20

20 Dynamic data and pointers

In all the examples considered so far, we have known how many data elements there will be, or we have at least decided on some maximum number that we are prepared to deal with. The data may be global, their space can be sorted out at compile-time (or link-load time); or the data may be temporary such as variables that belong to a function and which exist on the stack only while the code of that function is being executed. But a lot of problems involve data that aren't like that. You know that you have to deal with data objects, but you really don't know how many there will be, and nor do you know for how long you will need them.

Objects with variable lifetimes

The main example in this chapter is an "Air Traffic Control" game. The game involves the player controlling the movements of aircraft, giving them orders so that they line up with a runway, moving at an appropriate speed and rate of descent so that they can land safely. The game simulates the passage of time and introduces new aircraft into the controlled air space at a predefined frequency. The "lifetime" of an individual aircraft varies depending on the skill of the player. Aircraft get created by the game component; they get removed on successful landing (or by running out of fuel). If the player is skilful, the correct sequence orders is given to adjust the aircraft's speed, height and bearing through a number of stages until it is aligned with the runway. If the player makes a mistake, an aircraft may have to overshoot and circle around for a second try (or may run out of fuel). So the lifetime of aircraft does vary in essentially arbitrary ways. The number of aircraft in existence at a particular moment will also vary. At simple game levels there will be one or two (plus any making second attempts after aborted landings); at higher game levels there could be dozens.

The game program would represent the aircraft using data structures (either structs or instances of some class Aircraft). But these data structures have to be handled in ways that are quite different from any data considered previously.

The "heap"

Section 20.1 introduces the "heap" and the operators that can be used to create and destroy data objects in the heap. The "heap" is essentially an area of memory set aside for a program to use to create and destroy data structures that have varied lifetimes like the example aircraft.

Addresses and "Pointers"

The actual allocation of space in the heap for a new data object is handled by a run-time support routine. This "memory manager" routine will select some space in the heap when asked to create an object of a given size. The memory manager reports where the object has been placed by returning its address. This address becomes the value held in a "pointer variable". The heap-based object has to be accessed "indirectly via the pointer". Section 20.2 looks at issues like the definition of pointer variables, the use of pointers to access data members of an object, and operations on entire structures that are accessed through pointers.

"Air Traffic Controller"

A simple Air Traffic Control game is used in Section 20.3 as a practical illustration. This version doesn't have a particularly attractive user interface; it is just a framework with which to illustrate creation, access, and destruction of objects.

The "address of" operator

You can work with pointers to data elements other than those allocated on the heap. We have already had cases where the '`&`' address of operator was used to get a pointer value (when passing an address to the `read()` and `write()` low level i/o functions). Section 20.4 looks at the use of the `&` operator and a number of related issues. Many of the libraries that you use are still C language libraries and so you must learn something of C's styles of pointer usage.

Pointers and arrays

Another legacy of C is the rather poorly defined concept of an array. C programming idioms often abandon the concept of an array as a composite structure of many elements identifiable and accessible using their index value. Instead the C hacker style uses the assembly language level notions of a block of memory that can be accessed via address registers that have been loaded with the addresses of byte locations in memory. In this style, the contents of arrays are accessed using pointers and arithmetic operations are performed on pointers to change their values so as to identify different array elements. Because of the large amount of existing C code that you will need to work with, you need a "reading knowledge" of some of these pointer idioms. This is the content of Section 20.5. Note, it should be a "reading knowledge"; you should not write code employing these coding idioms.

Section 20.6 discusses "networks", showing how complicated structures can be built out of separate parts linked together with pointers.

20.1 THE "HEAP"

As illustrated in Figure 20.1, the memory allocated to a program is divided into four parts or "segments". The segments are: "code", "static data", "stack", and "heap". (This organization is conceptual only. The actual realization on a given computer architecture may well be considerably more complex.)

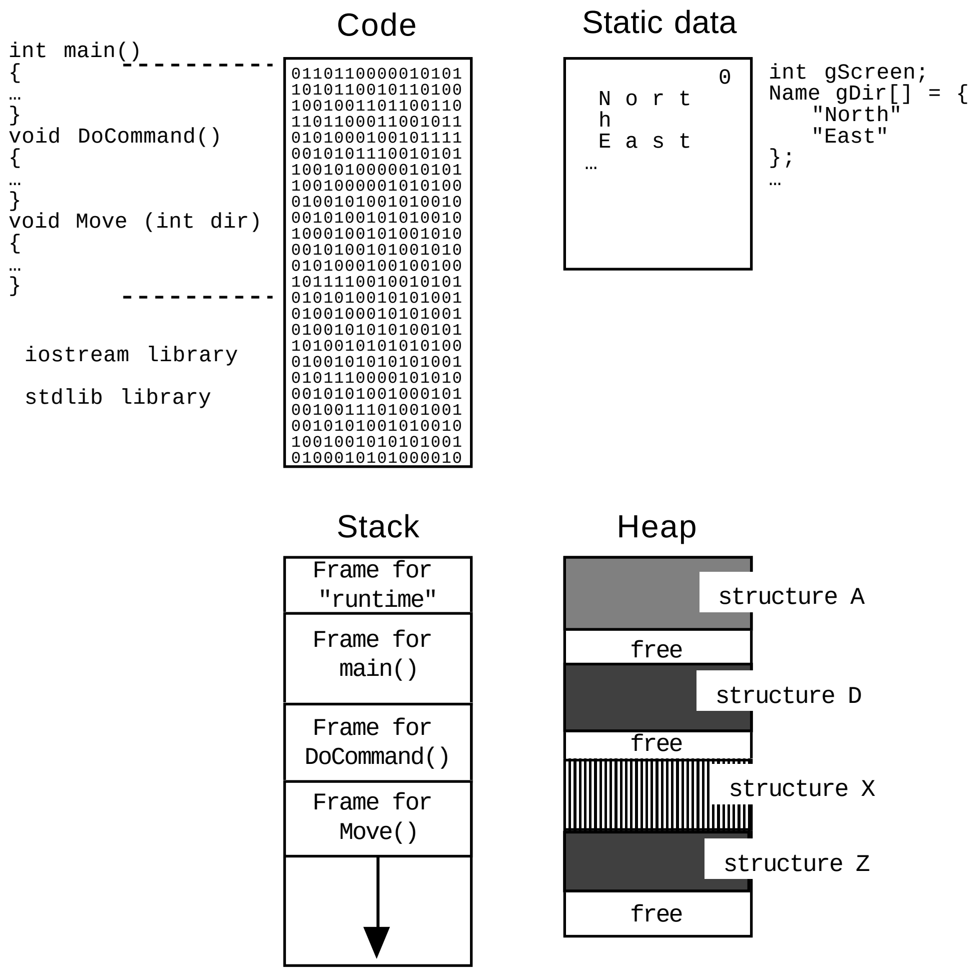

Figure 20.1 Program "segments" in memory: code, static data, stack, and "heap".

Code segment

The "code" segment contains the bit patterns for the instructions. The contents of the code segment are composed largely by the compiler; the linking loader finalises some of the addresses needed in instructions like function calls. The code segment will include library routines that have been linked with the code specifically written for the program. On machines with "memory protection" hardware, the code segment will be effectively "read only".

"Static data segment"

The "static data segment" is used for those data variables that are global or at least filescope. These are the variables that are defined outside of the body of a function. (In addition, there may be some variables that have been defined within functions, and which have function scope, but which have been explicitly allocated to the static data segment. Such variables are rarely used; there are none in this text book.) Space for variables in the static data segment is allocated by the linking loader. The variables are initialized by a run-time support routine prior to entry to `main()`. In some cases, special "at exit" functions may manipulate these variables after a return from `main()` (i.e. after the program is nominally finished!). Such variables remain in existence for the entire duration of the program (i.e. their lifetime exceeds that of the program).

Stack

The stack holds stack frames. The stack frames hold the local variables of a function together with the housekeeping details needed to record the function call and return sequence. A stack frame is created as a function is called and is freed on exit. Local variables of a function remain in existence during the execution of their own function and all functions that it calls.

The "heap"

The heap is a completely separate region of memory controlled by a run-time "memory manager" support routine. (This is not the operating system's memory manager that sorts out what space should be given to different programs. This run-time memory manager is a library function linked to your code.) This run-time memory manager handles requests for fixed sized blocks of memory.

"new" operator

In C it is normal to make requests direct to the memory manager specifying the size of blocks in terms of the number of bytes required. C++ has provided a higher level interface through the `new` operator explained below. Using `new`, a function can request the creation of a heap-based struct, a class instance, an array of variables of simple types (e.g. an array of characters) or even an array of class instances. The `new` operator works out the number of bytes needed and deals with all the other low level details associated with a call to the actual run-time memory manager.

When a program starts, the operating system gives it some amount of memory for its heap segment. The amount obviously varies with the system, but typical initial allocations would be in the range from a quarter megabyte to eight megabytes. On some systems, a program may start with a small area for its heap but is able to request that the OS enlarge the heap later. One of the start up routines would record details of the heap allocation and mark it all as "free".

Allocating space for data structures

When asked for a block of memory, the memory manager will search the heap looking for a "free" area that is large enough provide the space required and hold a little additional housekeeping information. The memory manager needs to keep track of the space it allocates. As illustrated in Figure 20.2, its "housekeeping records" are placed as "headers" and "trailers" in the bytes just before and just after the space reserved for a new structure. These housekeeping records note the size of the block and mark it as "in use". When the memory manager function has finished choosing the space to be allocated and has filled in its records, it returns the address of the start of the data area.

