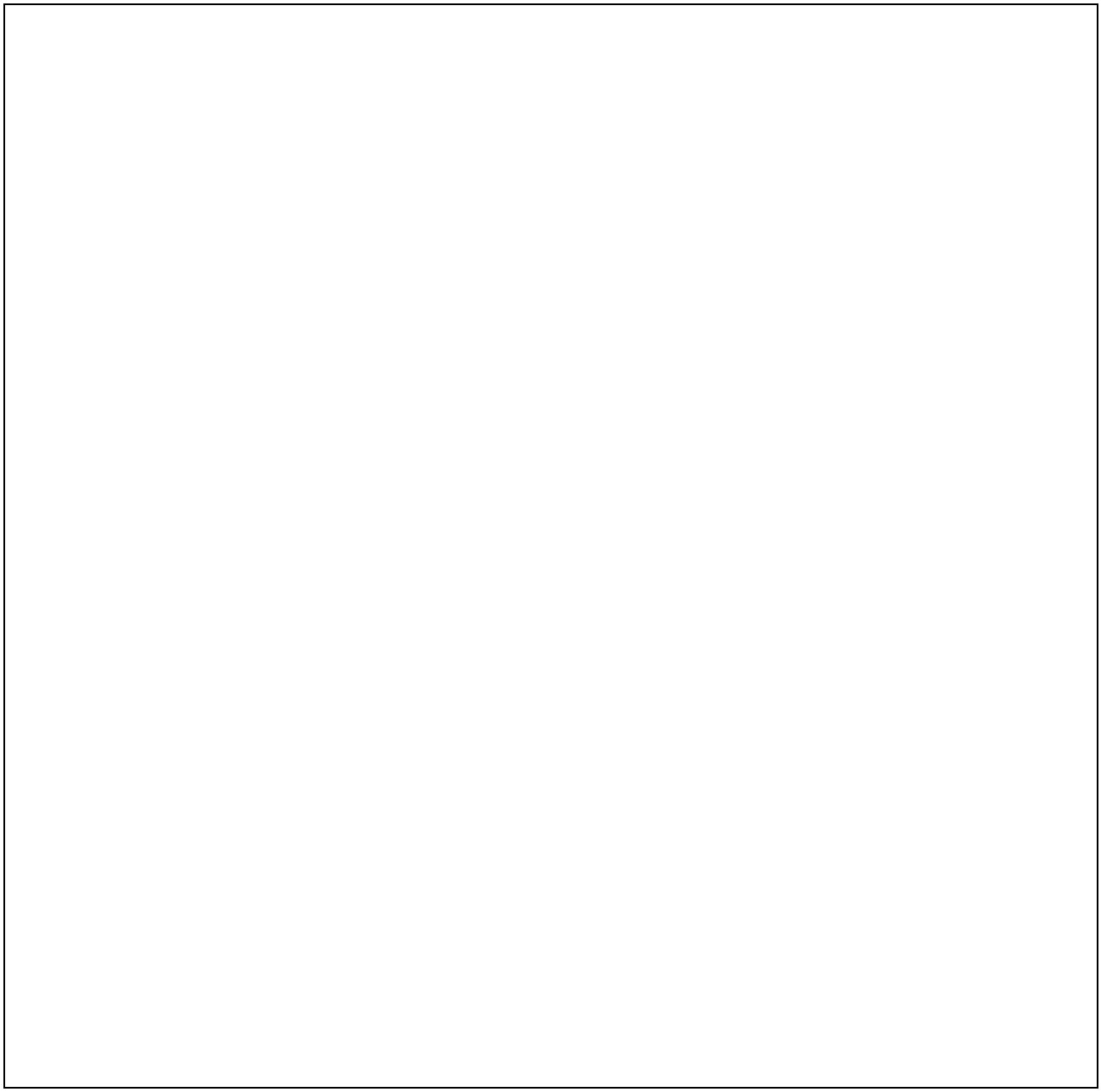


Figure 14.3: Construction of an Ordered Binary Tree.