Freeing unneeded structures

Structures allocated on the heap eventually get discarded by the program (as when, in the example, the aircraft land or crash). Even if your programs starts with an eight megabyte heap, you will eventually run out of memory if you simply created, used, and then discarded the structures.

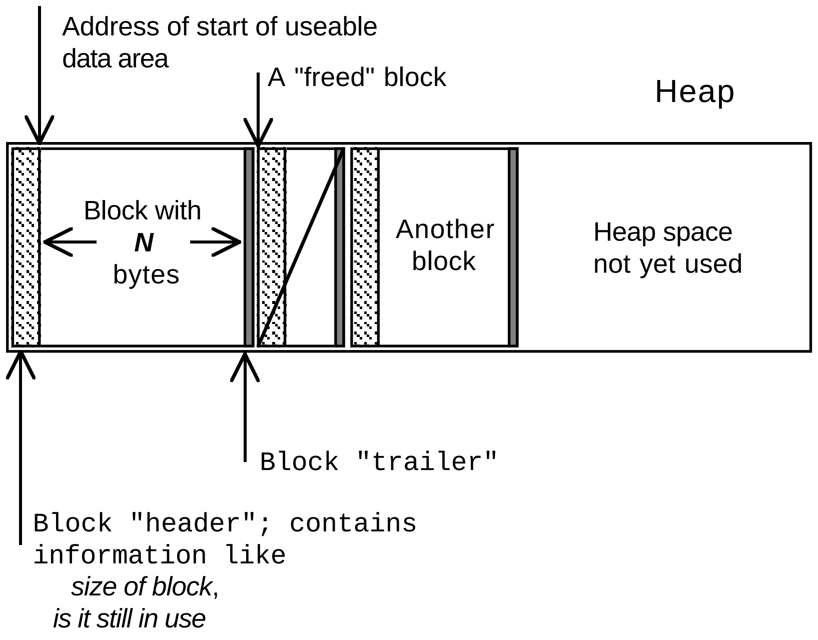
Address of start of useable data area
A "freed" block
Heap
Block with N bytes
Another block
Heap space not yet used
Block "trailer"
Block "header"; contains information like size of block, is it still in use

Figure 20.2 "Blocks" allocated on the heap.

delete operator

If you create dynamic structures in the heap, you are expected to give them back to the memory manager when you no longer need them. The memory manager can then reclaim the space they occupied. The memory manager will mark their headers as "free". Subsequently, these blocks may get reallocated or merged with neighboring free blocks. In C++, you pass discarded data structures back to the memory manager using the `delete` operator.

"Memory leaks"

It is common for programmers to be a bit careless about giving discarded data structures back to the memory manager. Some dynamically allocated structures just get forgotten. Such structures become "dead space" in the heap. Although they aren't used they still occupy space. A part of the program's code that creates, uses, but fails to delete structures introduces a "memory leak". If that code gets called many times, the available heap space steadily declines ("my memory appears to be leaking away").

Failure to allocate memory

If your code has a memory leak, or if you are allocating exceptionally large structures you may eventually run out of heap space. The C++ language has defined how the memory manager should handle situations where it is asked to create a new structure and it finds that there is insufficient space. These language features are a bit advanced; you will get to them in later studies. Initially, you can assume that requests for heap space will always succeed (if a request does fail, your program will be terminated and an error message will be printed).

Overheads when allocating heap structures

The headers and trailers added by the memory manager would typically come to about 16 bytes, maybe more. This is a "space overhead" associated with every structure allocated in the heap. A program that tries to allocate individual char or int variables in the heap almost certainly has a fundamental design error. The heap is meant to be used for creation of reasonable sized data objects like structs or arrays.

The work that the memory manager must perform to handle calls via `new` and `delete` is non-trivial. Quite commonly, profiling a program will reveal that a measurable percentage of its time is spent in the memory management routines. You should avoid creating and destroying structures inside deeply nested loops. The heap is meant to be used for the creation of data objects that have reasonable lifetimes.

Using the new operator

The following illustrate use of the `new` operator to create structures in the heap:

```
struct Point3d { double fX, fY, fZ, fR, fTheta, fPhi; }
class Bitmap;  // as declared in Chapter 19
class Number; // as declared in Chapter 19
…

…   =      new Point3d;
…
…   =      new Bitmap;
…
…   =      new Number("77777666555");
…
…   =      new char[kMAX];
```

Each of these uses of the `new` operator results in the return of the address of the start of a newly allocated block of memory in the heap. These address values must be assigned to pointer variables, as explained in the next section.

The first example creates a `Point3d` data structure. The data block allocated in the heap would be just the right size to hold six double precision numbers.

The second example does a little more. The memory manager would allocate block of heap space sufficient to hold a bit map object (an instance of class `Bitmap` from chapter 19). Class `Bitmap` has a constructor function that initializes a bit map. The code generated for the `new` operator has a call to this constructor function so as to initialize the newly allocated data area.

The third example is similar, except that it involves an instance of class `Number` . Class `Number` has several possible constructors; the one needed here is the one that takes a character string. The code generated for the `new` operator would include a call to that constructor so the newly allocated `Number` would be correctly initialized to a value a little over seventy seven thousand million.

new [] operator

The final example creates an array of characters. (The size of the array is determined by the value in the `[ ]` brackets. Here the value is a constant, but an expression is allowed. This make it possible to work out at run time the size of the array needed for some specific data.) Technically, this last example is using a different operator. This is the `new []` operator (the "make me an array operator").

It is quite common for a program to need to create an array of characters; some examples will occur later.

Illustrations of uses of the `delete` (and `delete []`) operators come toward the end of the next section after pointer variables have been discussed.

20.2 POINTERS

20.2.1 Some "pointer" basics

Defining pointer variables

Pointers are a derived data type. The pointyness, represented by a '*', is a modifier to some basic data type (either built in like int or a programmer defined struct or class type). The following are definitions of pointer variables:

```
int         *ptr1;
char        *ptr2;
Bitmap      *ptr3;
Aircraft    *ptr4;
```

These definitions make `ptr1` a pointer to a data element that is an int; `ptr2` is a pointer to a character; `ptr3` is a pointer to a `Bitmap` object; and `ptr4` is a pointer to an `Aircraft` object. Each of these pointer variables can be used to hold an address; it has to be the address of a data variable of the specified type.

Pointers are type checked to the same degree as anything else is in C++. If you want to store a value in `ptr1`, the value will have to be the address of a variable that is an integer.

Definitions of pointer variables can cause problems. Some of the problems are due to the free format allowed. As far as a C++ compiler is concerned, the following are identical:

```
int         *ptr1;
int*        ptr1;
int    *    ptr1;
```

but strictly the `*` belongs with the variable name. It does matter. Consider the following definition:

```
int*        pa, pb;
```

What are the data types of `pa` and `pb`?

In this case pa is a pointer to an integer (something that can hold the address of an integer variable) while pb is an integer variable. The * belongs on the variable name; the definition really is `int *pa, pb;`. If you wanted to define two pointers you would have to write `int *pa, *pb;`.

Although the "pointyness" qualifier * associates with a variable name, we need to talk about pointer types independent of any specific instance variable. Thus, we will be referring to `int*` pointers, `char*` pointers, and `Aircraft*` pointers.

Pointers and arrays

At the end of the last section, there was an example that involved creating an array of characters on the heap. The address returned by `new char[10]` has the type "address of array of characters". Now `char*` is a pointer to a character can therefore hold the address of a character. What would be the correct type declaration for a pointer to an array of characters (i.e. something that can hold the address of an array of characters)?

For reasons partly explained in 20.5, a pointer to an array of characters is also `char*`. This makes reading code a bit more difficult. If you see a variable of pointer type being defined you don't know whether it is intended to hold the address of a single instance of the specified data type or is meant to be used to refer to the start of an array of data elements.

"Generic pointers"

Although pointers have types, you quite often need to have functions that use a pointer to data of arbitrary type. A good example is the low-level `write()` function. This function needs a pointer to the memory area that contains the data to be copied to disk and an integer specifying the number of bytes to be copied. The actual write operation involves just copying successive bytes from memory starting at the address specified by the pointer; the same code can work for any type of data. The `write()` function will accept a pointer to anything.

Originally in C, a `char*` was used when the code required a "pointer to anything". After all, a "pointer to anything" must hold the address of a byte, a char is a byte, so a pointer to anything is a `char*`. Of course, this just increases the number of possible interpretations of `char*`. It may mean a pointer to a character, or it may mean a pointer to an array of characters, or it may mean a pointer to unspecified data.

void* pointer type

These days, the special type `void*` is generally preferred when a "pointer to anything" is needed. However, a lot of older code, and almost all the older C libraries that you may use from C++, will still use `char*`.

Pointer casts

C++ checks types, and tries to eliminate errors that could arise if you assign the wrong type of data to a variable. So, C++ would quite reasonably object to the following:

```
char        *aPtr;
…
aPtr = new Point3d;
```

Here the `new` operator is returning "address of a `Point3d`", a `char*` is something that holds an "address of a character". The type "address of a `Point3d`" is not the same as "address of a character". So, the assignment should be challenged by the compiler, resulting in at least a warning if not an error message.

But you might get the same error with the code:

```
Point3d     *bPtr;
…
bPtr = new Point3d;
…
theOutputFile.write(bPtr, sizeof(Point3d));
```

Function `write()` requires a pointer with the address of the data object, you want it to write the contents of the `Point3d` whose address is held in `bPtr`. You would get an error (depends on your compiler and version of iostream) if the function prototype was something like:

```
write(char*, int);
```

The compiler would object that the function wanted a `char*` and you were giving it a `Point3d*`.