LISTING 14.1: BINARY TREE CONSTRUCTION

```
enum {FALSE, TRUE} Boolean;
typedef int Key;

/* In this example, the information stored in each node
   will be an integer */
typedef int Info;

typedef struct node Node;

struct node{
      Key key;
      Info info;
      Node *left;
      Node *right;
      };

/* Declare pointers to the root and terminal nodes */
static Node *root, *z;

/* Initialise the tree root and terminal nodes */
void tree_initialise(void)
{
   if((root = (Node *)malloc(sizeof(Node))) == NULL ||
   (z = (Node *)malloc(sizeof(Node))) == NULL){
      fprintf(stderr, "Failed to initialise tree.\n");
      exit(EXIT_FAILURE);
}

root->key = 0;
root->left = z;
root->right = z;
z->left = z;
z->right = z;
z->info = -1;
}

/* Ordered insertion of new node */
int Insert(Key key, Info info)
{
   Node *parent, *current;

   current = root;

   /* Until we reach a terminal node... */
   while(current != z){
      /* We need to keep track of the parent of the current node,
      in order to add the new child node */
      parent = current;

      /* Implementation of the ordering relation */
      if(key < current->key)
           current = current->left; /* Descend to left child node */
      else
           current = current->right;/* Descend to right child node */
  }

/* Generate new node */
if((current = (Node *)malloc(sizeof(Node))) == NULL)
      return (FALSE);

/* Make the new node a child of the parent node. */
```

```
if(key < parent->key)
      parent->left = current;
else
      parent->right = current;

/* Initialise new node */
current->key = key;
current->info = info;
current->left = z;
current->right = z;

return (TRUE); /* New node successfully inserted */
}
```

LISTING 14.2: BINARY TREE SEARCH

```
Info search(Key key)
{
      Node *current;

      current = root;

      /* In case the key we are searching for isn't present in the
      tree, we assign it to the z node.  The search will then
      terminate when the z node is reached, and the dummy info
      structure contained in the z node will be returned */
      z->key = key;

      /* Search the binary tree according to the ordering relation
      with which it was constructed */
      while(key != current->key){
            if(key < current->key)
                  current = current->left;
            else
                  current = current->right;
      }

      return(current->info);
}
```

LISTING 15.1: RECURSIVE CALCULATION OF GCD

```
int gcd(int u, int v)
{
      if(u == 0) return (v);

      if(u > v)
            return(gcd(u % v, v));
      else
            return(gcd(v, u));
}
```

Call	u	v
1	461,952	116,298
2	113,058	116,298
3	116,298	113,058
4	3,240	113,058
5	113,058	3,240
6	2,898	3,240
7	3,240	2,898
8	342	2,898
9	2,898	342
10	162	342
11	342	162
12	18	162
13	162	18
14	0	18

Table 15.1: Recursive Calculation of GCD

LISTING 15.2: NON-RECURSIVE CALCULATION OF GCD

```
int gcd(int u, int v)
{
      int temp;

      while(u > 0)
      {
            if(u < v)
            {
                  temp = u; u = v; v = temp;
            }
            u = u % v;
      }

      return (v);
}
```

LISTING 15.3: RECURSIVE COMPUTATION OF NTH FIBONACCI NUMBER

```
long int Fibonacci(int n)
{

      if(n <= 1)
            {
            return 1;
            }

      return (Fibonacci(n-1) + Fibonacci(n-2));
}
```

LISTING 15.4: COMPUTATION OF NTH FIBONACCI NUMBER USING AN ARRAY

```
long int Fibonacci(int n)
{
      int i;
      long int *F;

      if((F = (long int *)malloc(n * sizeof(long int))) == NULL)
            {
            printf("Fibonacci:  failed to allocate memory.\n");
            exit(EXIT_FAILURE);
            }

      F[0] = F[1] = 1;

      for(i = 2; i < n; i++)
            F[i] = F[i-1] + F[i-2];

      return (F[n-1] + F[n-2]);
}
```

LISTING 15.5: RECURSIVE QUICKSORT ALGORITHM

```
void quicksort(Key keys[], int left, int right)
{
      int i;

      if(right > left)
      {
            i = partition(keys, left, right);
            quicksort(keys, left, i - 1);
            quicksort(keys, i + 1, right);
      }

      return;
}
```

LISTING 15.6: IMPLEMENTATION OF FUNCTION "PARTITION" FOR QUICKSORT

```
int partition(Key keys[], int left, int right)
{
      int i, j;
      Key sort_key, temp;

      sort_key = keys[right];
      i = left - 1;
      j = right;

      for(;;)
      {
            while(keys[++i] < sort_key);
            while(keys[--j] > sort_key && j > left);

            if(i >= j) break;

            temp = keys[i];
            keys[i] = keys[j];
            keys[j] = temp;
      }

      keys[right] = keys[i];
      keys[i] = sort_key;

      return (i);
}
```