In situations like this, you need to tell the compiler that you want it to change the interpretation of the pointer type. Although the `bPtr` really is a "pointer to a `Point3d`" you want it to be treated as if it were a "pointer to `char`".

Casting to char* or void*

You achieve this by using a "type cast":

```
char        *aPtr;
Point3d     *bPtr;
…
…
bPtr = new Point3d;
…
theOutputFile.write((char*)bPtr, sizeof(Point3d));

aPtr = (char*) new Point3d;
```

The construct:

```
(char*)
    (some address value from a pointer, a function,
           or an operator)
```

tells the compiler to treat the address value as being the address of a character. This allows the value to be assigned to a `char*` variable or passed as the value of a `char*` argument.

If a function requires a `void*` argument, most compilers allow you to use any type of pointer as the actual argument. A compiler allowing this usage is in effect putting a `(void*)` cast into your code for you. Occasionally, you might be required to put in an explicit `(void*)` cast.

Casting from specific pointer types like `Aircraft*`, `Point3d*`, or `Bitmap*` to general types like `char*` and `void*` is safe. Casts that convert general pointers back to specific pointer types are often necessary, but they do introduce the possibility of errors.

Casting a void* to a specific pointer type

In Chapter 21, we look at a number of general purpose data structures that can be used to hold collections of data objects. The example collection structures don't store copies of the information from the original data objects, instead they hold pointers to the data objects. These "collection classes" are intended to work with any kind of data, so they use `void*` data pointers. There is a difficulty. If you ask the object that manages the collection to give you back a pointer to one of the stored data objects you are given back a `void*` pointer.

Thus you get code like the following:

```
class Job {
public:
    Job(int codenum, Name customer, ....);
…
    int    JobNumber(void) const;
…
};

class Queue {
public:
    …
    void Append(void* ptr_to_newitem);
    int    Length(void) const;
    void   *First(void);
    …
};

// make a new job and add it to the Queue
Job*       j = new Job(worknum++, cName, ...);
…
theQueue.Append(j);
…
// Look at next queued job
```

```
      ?? = theQueue.First();
```

The `Queue` object returns a `void*` pointer that holds the address of one of the `Job` objects that was created earlier. But it is a `void*` pointer. You can't do anything much with a `void*` .

A type cast is necessary:

```
   Job *my_next_task = (Job*) theQueue.First();
   cout << "Now working on " << my_next_task->JobNumber() << endl;
```

A cast like this is perfectly reasonable and safe provided the program is properly designed. The programmer is telling the compiler, "you think it could be a pointer to any kind of data, I *know* that it is a pointer to a `Job` object, let me use it as such".

These casts only cause problems if there are design flaws. For example, another programmer might incorrectly imagine that the queue held details of customers rather than jobs and write code like:

```
   Customer*   c;
   // get customer from queue, type cast that void*
   c = (Customer*) theQueue.First();
```

The compiler has to accept this. The compiler can't tell that this programmer is using the queue incorrectly. Of course, the second programmer will soon be in difficulties with code that tries to treat a `Job` object as if it were a `Customer` object.

When you write or work with code that type casts from general (`void*`) to specific (`Job*` or `Customer*`) you should always check carefully to verify the assumptions being made in relation to the cast.

Null pointers and uninitialized pointers

NULL

Pointers don't have any meaningful value until you've made them point somewhere! You make a pointer variable point somewhere by assigning a value; in C++ this will most often be a value returned by the `new` operator. There is a constant, `NULL`, defined in several of the header files, that represents the concept of "nowhere". You can assign the constant `NULL` to a pointer variable of any type:

```
   char        *ptr1 = NULL;
   Aircraft    *ptrA = NULL;
   …
```

Often you will be working with collections of pointers to data items, a pointer whose value is `NULL` is frequently used to mark the last element of the collection. The loops that control working through the collection are set up so that they stop on finding a

`NULL` pointer. These `NULL` pointers serve much the same role as "sentinel values" used in loops that read input (as discussed in Chapter 9).

You can test whether a pointer is `NULL` using code like:

```
if(ptrA != NULL) {
    // Process Aircraft accessed via ptrA
    …;
    }
```

or:

```
if(ptrA) {
    // Process Aircraft accessed via ptrA
    …;
    }
```

In effect, `NULL` equates to 0 or "false". The second form is extremely common; the first version is actually slightly clearer in meaning.

Global and filescope pointer variables are initialized to `NULL` by the linking-loader. Automatic pointer variables, those defined as local to functions, are not normally initialized. Their initial contents are arbitrary; they contain whatever bit pattern was in memory at the location that corresponds to their place in the function's stack frame.

Beware of uninitialized pointers

An amazingly large proportion of the errors in C and C++ programs are due to programmers using pointers that have never been set to point anywhere. The arbitrary bit pattern in an uninitialized pointer may represent an "illegal address" (e.g. address -5, there is no byte whose address is -5). These illegal addresses are caught by the hardware and result in the operating system stopping the program with an error such as "segmentation fault", "bus error", "system error 2" etc.

Other uninitialized pointers may by chance hold addresses of bytes in one of the program's segments. Use of such an address may result in changes to arbitrary static, stack-based, or heap-based variables or even overwriting of code. Such errors can be quite hard to track down because the problems that they cause frequently don't show up until long after the time that the uninitialized pointer was used. Fortunately, modern compilers can spot many cases where code appears to be using an uninitialized variable; these cases result in warning messages that must be acted on.

Input and output of pointers?

No input, and not much output!

You can get the value of a pointer printed. This is sometimes useful for debugging purposes (the output format usually defaults to hex: for pointers):

```
cout << "ptrA now holds address " << hex << ptrA << endl;
```

In some cases you will need to convert the pointer to a long integer, e.g. `long(ptrA)`.

There is no purpose in writing the value of a pointer to a disk file (nor of writing out a structure that contains pointer data members). The data written would be useless if read back in on a subsequent run of the program.

The value in a pointer is going to be an address, usually one chosen by the run-time memory manager and returned by the `new` operator. If you run the program another time, the run-time memory manager might start with a different area of memory to work with and almost certainly will allocate data areas differently. Consequently, data objects end up at quite different places in memory on different runs. Yesterday's addresses are no use.

20.2.2 Using pointers

Assignment of pointers

Consider the following code fragment:

```
class Demo {
public:
    Demo(int i, char c);
    …
private:
    int    fi;
    char   fc;
};

Demo::Demo(int i, char c) { fi = i; fc = c; }

int main()
{
    Demo   *ptr1 = NULL;
    Demo   *ptr2 = NULL;
// stage 1
    ptr1 = new Demo(6, 'a');
    ptr2 = new Demo(7, 'b');
// stage 2
    ptr2 = ptr1;
// stage 3
    …
}
```

The situation in memory after each stage is illustrated in Figure 20.3.

At the end of stage 1, the stack frame for `main()` has been built in the stack and the two pointer variables are both `NULL`; the heap is empty. In stage 2, the two data structures are built in the heap; the addresses returned by `new` are copied into the pointer variables so that these now "point to" their heap structures.

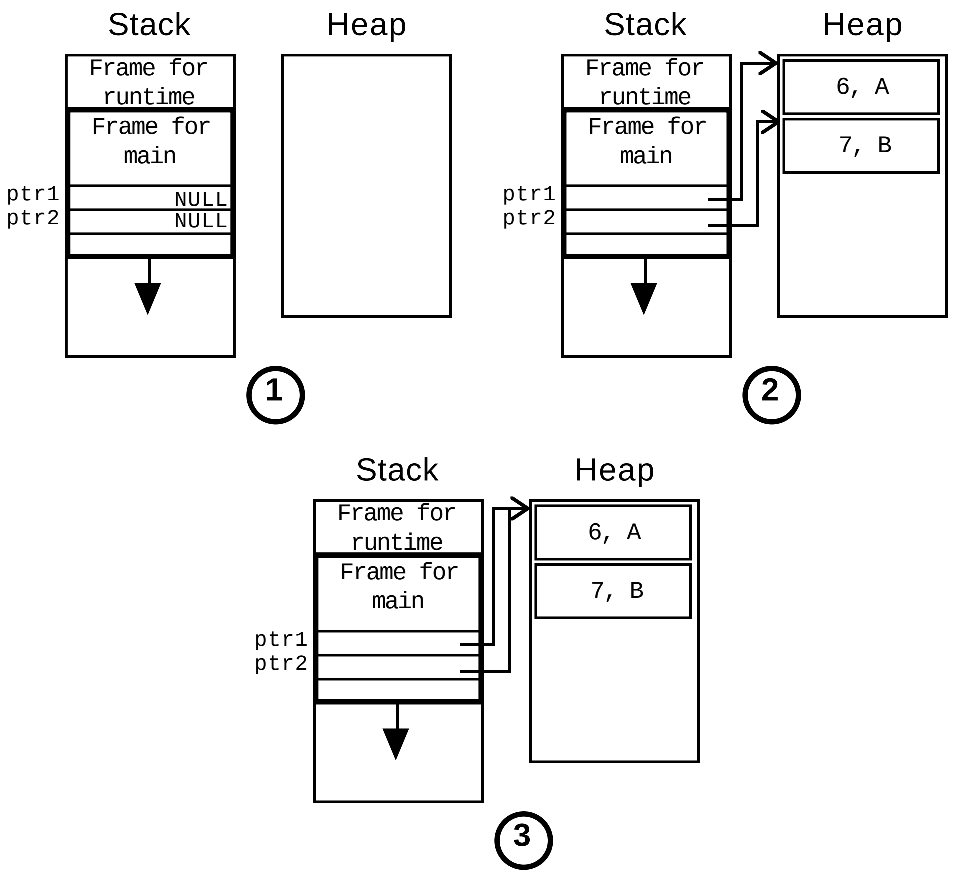

Figure 20.3 Pointer assignment.