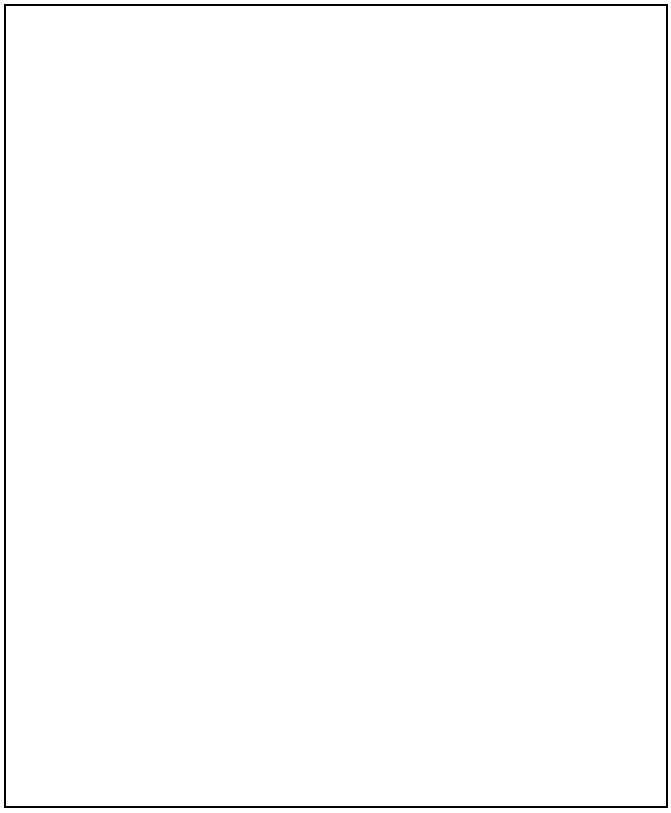


Figure 15.1: An Illustration of the Partitioning Function.

LISTING 15.7: QUICKSORT, WITH END-RECURSION REMOVAL

```
void quicksort(Key keys[], int left, int right)
{
      int i;

      start:
      if(right > left)
      {
            i = partition(keys, left, right);
            quicksort(keys, left, i - 1);
            left = i + 1;     /*Manually modify arguments*/
            goto start;
      }

      return;
}
```

LISTING 15.8: STRUCTURED QUICKSORT, WITH END-RECURSION REMOVAL

```
void quicksort(Key keys[], int left, int right)
{
      int i;

      while(right > left) /* while replaces if-goto*/
      {
            i = partition(keys, left, right);
            quicksort(keys, left, i - 1);
            left = i + 1;     /*Manually modify arguments*/
      }

      return;
}
```

LISTING 15.9: QUICKSORT WITH PARTITION SIZE COMPARISON

```
void quicksort(Key keys[], int left, int right)
{
      int i;

      while(right > left) /* while replaces if-goto*/
      {
            i = partition(keys, left, right);

            if(i - left < right - i) /*Compare partition sizes*/
            {
                  quicksort(keys, left, i - 1);
                  left = i + 1;     /*Manually modify arguments*/
            }
            else
            {
            quicksort(keys, i + 1, right);
            right = i - 1;
            }
      }
      return;
}
```

LISTING 15.10: QUICKSORT IMPLEMENTED USING PUSHDOWN STACK

```
void quicksort(Key keys[], int n)
{
      int i, left, right;

      stack_initialise();
      push(0);
      push(n-1);

      while(!isempty())
      {
            right = pop();
            left = pop();

            while(right > left)
            {
                  i = partition(keys, left, right);

                  if(i - left > right - i)
                  {
                        push(left);
                        push(i - 1);
                        left = i + 1;
                  }
                  else
                  {
                        push(i + 1);
                        push(right);
                        right = i -  1;
                  }
            }

      }
      return;
}
```

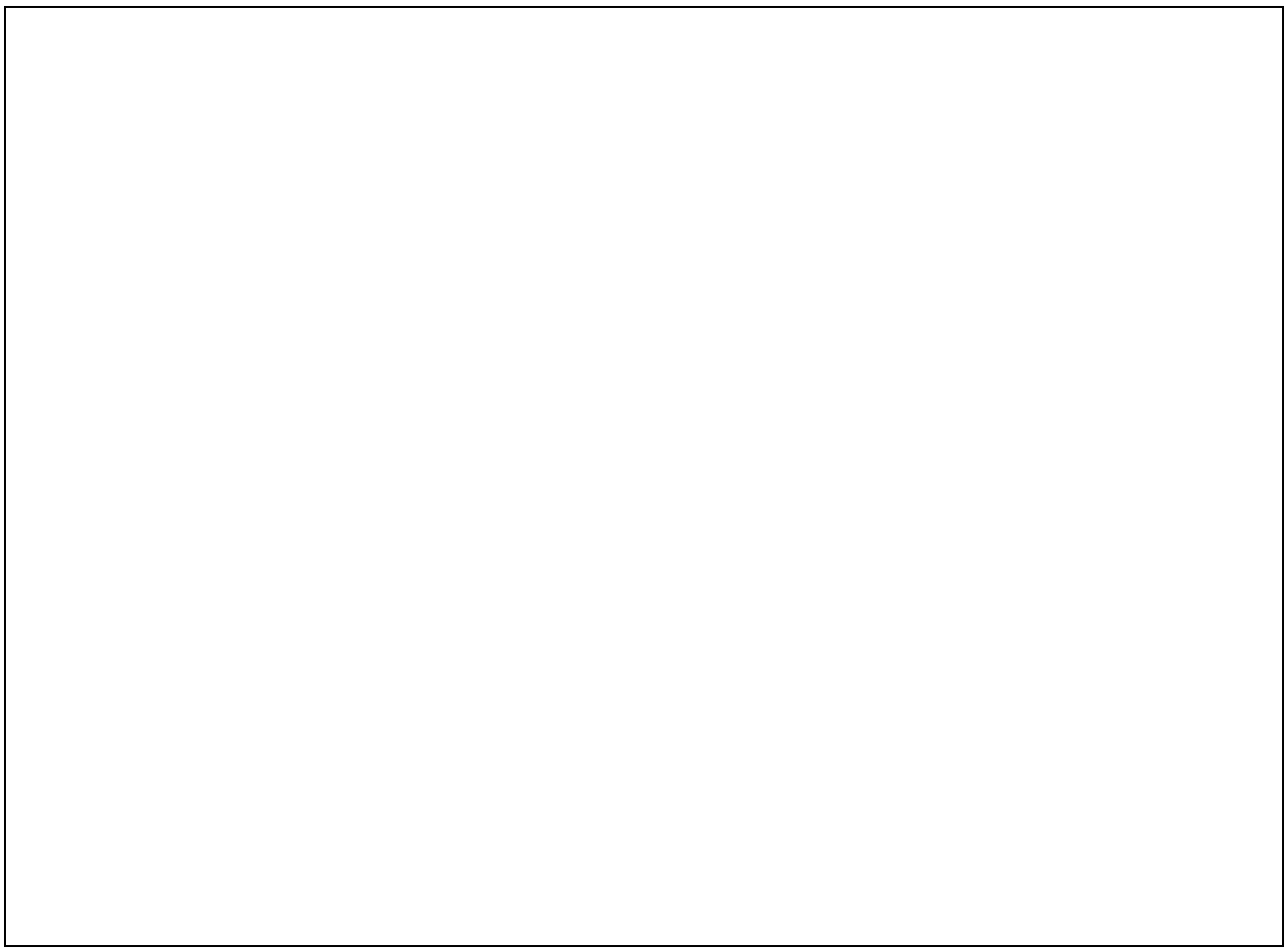


Figure 15.2: In-order Binary Tree Traversal.

LISTING 15.11: RECURSIVE, IN-ORDER BINARY TREE TRAVERSAL

```
void traverse(Node *node)
{
      if(node != z){
            traverse(node->left);
            print(node);
            traverse(node->right);
      }
}
```

LISTING 15.12: OPERATION-INDEPENDENT IN-ORDER TRAVERSAL

```
void traverse(Node *node, void (*visit)(Node *))
{
      if(node != z){
            traverse(node->left);
            visit(node);
            traverse(node->right);
      }
}
```

LISTING 15.13: PRE-ORDER TRAVERSAL

```
void traverse(Node *node, void (*visit)(Node *))
{
      if(node != z){
            visit(node);
            traverse(node->left);
            traverse(node->right);
      }
}
```

LISTING 15.14: POST-ORDER TRAVERSAL

```
void traverse(Node *node, void (*visit)(Node *))
{
      if(node != z){
            traverse(node->left);
            traverse(node->right);
            visit(node);
      }
}
```