In stage 3, the contents of `ptr1` (the address of the `Demo` object with values 6, A) is copied into `ptr2`. This makes both pointers hold the same address and so makes them point to the same object. (Note that this code would have a memory leak; the second `Demo` object, 7,B, has been abandoned but remains in the heap.)

Assignment of pointer variables simply means copying an address value from one to another. The data addressed by the pointers are not affected.

Using pointers to access the data members and member functions of structures

If a structure can be accessed by a pointer, its data members can be manipulated. C and C++ have two styles by which data members can be referenced.

The more common style uses the `->` (data member access) operator:

The -> operator

```
struct Thing {
    int           fNum;
    double        fD;
    char          fX;
};

int main()
{
    Thing   *pThing;
    pThing = new Thing;
    pThing->fNum = 17;
    pThing->fX = '?';
    pThing->fD = 0.0;
    …
    …
    if(pThing->fNum < kLIM)
           …;
    …
    xv += pThing->fD;
```

The `->` operator takes the name of a (typed) pointer variable on its left (e.g. `pThing`, a `Thing*` pointer), and on its right it takes the name of a data member defined as part of that type (e.g. `fX`; the compiler checks that `fX` is the name of a data member of a `Thing`). The `->` operator produces an address: the address of the specified data member of a structure starting at the location held in the pointer. (So `pThing->fNum` would typically return the address held in `pThing`, `pThing->fD` would return an address value 4 greater, while `pThing ->fX would` return an address value 12 greater than the starting address.)

If the expression involving the `->` operator is on the left side of an `=` assignment operator (i.e. it is an "lvalue"), the calculated address specifies where something is to be stored (e.g. as in `pThing->fX = '?'`, where the address calculated defines where the '?' is to be stored). Otherwise, the address is interpreted as the place from where a data value is to be fetched (e.g. as in `if(pThing->fNum …)` or `+= pThing->fD;`).

There is a second less commonly used style. The same operations could be coded as follows:

(*). operator combination

```
    (*pThing).fNum = 17;
    (*pThing).fX = '?';
    (*pThing).fD = 0.0;
    …
    …
    if((*pThing).fNum < kLIM)
           …;
    …
    xv += (*pThing).fD;
```

This uses `*` as a "dereferencing" operator. "Dereferencing" a pointer gives you the object pointed to. (Lots of things in C and C++ get multiple jobs to do; we've seen '`&`' work both as a bit wise "And" operator and as the "address of" operator. Now its `*`'s turn; it may be an innocent multiply operator, but it can also work as a "dereferencing" operator.)

Dereferencing a pointer gives you a data object, in this case a `Thing` object. A `Thing` object has data members. So we can use the "." data member selection operator to chose a data member. Hence the expressions like `(*pThing).fD`.

Calling member functions

If you have a pointer to an object that is an instance of a class, you can invoke any of its member functions as in the example in 20.2.1:

```
Job *my_next_task;
…
cout << "Now working on " << my_next_task->JobNumber() << endl;
```

Manipulating complete structures referenced by pointers

Though you usually want to access individual data members (or member functions) of an object, you sometimes need to manipulate the object as a whole.

You get the object by dereferencing the pointer using the `*` operator. Once you have the object, you can do things like assignments:

```
int main()
{
    Demo   *ptr1 = NULL;
    Demo   *ptr2 = NULL;
    ptr1 = new Demo(6, 'a');
    ptr2 = new Demo(7, 'b');

// Change the contents of second Demo object to make it
// identical to the first
    *ptr2 = *ptr1;
    …
}
```

The `*ptr2` on the left side of the `=` operator yields an address that defines the target area for a copying (assignment) operation. The `*ptr1` on the right hand side of the = operator yields an address that is interpreted as the address of the source of the data for the copying operation.

****this***

In the example on class `Number`, we sometimes needed to initialize a new `Number` (`result`) to the same value as the current object. The code given in the last chapter used an extra `CopyTo()` member function. But this isn't necessary because we can code the required operation more simply as follows:

```
Number Number::Subtract(const Number& other) const
```

```
{
    Number result;
    result = *this;

    if(other.Zero_p()) return result;

    …
}
```

The implicitly declared variable `this` is a `Number*`. It has been initialized to hold the address of the object that is executing the `Subtract()` function. If we want the object itself, we need to dereference `this` (hence `*this`). We can then directly assign its value to the other `Number result`.

Alternatively we could have used the copy constructor `Number(const Number&)`. This function has to be passed the `Number` that is to be copied. So we need to pass `*this`:

```
Number Number::Subtract(const Number& other) const
{
    Number result(*this);

    if(other.Zero_p()) return result;
    …
}
```

Working with a pointer to an array

The following code fragment illustrates how you could work with an array allocated in the heap. The array in this example is an array of characters. As explained previously, the pointer variable that is to hold the address returned by the `new []` ("give me an array" operator) is just a `char*`. But once the array has been created, we can use the variable as if it were the name of a character array (i.e. as if it had been defined as something like `char ptrC[50]`).

```
#include <iostream.h>

int main()
{
    cout << "How big a string do you want? ";
    int len;
    cin >> len;

    char *ptrC = new char[len];
    for(int i = 0; i < len-1; i++)
           ptrC[i] = '!';
    ptrC[len - 1] = '\0';
    cout << "Change some letters:" << endl;
```

```
        for(;;) {
                int lnum;
                char ch;
                cout << "# ";
                cin >> lnum;
                if((lnum < 0) || (lnum >= len-1)) break;

                cout << "ch : ";
                cin >> ch;

                ptrC[lnum] = ch;
                }

        cout << "String is now ";
        cout << ptrC;
        cout << endl;

        delete [] ptrC;
        return 0;
}
```

Note that the programmer remembered to invoke the `delete []` operator to get rid of the array when it was no longer required.

Awful warning on dynamically allocated arrays

The code shown makes certain that character change operations are only attempted on characters that are in the range `0 … N-2` for a string of length `N` (the last character in position `N-1` is reserved for the terminating '\0'). What would happen if the checks weren't there and the code tried to change the -1th element, or the 8th element of an array 0…7?

The program would go right ahead and change these "characters". But where are they in memory?

If you look at Figure 20.2, you will see that these "characters" would actually be bytes that form the header or trailer housekeeping records of the memory manager. These records would be destroyed when the characters were stored.

Now it may be a long time before the memory manager gets to check its housekeeping records; but when it does things are going to start to fall apart.

Bugs related to overwriting the ends of dynamically allocated arrays are very difficult to trace. Quite apart from the delay before any error is detected, there are other factors that mean such a the bug will be intermittent!

It is rare for programs to have mistakes that result in negative array indices so overwriting the header of a block containing an array is not common. Usually, the errors relate to use of one too many data element (and hence overwriting of the trailer record). But often, the trailer record isn't immediately after the end of the array, there is a little slop of unused space.

The program will have asked for an array of 40 bytes; the memory manager may have found a free block with 48 bytes of space (plus header and trailer). This is close enough; there is no need to find something exactly the right size. So this block gets returned and used. An overrun of one or two bytes won't do any damage.

But the next time the program is run, the memory manager may find a free block of exactly the right size (40 bytes plus header and trailer). This time an overrun causes problems.

Be careful when using dynamically allocated arrays!

Deleting unwanted structures

delete operator

If you have a pointer to a dynamically allocated struct or class instance you don't want, simply invoke the `delete` operator on that pointer:

```
Aircraft *thePlane;
…
if(thePlane->OutOfFuel())
    delete thePlane;
```

The `delete` operator passes details back to the memory manager which marks the space as free, available for reallocation in future.

The pointer isn't changed. It still holds the value of the now non-existent data structure. This is dangerous. If there is an error in the design of the code, there may be a situation where the data object is accessed after it is supposed to have been deleted. Such code will usually appear to work. Although the memory area occupied by the data object is now marked as "free" it is unlikely to be reallocated immediately; so it will usually contain the data that were last saved there. But eventually, the bug will cause problems.

It is therefore wise to change a pointer after the object it points to is deleted:

```
if(thePlane->OutOfFuel()) {
    delete thePlane;
    thePlane = NULL;
    }
```

Setting the pointer to `NULL` is standard. Some programmers prefer to use an illegal address value:

```
thePlane = (Aircraft*)0xf5f5f5f5
```

Any attempt to reuse such a pointer will immediately kill the program with an address error; such an automatic kill makes it easier to find the erroneous code.

delete [] operator

The `delete []` operator should be used to free an array created using the `new []` operator.

20.2.3 Strings and hash tables revisited

These new pointer types permit slightly better solutions to some of the examples that we have looked at previously.

Strings

Section 11.7 introduced the use of character arrays to hold constant strings and string variables. Things like `Message` were defined by typedefs:

```
typedef char Message[50];
```

(making `Message` a synonym for an array of 50 characters) and arrays of `Message` were used to store text:

```
Message gMessage[] {
    "Undefined symbol",
    "Illegal character",
    …
    "Does not compute"
};
```

Similar constructs were used for arrays of keywords, or menu options, and similar components introduced in Chapter 12 and later chapters.

These things work. But they are both restricting and wasteful. They are restricting in that they impose a maximum length on the keywords or menu messages. They are wasteful in that usually the vast majority of the keywords are very much shorter than the specified size, but each occupies the same amount of memory.

Sometimes it is helpful to have all data elements the same size. For example, if we want to write some `Messages` to a binary disk file it is convenient to have them all the same size. This gives structure to the file (we can find a specific entry in the `Message` file) and simplifies input.

But if the data are only going to be used in main memory, then there is less advantage in having them the same size and the "wasted space" becomes a more important issue.

The following structure would take up much less overall memory than the `gMessage[]` table defined above:

```
char       *gMsgPtrs[] = {
    "Undefined symbol",
    "Illegal character",
    …
    "Does not compute"
};
```

Figure 20.4 illustrates how this `gMsgPtrs[]` structure might represented in memory. This representation has the overhead of a set of pointers (estimate at 4 bytes each) but each individual message string occupies only the amount of space that it needs instead of the 50 bytes previously allocated. Usually, this will result in a significant saving in space.

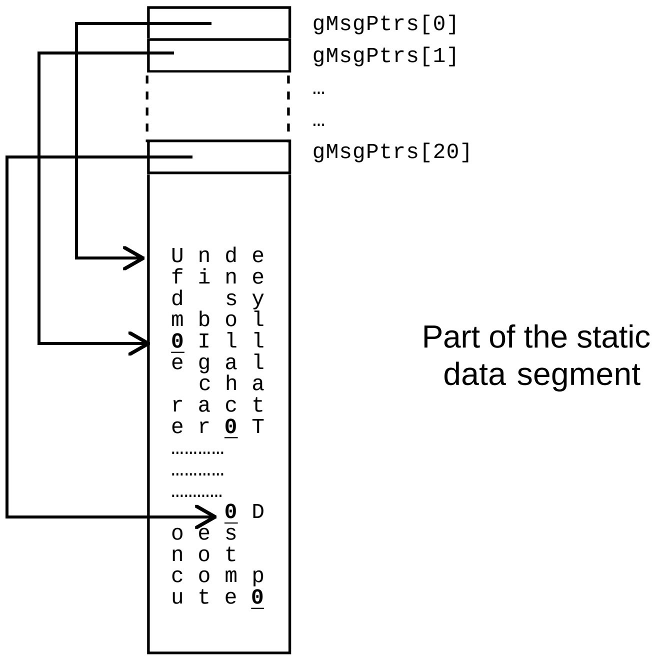
gMsgPtrs[0]
gMsgPtrs[1]
…
…
gMsgPtrs[20]
U n d e f i n e d s y m b o l 0 I l l e g a l c h a r a c t e r 0 T
…
0 D o e s n o t c o m p u t e 0
Part of the static data segment

Figure 20.4 Representing an array of initialized character pointers.

Pointers to const data

The messages referred to by the `gMsgPtrs[]` array would probably be intended as constants. You could specify this:

```
const char *gMsgPtrs[] = {
    "Undefined symbol",
    "Illegal character",
    …
    "Does not compute"
};
```

This makes `gMsgPtrs[]` an array of pointers to constant character strings. There will be other examples later with definitions of pointers to constant data elements.

Many standard functions take pointers to constant data as arguments. This is very similar to const reference arguments. The function prototype is simply indicating that it doesn't change the argument that it can access by the pointer.

Fixed size character arrays are wasteful for program messages and prompts, and they may also be wasteful for character strings entered as data. If you had a program that had to deal with a large number of names (e.g. the program that sorted the list of pupils according to their marks), you could use a struct like the following to hold the data:

```
struct pupil_rec {
    int    fMark;
    char   fName[60];
};
```

but this has the same problem of wasting space. Most pupils will have names less than sixty characters (and you are bound to get one with more than 60).

You would be better off with the following:

```
struct Pupil_Rec {
    int    fMark;
    char   *fName;
};
```

The following example code fragment illustrates creation of structs and heap-based strings:

```
Pupil_Rec *GetPupilRec(void)
{
    cout << "Enter mark, or -1 if no more records" << endl;
    int mark;
    cin >> mark;
```

Return NULL if no structure needed

```
    if(mark < 0)
           return NULL;
```

Create structure if necessary

```
    Pupil_Rec *result = new Pupil_Rec;
    result->fMark = mark;
    cin.ignore(100,'\n');
    cout << "Enter names, Family name, then given names"
                  << endl;
```

Read string into temporary "buffer"

```
    char buffer[200];
    cin.getline(buffer,199,'\n');

    int namelen = strlen(buffer);
```

Make character array of required size and link to struct

```
    result->fName = new char[namelen+1];

    strcpy(result->fName, buffer);
```

Fill in character array

```
    return result;
}
```

This function is defined as returning a pointer to a `Pupil_Rec`; it will have to create this `Pupil_Rec` structure on the heap using the `new` operator. If the mark input is negative, it means there is no more input; in this case the function returns `NULL`. This is a very typical style in a program that needs to create a number of data records based on input data. The calling program can have a loop of the form `while((rec = GetPupilRec()) != NULL) { … }`.

If the mark is not negative, another `Pupil_Rec` struct is created using the `new` operator and its `fMark` field is initialized. Any trailing input following the mark is removed by the call to `ignore()`. The function then prompts for the pupil's names.

There has to be some character array allocated into which the name can be read. This is the role of `buffer`. It gets filled with a complete line from the input.

The number of characters needed for the name is then determined via the call to `strlen()`. A character array is allocated on the heap using `new []` (the character array is one longer than the name so as to leave room for a terminating '\0' character). The name is then copied into this array.

The new structure has been built and so it can be returned. Note how every structure is represented by two separate blocks of information in the heap, with the character array linked to the main `Pupil_Rec` struct. In most of your later programs you will have hundreds if not thousands of separate objects each stored in some block of bytes on the heap. Although separately allocated in the heap, these objects are generally linked together to represent quite elaborate structural networks.

Hash table

Sections 18.2.2 and 18.2.3 contained variations on a simple hash table structure. One version had an array of with fixed 20 character words, the other used an array of structs incorporating fixed sized words and an integer count. These arrays again "wasted" space.

Ideally, a hash table should be not much more than half occupied; so many, perhaps almost half of the table entries will not be used even when all the data have been loaded. Most of the words entered into the hash table would have been less than twelve characters so much of the space allocated for each word is wasted (and of course the program can't deal with the few words that do exceed twenty characters).

A slightly more space-efficient variation would use a table of pointers to character arrays (for the version that is to store words) or pointers to structs for the version that needs words and associated integer data values). This arrangement for the table for words would be as shown in Figure 20. 5 (compare with version in Figure 18.1). Note that the words represented as separately allocated arrays in the heap do require those headers and trailers. The space costs of these cut into the savings made by only using the number of characters required for each word. The only significant saving is going to be that due to the fact that only half the table is populated.

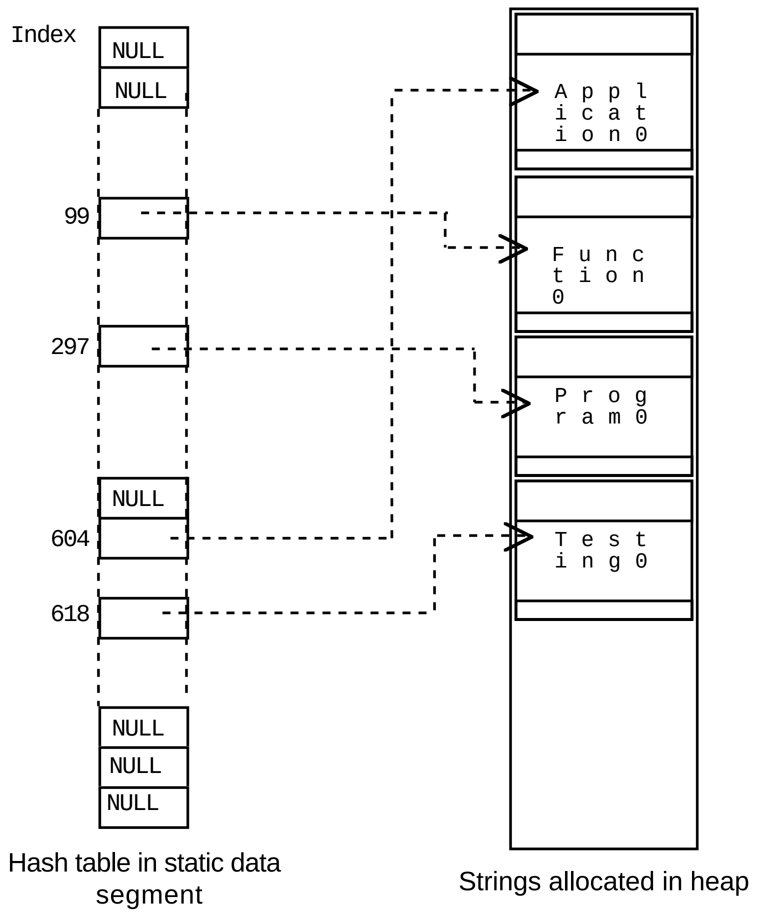

Figure 20.5 Hash table using pointers to strings.