LISTING 15.15: NON-RECURSIVE PRE-ORDER TRAVERSAL

```
void Preorder(Node *node, void (*visit)(Node *))
{
      push(node);

      while(!isempty()){
            node = pop();

            while(node != z){
                  visit(node);
                  push(node->right);
                  node = node->left;
            }
      }
}
```

LISTING 15.16: NON-RECURSIVE IN-ORDER TRAVERSAL

```
void Inorder(Node *node, void (*visit(Node *))
{
      push(node);

      while(1){
            node = pop();

            while(node != z){
                  push(node);
                  node = node->left;
            }

            if(stack_empty()) break;
            node = pop();
            node_print(node);
            push(node->right);
      }
}
```

LISTING 16.1: FUNCTION TO MERGE TWO ARRAYS

```
void merge(Key keya[], int m, Key keyb[], int n, Key keyc[])
{
      int i, j, k;

      i = j = 0;

      for(k = 0; k < m + n; k++){
            if(j == n || (i != m && keya[i] <= keyb[j]))
                  keyc[k] = keya[i++];
            else
                  keyc[k] = keyb[j++];
      }
}
```

LISTING 16.2: FUNCTION TO MERGE TWO LINKED LISTS

```
Node *merge(Node *ptra, Node *ptrb)
{
  Node *ptrc;

  ptrc = z;

  do {
    if(ptrb == z || (ptra != z && ptra->key <= ptrb->key))
    {
      ptrc->next = ptra;
      ptrc = ptra;
      ptra = ptra->next;
    }
    else
    {
      ptrc->next = ptrb;
      ptrc = ptrb;
      ptrb = ptrb->next;
    }
  } while(ptrc != z);

  ptrc = z->next;
  z->next = z;

  return ptrc;
}
```

LISTING 16.3: RECURSIVE MERGESORT, USING ARRAYS

```
void mergesort(Key keys[], int n, Key temp[])
{
      int i, mid;

      if(n > 1){
            mid = n / 2;
            mergesort(keys, mid, temp);
            mergesort(keys + mid, n - mid, temp);
            merge(keys, mid, keys + mid, n - mid, temp);

            for(i = 0; i < n; i++)
                  keys[i] = temp[i];
      }
}
```

LISTING 16.4: RECURSIVE MERGESORT, USING LINKED LISTS

```
Node *mergesort(Node *ptr, unsigned int n)
{
  unsigned int i, mid;
  Node *current, *ptrb;

  if(n > 1){
    mid = n / 2;
    current = ptr;

    for(i = 1; i < mid; i++)
      current = current->next;

    ptrb = current->next;
    current->next = z;

    ptr = mergesort(ptr, mid);
    ptrb = mergesort(ptrb, n - mid);
    ptr = merge(ptr, ptrb);
    }

  return (ptr);
}
```