The changes to the code needed to use the modified structure aren't substantial. The following are rewrites of a couple of the functions given in Section 18.2.

```
const int kTBLSIZE       = 1000;

char *theTable[kTBLSIZE];

void InitializeHashTable(void)
{
    for(int i=0; i< kTBLSIZE; i++)
            theTable[i] = NULL;
}
```

```
int NullEntry(int ndx)
{
    return (theTable[ndx] == NULL);
}
```

Functions `InitializeHashTable()` and `NullEntry()` both have minor changes to set pointers to `NULL` and to check for `NULL` pointers.

The code for `MatchEntry()` is actually unchanged! There is a subtle difference. Previously the argument was an array of characters, now it is a pointer to an array of characters. But because of the general equivalence of arrays and pointers these can be dealt with in the exact same manner.

```
int MatchEntry(int ndx, const char str[])
{
    return (0 == strcmp(str, theTable[ndx]));
}
```

The `InsertAt()` function has been changed so that it allocates an array on the heap to store the string that has to be inserted. The string's characters are copied into this new array. Finally, the pointer in the table at the appropriate index value is filled with the address of the new array.

```
void InsertAt(int ndx, const char str[])
{
    char   *ptr;
    ptr = new char[strlen(str) + 1];
    strcpy(ptr,str);
    theTable[ndx] = ptr;
}
```

20.3 EXAMPLE: "AIR TRAFFIC CONTROLLER"

Problem

Implement the simulated "Air Traffic Control" (ATC) trainer/game described below.

The ATC trainer is to give users some feel for the problems of scheduling and routing aircraft that are inbound to an airport. Aircraft are picked up on radar at a range of approximately150 miles (see Figure 20.6). They are identified by call sign, position, velocity, acceleration, and details of the number of minutes of fuel remaining. An aircraft's velocity and acceleration are reported in terms of x', y', z' and x", y", z" components. The user (game-player, trainee air-controller, or whatever) must direct aircraft so that they land successfully on the single east-west runway (they must approach from the west).

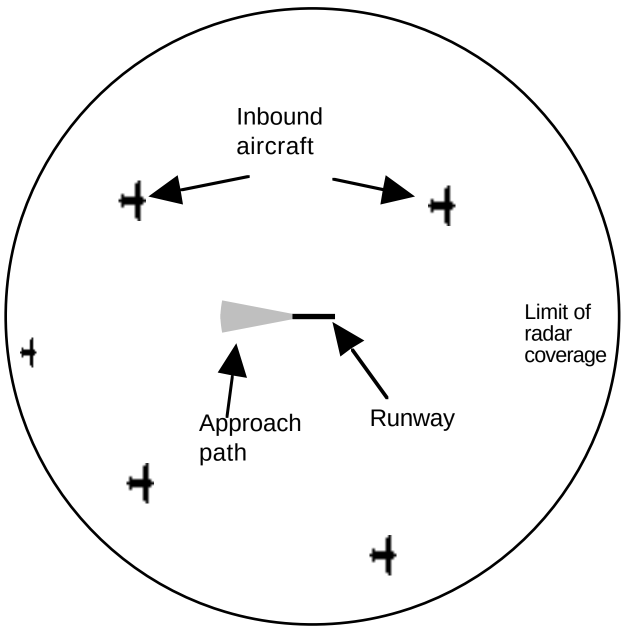

Figure 20.6 The "Air Traffic Control" problem.

There are a number of potential problems that the controller should try to avoid. If an aircraft flies to high, its engines stall and it crashes. If it flies too low it runs into one of the hills in the general vicinity of the airport. If it runs out of fuel, it crashes. If its (total) horizontal speed is too low, it stalls and crashes. If the horizontal speed is too high, or if it is descending too fast, the wings fall off and, again, it crashes.

At each cycle of the simulation, the controller may send new directives to any number of inbound aircraft. Each cycle represents one minute of simulated time. A directive specifies the accelerations (as x", y", z" components) that should apply for a specified number of minutes. Subsequent directives will override earlier settings. The aircraft pilot will check the directive. If accelerations requested are not achievable, or it is apparent that they would lead to excessive velocities within a minute, the directive is ignored. An acceptable directive is acknowledged.

At each cycle of the simulation, every aircraft recomputes its position and velocity. Normally, one unit of fuel is burnt each minute. If an aircraft has a positive vertical acceleration ("*Climb!*", "*Pull out of that dive!*", …) or its horizontal accelerations lead to an increase in its overall horizontal speed, it must have been burning extra fuel; in such cases, its remaining fuel has to be decremented by one extra unit.

Aircraft must be guided so that they line up with the approach path to the runway. If they are flying due east at an appropriate velocity, within certain height limits and with some remaining fuel, they may be handed over to the airport's automated landing system. These aircraft are considered to have been handled successfully, and are deleted from the simulation.

Details

Equations of motion for aircraft

Controlling aircraft by specifying accelerations is unnatural, but it makes it easy to write code that defines how they move. You need simply remember your high school physics equations:

u	initial velocity
v	final velocity
s	distance travelled
α	acceleration
t	time

$$v = u + \alpha * t$$
$$s = u * t + 0.5 * \alpha * t^2$$

When recomputing the position and velocity of an aircraft, the x, y (and x' , y') components can be treated separately. Just calculate the distance travelled along each axis and update the coordinates appropriately. The total ground speed is given by:

$$\text{speed} = \sqrt{((x')^2 + (y')^2)}$$

Aircraft limits

The following limits can be used for aircraft performance (pure inventions, no claims to physical reality):

Maximum height	40000 feet
Minimum safe height	450 feet
Maximum horizontal speed	12 miles per minute (mpm)
Minimum horizontal speed	2 mpm
Maximum rate of descent	600 feet per minute (fpm)
Maximum rate of ascent	1000 fpm
Maximum acceleration/ deceleration (horizontal)	2 miles per minute per minute
Maximum positive vertical acceleration	800 feet per minute per minute
Maximum negative vertical acceleration	400 fpmpm

The maximum rate of ascent is simply a limit value, you cannot go faster even if you try, but nothing disastrous happens if you do try to exceed this limit. The other speed limits are critical, if you violate them the aircraft suffers.

Landing conditions

For an aircraft to achieve a safe landing, the controller must bring it into the pick up area of the automated landing system. The aircraft must then satisfy the following constraints:

```
Approach height                 (600 … 3000) feet
Distance west of the runway            4 miles … 10 miles
Distance north/south of runway  < 0.5 miles
Fuel remaining                  > 4 minutes
x'                              +2 … +3 mpm
           (i.e. flying east 120-180mph)
y'                              -0.1…+0.1 mpm
           (minimal transverse speed)
z'                              -500 … 0 fpm
           (descending, but not too fast)
x'', y'', z''                          ≈0 (no accelerations)
```

Reports and command interface

A relatively crude status display, and command line interface for user input will suffice. For example, a status report listing details of all aircraft in the controlled space could be something like:

```
BA009  (…,…,…) (…,…,…) (…,…,…) fuel = …
JL040  (…,…,…) (…,…,…) (…,…,…) fuel = …
```

Each line identifies an aircraft's call sign; the first triple gives its x, y, z position (horizontal distances in miles from the airport, height in feet); the second triple gives the x', y', z' velocities (miles per minute, feet per minute); the third triple is the accelerations x'', y'', z'' (miles per minute per minute, feet per minute per minute); finally the number minutes of fuel remaining is given. For example:

```
BA009  (-128,4.5,17000) (4,-0.5,-1000) (0,0,0) fuel = 61
```

This aircraft is flying at 17000 feet. Currently it is 128 miles west of the airport and 4.5 miles north of the runway. Its total ground speed is approximately 4.03 miles per minute (241mph). It is descending at 1000 feet per minute. It is not accelerating in any way, and it has 61 minutes of fuel remaining.

The user has to enter commands that tell specific aircraft to accelerate (decelerate). These commands will have to include the aircraft's call sign, and details of the new accelerations. For example:

```
BA009  0 0 100 3
```

This command instructs plane BA009 to continue with no horizontal accelerations (so no change to horizontal velocities), but with a small positive vertical acceleration that is to apply for next three minutes (unless changed before then in a subsequent command). This will cause the aircraft to reduce its rate of descent.

Adding aircraft to the controlled airspace

Entry of planes into controlled air-space can be handled by a function that uses arrays of static data. One array of integers can hold the arrival times (given in minutes from the start of simulation), another array of simple structs can hold details of flight names and the initial positions and velocities of the aircraft. If this function is called once on each cycle of the simulation, it can return details of any new aircraft (assume at most one aircraft arrives per minute).

Design

The objects and their classes

To start, what are the objects needed in this program?

Aircraft are obvious candidates. The program is all about aircraft moving around, responding to commands that change their accelerations, crashing into hills, and diving so fast that their wings fall off. Each aircraft owns some data such as its call sign, its current x, y, z position, its x', y', z' velocities. Each aircraft provides all services related to its data. For example, there has to be a way of determining when an aircraft can be handed over to the automated landing system. The controller shouldn't ask an aircraft for details of its position and check these, then ask for velocities and check these. Instead the controller should simply ask the aircraft "Are you ready to auto land?"; the aircraft can check all the constraints for itself.

class Aircraft

So we can expect class Aircraft. Class Aircraft will own a group of data members and provide functions like "Update accelerations, change to these new values" and "Check whether ready to auto-land", "Fly for another minute and work out position".

class Aircontroller?

A class representing the "air traffic controller" is not quite so obvious. For a start, there will only ever be one instance of this class around. It gets created at the start of the game and is used until the game ends.

When you've programmed many applications like this you find that usually it is helpful to have an object that owns most of the other program elements and provides the primary control functions.

In this example, the aircontroller will own the aircraft and will provide a main "run" function. It is in this "run" function that we simulate the passage of time. The "run" function will have a loop, each cycle of which represents one minute of elapsed time. The run-loop will organize things like letting each aircraft update its position, prompting the user to enter commands, updating some global timer, and checking for new aircraft arriving.

The main program can be simplified. It will create an "AirController" and tell it to "run".

The process of completing the design will be iterative. Each class will get considered and its details will be elaborated. The focus of attention may then switch to some other class, or maybe to a global function (a function that doesn't belong to any individual class). When other details have been resolved, a partial design for a class may get reassessed and then expanded with more detail.

class Aircontroller

What does an Aircontroller own?

Data owned

It is going to have to have some data members that represent the controlled airspace. These would probably include a count of the number of aircraft in the controlled space, and in some way will have to include the aircraft themselves. The aircraft are going to be class instances created in the heap, accessed by Aircraft* pointers. The Aircontroller object can have an array of pointers.

What does an Aircontroller do?

Services provided

It gets created by the main program and then runs. So it will need a constructor that initializes its records to show that the airspace is empty, and a "run" function that controls the simulation loop.

The main loop is going to involve the following steps:

```
Update a global timer
Let all planes move
Report current status, getting each plane to list its details
Get all planes to check whether they've crashed or can
    auto-land, remove those no longer present in the airspace
if can handle more planes (arrays not full)
    check if any arrived
if airspace is not empty
    give user the chance to enter commands
    check for quit command
```

Naturally, most of these steps are going to be handled by auxiliary private member functions of the Aircontroller class.

The timer will be a global integer variable. It is going to get used by the function that adds aircraft to the airspace. (In this program, the timer could be a local variable of the "run" function and get passed as an argument in the call to the function that adds aircraft. However, most simulations require some form of global variable that represents the current time; so we follow the more general style.)

The array of pointers to aircraft will start with all the pointers `NULL`. When an aircraft is added, a `NULL` pointer is changed to hold the address of the new aircraft. When an aircraft crashes or lands, it gets deleted and the pointer with its address is reset to `NULL`. At any particular moment in the simulation, the non-`NULL` entries will be scattered through the array. When aircraft need to be activated, a loop like the following can be used:

```
for i = 0; i < max; i++
    if aircraft[i] != NULL
           aircraft[i] do something
```

Similar loop constructs will be needed in several of the auxiliary private member functions of class Aircontroller.

For example, the function that lets all the planes move is going to be:

```
move planes
for i = 0; i < max; i++
    if aircraft[i] != NULL
           aircraft[i] move
```

while that which checks for transfers to the auto lander would be:

```
check landings
for i = 0; i < max; i++
    if aircraft[i] != NULL
           if aircraft[i] can transfer
                  report handoff to auto lander
                  delete the aircraft
                  aircraft[i] = NULL;
                  reduce count of aircraft in controlled space
```

Getting user commands

Another major role for the Aircontroller object will be to get commands from the user. The user may not want to enter any commands, or may wish to get one plane to change its accelerations, or may need to change several (or might even wish to quit from the game). It would be easiest to have a loop that kept prompting the user until some "ok no more commands" indicator was received. The following outline gives an idea for the structure:

```
Get user commands
    prompt for command entry
    loop
           read word
           if( word is "Quit")
                  arrange to terminate program
           if( word is "OK")
                  return

           read accelerations and time

           if(any i/o problems) warn, and just ignore data
           else handle command
```

Handling commands will involve first finding the aircraft with the call sign entered, then telling it of its new accelerations:

```
Handle command
    identify target aircraft (one with call sign entered)
    if target not found
           warn of invalid call sign
    else tell target to adjust accelerations
```

The target can be found in another function that loops asking each aircraft in turn whether it has the call sign entered.

class Aircraft

Data owned What does an individual aircraft own?

There is a whole group of related data elements – the call sign, the coordinates, velocities. When an aircraft is created, these are to be filled in with predefined data from a static array that specifies the planes. It would actually be convenient to define an extra struct whose role is just to group most or all of these data elements. The predefined data used to feed aircraft into the airspace could be represented as an array of these structs.

So, we could have:

```
struct PlaneData {
    double        x, y, z;      // coords
    double        vx, vy, vz;   // velocities
    …
    short         fuel;         // minutes remaining
    char          name[7];      // 6 character call name
};
```

An Aircraft object would have a `PlaneData` as a data member, and maybe some others.

What do Aircraft do? Aircraft are going to be created; when created they will be given information to fill in their `PlaneData` data member. They will get told to move, to print their details for reports, to change their accelerations, to check whether they can land, and to check whether they are about to suffer misfortune like flying into the ground. There were quite a number of "terminating conditions" listed in the problem description; each could be checked by a separate private member function.

Some of the member functions will be simple:

```
toofast
    return true if speed exceeds safe maximum

tooslow
    return true if speed exceeds safe minimum
```

The overall horizontal speed would have to be a data member, or an extra auxiliary private member function should be added to calculate the speed.

The function that checks for transfer to auto lander will also be fairly simple:

```
making final approach
    if(not in range 4…10 miles west of runway)
           return false
```

```
       if(too far north or south)
              return false
       if( speed out of range)
              return false
       …
       return true
```

The "move" and "update" functions are more complex. The update function has to check that the new accelerations are reasonable:

```
   update
       check time value (no negatives!)

       check each acceleration value against limits
              if any out of range ignore command

       calculate velocity components after one minute
              of new acceleration

       if any out range ignore command

       acknowledge acceptable command

       change data members
```

The move function has to recompute the x, y, z coordinates and the velocities. If the specified number of minutes associated with the last accelerations has elapsed, these need to be zeroed. The fuel left has to be reduced by an appropriate amount.

```
   move

       calculate distance travelled
              assuming constant velocity

       if(command time >0)
              /* Still accelerating */
              work out new velocities,
                     (note: limit +ve vertical velocity)
              correct distances to allow for accelerations

              allow for extra fuel burn if accelerating

              decrement command time
                     and if now zero clear those
                     accelerations

       fix up coordinates
       allow for normal fuel usage
```

Aircraft generating function

This would use a global array with "arrival times" of aircraft, another global array with aircraft details, and an integer counter identifying the number of array entries already used. A sketch for the code is:

```
new arrivals function
    if time < next arrival time
           return null

    create new aircraft initializing it with data
           from PlaneData array

    update count of array entries processed

    return aircraft
```

Refining the initial designs

The initial designs would then be refined. For the most part, this would involve further expansion of the member functions and the identification of additional auxiliary private member functions. The process used to refine the individual functions would be the same as that illustrated in the examples in Part III.

The design process lead to the following class definitions and function specifications:

```
struct PlaneData {
    double x,y,z;
    double vx,vy,vz;
    double ax,ay,az;
    short  fuel;
    char   name[7];
};

class Aircraft {
public:
    Aircraft(const PlaneData& d);
    void          PrintOn(ostream& os) const;
    void          Update(double a1,
                         double a2, double a3, short timer);
    void          Move();
    Boolean       MakingApproach() const;
    Boolean       Terminated() const;
    Boolean       CheckName(const char str[]) const;
    const char*   ID() const;
private:
    Boolean       Stalled() const;
    Boolean       Pancaked() const;
    Boolean       NoFuel() const;
    Boolean       TooFast() const;
    Boolean       TooSlow() const;
    Boolean       CrashDiving() const;
    double        Speed() const;

    PlaneData     fData;        // main data
    short         fTime;        // time remaining for command
};
```

(The implementation uses a typedef to define "Boolean" as a synonym for unsigned char and defines false and true.) The member functions would be further documented:

```
Aircraft(const PlaneData& d);
```

Initialize new Aircraft from PlaneData

```
void       PrintOn(ostream& os) const;
```

Output details as needed in report

```
void       Update(double a1,
                  double a2, double a3, short timer);
```

Verify acceleration arguments, update data members if appropriate, acknowledge.

```
void       Move();
```

Recompute position and velocity.

```
Boolean    MakingApproach() const;
```

Check whether satisfies landing conditions.

```
Boolean Terminated() const;
```

Check if violates any flight constraints.

```
Boolean CheckName(const char str[]) const;
```

Check str argument against name

```
const char*       ID() const;
```

Return name string (for reports like "XXX has landed/crashed")

```
Boolean    Stalled() const;
```

Check if flown too high.

```
Boolean    Pancaked() const;
```

Check if flown too low.

```
Boolean    NoFuel() const;
```

Check fuel.

```
Boolean    TooFast() const;
```
Check if flown too fast.

```
Boolean    TooSlow() const;
```
Check if flown too slow.

```
Boolean    CrashDiving() const;
```
Check if exceeding descent rate.

```
double     Speed() const;
```
Check ground speed

The AirControlller class is:

```
class AirController {
public:
    AirController();
    void          Run();
private:
    void          Report();
    void          MovePlanes();
    void          AddPlanes();
    void          Validate();
    void          CheckCrashes();
    void          CheckLandings();

    Boolean       GetCommands();
    void          HandleCommands(const char name[],
                         double,double,double,short);
    Aircraft      *IdentifyByCall(const char name[]);

    Aircraft      *fAircraft[kMAXPLANES];
    int                  fControlling;
};
```

The member functions are:

```
AirController();
```
Constructor, initialize air space to empty.

```
void       Run();
```
Run main loop moving aircraft, checking commands etc

```
void       Report();
```
Get reports from each aircraft.

```
void        MovePlanes();
```
Tell each aircraft in turn to move to new position.

```
void        AddPlanes();
```
Call the plane generating function, if get plane returned add it to array and notify player of arrival.

```
void        Validate();
```
Arrange checks for landings and crashes.

```
void        CheckCrashes();
```
Get each plane in turn to check if any constraints violated.

```
void        CheckLandings();
```
Get each plane in turn to check if it can autoland, sign off those that can autoland.

```
Boolean     GetCommands();
```
Get user commands, returns true if "Quit"

```
void        HandleCommands(const char name[],
                          double,double,double,short);
```
Pass a "change accelerations" command on to correct aircraft.

```
Aircraft    *IdentifyByCall(const char name[]);
```
Identify aircraft from call sign.