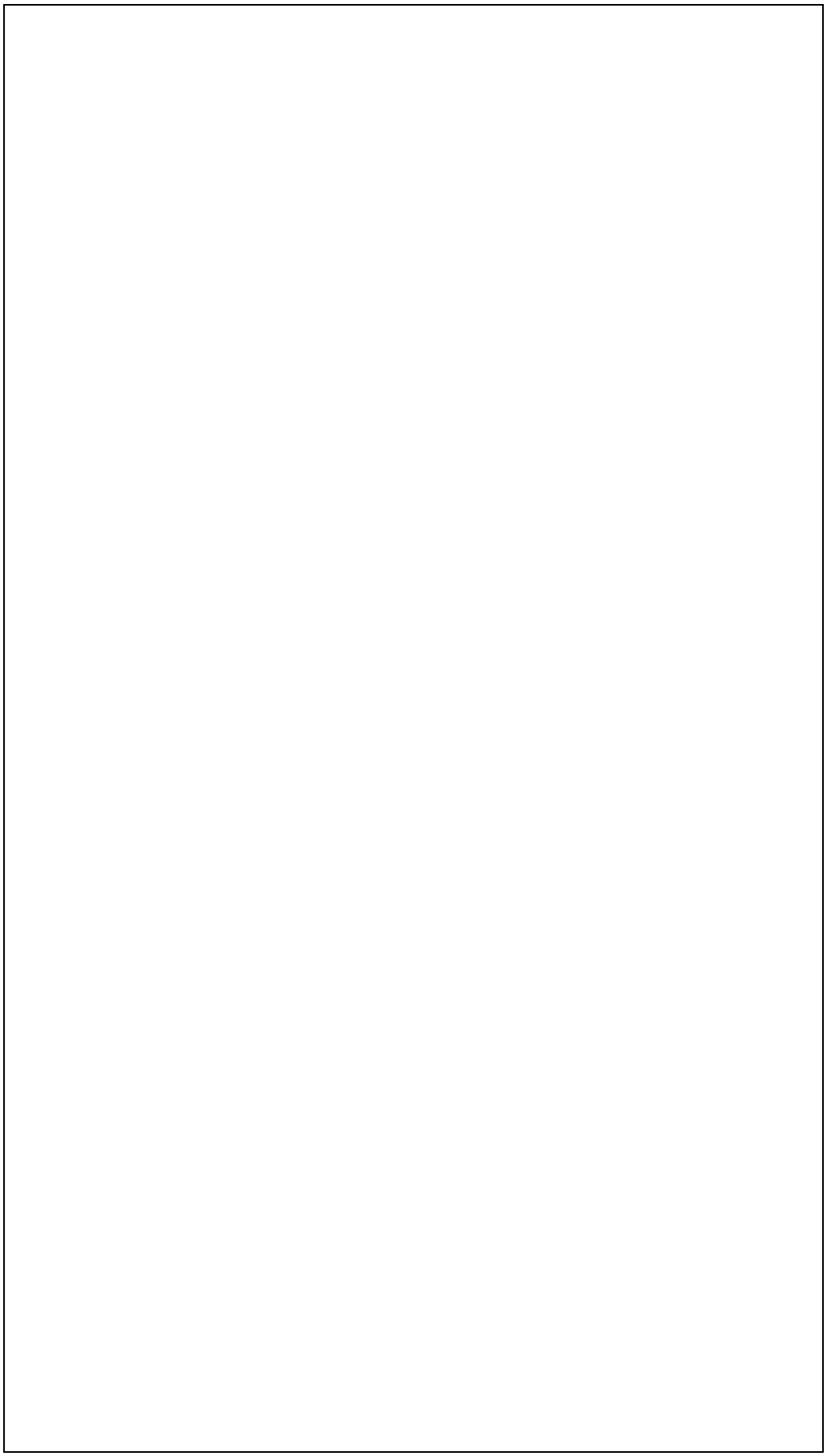


Figure 16.1: Illustration of Recursive Mergesort.

LISTING 17.1: SIMPLE ALGORITHM TO SUM MATRIX ELEMENTS

```
#define N 100

int main(void)
{
      int matrix[N][N];
      int grand_total=0,i,j;

      for (i=0;i<N-1;i++)
      {
            rows[i]=0;
            for (j=0;j<n-1;j++)
            {
                  rows[i]=rows[i]+matrix[i,j];
                  grand_total=grand_total+matrix[i,j];
            }
      }
      return(0);
}
```

LISTING 17.2: ALTERNATIVE ALGORITHM TO SUM MATRIX ELEMENTS

```
#define N 100
int main(void)
{
      int matrix[N][N];
      int grand_total=0,I,j;

      for (i=0;i<N-1;i++)
      {
            rows[i]=0;
            for (j=0;j<N-1;j++)
            {
                  rows[i]=rows[i]+matrix[i,j];
            }
            grand_total=grand_total+rows[i];
      }
      return(0);
}
```

LISTING 17.3: BUBBLESORT ALGORITHM WITH DATA MOVES

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>


#define ARRAYSIZE 8

typedef enum {FALSE,TRUE} Boolean; /* Used for Boolean (True/False)
variables */
typedef int Key;   /* The elements of the  array to be sorted are
integers */
void bubble_sort(Key * array, int n);

int main(void)
{
      int i;
```

```
      Key array[ARRAYSIZE];
      printf("UNSORTED LIST\n\n");
      for (i=0;i<ARRAYSIZE;i++)
      {
            array[i]=random(ARRAYSIZE);
            printf("Element %d is:  %d\n",i,array[i]);
      }
      bubble_sort(array,ARRAYSIZE);
      printf("\n\nSORTED LIST\n\n");
      for (i=0;i<ARRAYSIZE;i++)
      {
            Printf("Element %d is:  %d\n",i,array[i]);
      }

      return(0);
}

/** BUBBLESORT ALGORITHM **/
void bubble_sort(Key *array, int n)
{
      int j, k;
      Boolean exchange_made;
      Key temp;
      k = 0;
      exchange_made = TRUE;

      /* Make up to n - 1 passes through array, exit early if no
         exchanges are made on previous pass  */

      while ((k < n - 1) && exchange_made)
      {
            exchange_made = FALSE;
            ++k;
            for (j = 0; j < n - k; ++j)   /* Number of comparisons on
                                             kth pass  */
                  if (array[j] > array[j + 1])
                  {
                        temp = array[j];  /* Exchange must be made*/
                        array[j] = array[j + 1];
                        array[j + 1] = temp;
                        exchange_made = TRUE;
                        }
      }
}
```

LISTING 17.4: BUBBLESORT ALGORITHM WITH NO DATA MOVES

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define ARRAYSIZE                                                    8

typedef enum {FALSE,TRUE} Boolean; /* Used for Boolean (True/False)
variables */
typedef int Key; /* The elements of the  array to be sorted are
integers */
typedef int pointer_array[ARRAYSIZE]; /* We now have an auxiliary
matrix of "pointers" */
void pointer_bubble_sort(Key * array, pointer_array pointer, int n);

int main(void)
{
      int i;
```

```
      Key array[ARRAYSIZE];
      pointer_array pointer;
      printf("UNSORTED LIST\n\n");
      for (i=0;i<ARRAYSIZE;i++)
      {
            array[i]=random(ARRAYSIZE);
            printf("Element %d is:  %d\n",i,array[i]);
      }
      pointer_bubble_sort(array,pointer,ARRAYSIZE);

      return(0);
}


void pointer_bubble_sort(Key *array, pointer_array pointer,int n)
{
      int j, k;
      Boolean exchange_made;
      Key temp;
      for (k=0;k<n;k++)
      {
            pointer[k]=k;
      }

      k = 0;
      exchange_made = TRUE;

      /* Make up to n-1 passes through array, exit early if no
         exchanges are made on previous pass  */

      while ((k < n - 1) && exchange_made)
      {
            exchange_made = FALSE;
            ++k;
            for (j = 0; j < n - k; ++j)   /* Number of comparisons on
            kth pass  */
                  if (array[pointer[j]] > array[pointer[j + 1]])
                  {
                        temp = pointer[j];/*Exchange must be made*/
                        pointer[j] = pointer[j + 1];
                        pointer[j + 1] = temp;
                        exchange_made = TRUE;
                        }
      }

      /* Simple printout of results for comparison */
      printf("\n\nSORTED LIST\n\n");
      for (k=0;k<n;k++)
      {
            printf("Element %d is:  %d\n",k,array[pointer[k]]);
      }
}
```

LISTING 18.1: EXAMPLE OF PROGRAM USING CLASS STAFFMEMBER

```
#include <iostream.h>  // Needed to provide basic screen input/output


// Declaration of a class called StaffMember
class StaffMember
{
      private:          //private data and methods
            int age;
            long int staff_number;
```

```
            float salary;
      public:                 // public data and methods
            StaffMember(int,long int, float);
            void showStaffMember(void);
            void change_salary(float);

};

// Implement methods of StaffMember
StaffMember::StaffMember(int a, long int n, float s)
// Constructor method
{
      age=a;
      staff_number=n;
      salary=s;
}

void StaffMember::showStaffMember(void)
// Display values of StaffMember data
{
      cout << "Age is " << age << "\tStaff Number is " <<
staff_number << "\tSalary is £" << salary << "\n";
}

void StaffMember::change_salary(float x)
// Change the StaffMember's salary
{
      salary=x;
}


// The main routine - this is the glue that holds everything together
int main(void)
{
      StaffMember secretary(34,9008010025L,15450.0);
      // Define a StaffMember
      secretary.showStaffMember();  // Show its values
      secretary.change_salary(16250.0); // Change salary
      secretary.showStaffMember(); // Show its new values

      return(0);
}
```

LISTING 18.2: EXAMPLE OF CLASS WITH MULTIPLE CONSTRUCTORS

```
#include <iostream.h>
// Illustration of different types of constructor
class StaffMember
{
      private:
            int age;
            long int staff_number;
            float salary;
      public:
            StaffMember(void);       // Default constructor when no
args given
            StaffMember(int,long int, float);
            StaffMember(int , long int );
            void showStaffMember(void);
            void change_salary(float);

};
```

```
// Implement member functions of StaffMember
StaffMember::StaffMember()     // Default constructor when no args
given
{
      age=0;
      staff_number=99999999;
      salary=1.0;
}
//
StaffMember::StaffMember(int a, long int n, float s)
{ // Constructor method number 2
      age=a;
      staff_number=n;
      salary=s;
}
//
StaffMember::StaffMember(int a, long int s)
// Constructor method number 3
{
      age=a;
      staff_number=s;
      salary=1.0;
}

void StaffMember::showStaffMember(void)
{
      cout << "Age is " << age << "\tStaff Number is " <<
staff_number << "\tSalary is £" << salary << "\n";
}

void StaffMember::change_salary(float x)
{
      salary=x;
}

int main(void)
{
      StaffMember secretary(34,900084213L,15450.0); // Constructor 2
      StaffMember vicepresident; //Constuctor 1
      StaffMember president(56,89000121L); // Constructor 3
      secretary.showStaffMember();
      vicepresident.showStaffMember();
      president.showStaffMember();

      return(0);
}
```

LISTING 18.3: EXAMPLE OF PROGRAM USING INHERITANCE FROM CLASS STAFFMEMBER

```
#include <iostream.h>
// Illustration of different types of constructor
class StaffMember
{
      protected:
            int age;
            long int staff_number;
            float salary;
      public:
            StaffMember(void);       // Default constructor when no
args given
            StaffMember(int,long int, float);
            StaffMember(int , long int );
            void showStaffMember(void);
```

```
            void change_salary(float);

};

// Implement member functions of StaffMember
StaffMember::StaffMember()     // Default constructor when no args
given
{
      age=0;
      staff_number=99999999L;
      salary=1.0;
}
//
StaffMember::StaffMember(int a, long int n, float s)
{
      age=a;
      staff_number=n;
      salary=s;
}
//
StaffMember::StaffMember(int a, long int s)
// Constructor method
{
      age=a;
      staff_number=s;
      salary=1.0;
}

void StaffMember::showStaffMember(void)
{
      cout << "Age is " << age << "\tStaff Number is " <<
staff_number << "\tSalary is £" << salary << "\n";
}

void StaffMember::change_salary(float x)
{
      salary=x;
}

class Supervisor: public StaffMember   // Supervisor is derived from
StaffMember
{
            private:
                  int number_of_employees;
            public:
                  Supervisor(int a,long int s,float salary, int n);
                  void showEmployeeNumber();

};


Supervisor::Supervisor(int a, long int s, float salary, int
n):StaffMember(a,s,salary)
// Constructor method
{
      number_of_employees=n;
}

void Supervisor::showEmployeeNumber(void)
{
      cout    <<    "Number    of    employees    supervised    "    <<
number_of_employees;
}

// Create some instance and call some methods
int main(void)
```

```
{
      StaffMember secretary(34,900084213L,15450.0);
      Supervisor linemanager(45,988888888L,16125.0,10);
      secretary.showStaffMember();
      linemanager.showStaffMember();
      linemanager.showEmployeeNumber(); // Should display number 10

      return(0);
}
```

LISTING 18.4: EXAMPLE OF PROGRAM USING POLYMORPHISM

```
#include <iostream.h>
//  *********** CLASS STAFFMEMBER ******************
class StaffMember
{
      protected:
            int age;
            long int staff_number;
            float salary;
      public:
            StaffMember(int,long int, float);
            virtual void showStaffMember(void);
            void change_salary(float);

};

// Implement member functions of StaffMember
StaffMember::StaffMember(int a, long int n, float s)
{
      age=a;
      staff_number=n;
      salary=s;
}

void StaffMember::showStaffMember(void)
{
      cout << "Age is " << age << "\tStaff Number is " <<
staff_number
      << "\tSalary is £" << salary << "\n";
}

void StaffMember::change_salary(float x)
{
      salary=x;
}

//  *********** CLASS SUPERVISOR ******************
class Supervisor: public StaffMember   // Supervisor is derived from
StaffMember
{
            private:
                  int number_of_employees;
            public:
                  Supervisor(int a,long int s,float salary, int n);
                  virtual void showStaffMember(void);
};


Supervisor::Supervisor(int a, long int s,float salary, int
n):StaffMember(a,s,salary)
{
      number_of_employees=n;
}
```

```
void Supervisor::showStaffMember(void)
{
      cout << "Age is " << age << "\tStaff Number is " <<
staff_number <<
      "\tSalary is £" << salary << "\t Number of employees supervised
"
      << number_of_employees <<"\n";
}

// ********** The main function ***************

// Create some instance and call some methods
int main(void)
{
      StaffMember secretary(34,900084213L,15450.0);
      Supervisor linemanager(45,988888888L,16125.0,10);
      secretary.showStaffMember();
      linemanager.showStaffMember();
      return(0);
}
```