(Really, the program should check for aircraft collisions as well as landings and crash conditions. Checking for collisions is either too hard or too boring. The right way to do it is to calculate the trajectories (paths) flown by each plane in the past minute and determine if any trajectories intersect. This involves far too much mathematics. The boring way of doing the check is to model not minutes of elapsed times but seconds. Each plane moves for one second, then distances between each pair of planes is checked. This is easy, but clumsy and slow.)

Implementation

The following are an illustrative sample of the functions. The others are either similar, or trivial, or easy to code from outlines given earlier.

The `main()` function is concise:

```
int main()
{
    AirController me;
    me.Run();
    return 0;
}
```

This is actually a fairly typical `main()` for an object-based program – create the principal object and tell it to run.

The file with the application code will have the necessary #includes and then start with definition of constants representing limits:

```
// Constants that define Performance Limits
const double kMaxHeight                      = 40000.0;
const double kMinHeight                      = 450.0;

…
const double kMaxVDownAcceleration           = -400.0;

// Constants that define conditions for landing
const double     kminh                       =  600.0;
const double     kmaxh                       = 3000.0;

const double  kinnerrange                    = -4.0;
…
const short      kminfuel                    = 4;
```

There are a couple of globals, the timer and a counter that is needed by the function that adds planes:

```
static     long   PTimer = 0;
static short      PNum = 0;
```

The arrays with arrival times and prototype aircraft details would have to be defined:

```
short      PArrivals[] = {
    5, 19,  31, 45, 49,
    …
};

PlaneData  ExamplePlanes[] = {
    { -149.0,   12.0,  25000.0,
           7.0,  0.0, -400.0,
                  0.0, 0.0,  0.0,
                         120,  "BA009" },
    { -144.0,   40.0,  25000.0,
            4.8, -1.4,    0.0,
                  0.0, 0.0,  0.0,
                         127,  "QF040" },
    …

    };
```

```
static short NumExamples = sizeof(ExamplePlanes) /
                                       sizeof(PlaneData);
```

When appropriate, function `NewArrival()` creates an `Aircraft` on the heap using the `new` operator; a `PlaneData` struct is passed to the constructor. The new `Aircraft` is returned as a result. The value `NULL` is returned if it was not time for a new `Aircraft`.

```
Aircraft *NewArrival(void)
{

    if(PNum==NumExamples) return NULL;

    if(PTimer < PArrivals[PNum]) return NULL;
    Aircraft *newPlane = new Aircraft(ExamplePlanes[PNum]);
    PNum++;
    return newPlane;
}
```

The constructor for controller simply sets all elements of the array `fAircraft` to `NULL` and zeros the `fControlling` counter. The main `Run()` function is:

```
void AirController::Run()
{
    for(Boolean done = false; !done;) {
           PTimer++;
           MovePlanes();
           Report();
           Validate();
           if(fControlling < kMAXPLANES)
                  AddPlanes();
           if(fControlling>0)
                  done = GetCommands();
           }
}
```

The `Report()` and `MovePlanes()` functions are similar; their loops involve telling each aircraft to execute a member function (`PrintOn()` and `Move()` respectively). Note the way that the member function is invoked using the `->` operator (`fAircraft[i]` is a pointer to an `Aircraft`).

```
void AirController::Report()
{
    if(fControlling < 1) cout << "Airspace is empty\n";
    else
    for(int i=0;i<kMAXPLANES; i++)
           if(fAircraft[i]!= NULL)
                  fAircraft[i]->PrintOn(cout);
}
```

The `AddPlanes()` function starts with a call to the plane generator function `NewArrival()`; if the result from `NewArrival()` is `NULL`, there is no new `Aircraft` and `AddPlanes()` can exit.

```
void AirController::AddPlanes()
{
    Aircraft*     aPlane = NewArrival();
    if(aPlane == NULL)
           return;

    cout << "New Aircraft:\n\t";
    fControlling++;
    aPlane->PrintOn(cout);
    for(int i=0;i<kMAXPLANES; i++)
           if(fAircraft[i] == NULL) {
                  fAircraft[i] = aPlane;
                  aPlane = NULL;
                  break;
                  }
    if(aPlane != NULL) {
           cout << "??? Planes array full, program bug??"
                  << endl;
           delete aPlane;
           }
}
```

In other cases, the count of aircraft in the controlled space is increased, and an empty slot found in the `fAircraft` array of pointers. The empty slot is filled with the address of the newly created aircraft.

The `CheckCrashes()` and `CheckLanding()` functions called from `Validate()` are similar in structure. They both have loops that check each aircraft and dispose of those no longer required:

```
void AirController::CheckCrashes()
{
    for(int i=0;i<kMAXPLANES; i++)
           if((fAircraft[i] != NULL) &&
                  (fAircraft[i]->Terminated())) {
                         cout << fAircraft[i]->ID();
                         cout << " CRASHED!" << endl;
                         delete fAircraft[i];
                         fAircraft[i] = NULL;
                         fControlling--;
                         }

}
```

Input to `GetCommands()` will start with a string; this should be either of the key words `OK` or `Quit` or the name of a flight. A buffer of generous size is allocated to store this string temporarily while it is processed.

The function checks for, and acts on the two key word commands. If the input string doesn't match either of these it is assumed to be the name of a flight and the function therefore tries to read three doubles (the new accelerations) and an integer (the time). Checks on input errors are limited but they are sufficient for this kind of simple interactive program. If data are read successfully, they are passed to the `HandleCommand()` function.

```
Boolean AirController::GetCommands()
{
    char   buff[120];
    cout << "Enter Flight Commands:\n"; cout.flush();

    for(;;) {
           cin >> buff;
           if(0 == ::strcmp(buff,"Quit"))return true;
           if(0 == ::strcmp(buff,"OK")) {
                  cin.ignore(SHRT_MAX,'\n');
                  return false;
                  }
           double a1,a2,a3;
           short  t;
           cin >> a1 >> a2 >> a3 >> t;
           if(cin.fail()) {
                  cout << "Bad input data\n";
                  cin.clear();
                  cin.ignore(SHRT_MAX,'\n');
                  }
           else HandleCommands(buff,a1,a2,a3,t);
           }

}
```

The `HandleCommands()` function uses the auxiliary function `IdentifyByCall()` to find the target. If target gets set to `NULL` it means that flight identifier code was incorrectly entered. If the target is found, it is asked to update its accelerations:

```
void AirController::HandleCommands(const char* call, double a1,
double a2, double a3, short n)
{
    Aircraft*     target = IdentifyByCall(call);

    if(target == NULL) {
           cout << "There is no aircraft with " << call
                  << " as call sign.\n";
           return;
           }
    target->Update(a1,a2,a3,n);
}
```

Function `IdentifyByCall()` loops through the collection of aircraft asking each in turn whether its call sign matches the string entered by the user. The function returns a pointer to an `Aircraft` with a matching call sign, or `NULL` if none match.

```
Aircraft *AirController::IdentifyByCall(const char name[])
{
    for(int i=0; i < kMAXPLANES; i++)
           if((fAircraft[i] != NULL) &&
                  (fAircraft[i]->CheckName(name)))
                         return fAircraft[i];
    return NULL;
}
```

The constructor for `Aircraft` initializes its main data from the `PlaneData` struct passed as an argument and zeros the command timer. The `PrintOn()` function simply outputs the data members:

```
Aircraft::Aircraft(const PlaneData& d)
{
    fData = d;
    fTime = 0;
}

void Aircraft::PrintOn(ostream& os) const
{
    os.setf(ios::fixed,ios::floatfield);
    os.precision(2);
    os << fData.name;

    os << "\t(" << fData.x << "," << fData.y << ","
           << fData.z << ")\t(";
    …
    os << endl;
}
```

An aircraft's name is needed at a couple of points in the code of `AirController`. The `ID()` function returns the name as a `char*` ("pointer to array of characters").

Returning const something*

Now the address returned is that of the `fData.name` data member. If you return the address of a data member, the "wall around the data" can be breached. The data values can be changed directly instead of through member functions. Any pointers to data members that are returned should be specified as "pointers to constant data". So here the return type is `const char*` (i.e. a pointer to an array of characters that shouldn't be changed).

```
const char* Aircraft::ID() const
{
    return fData.name;
}
```

Function `Move()` sorts out the changes to position, velocity, and fuel:

```
void Aircraft::Move()
{
    double dx, dy, dz;
    dx = fData.vx;
    dy = fData.vy;
    dz = fData.vz;

    double oldspeed = Speed();

    if(fTime>0) {
           /* Still accelerating */
           fData.vx += fData.ax; dx += fData.ax * 0.5;
           fData.vy += fData.ay; dy += fData.ay * 0.5;
           fData.vz += fData.az; dz += fData.az * 0.5;

           if((fData.az>0) || (Speed() > oldspeed))
                  fData.fuel--;

           if(fData.vz>kMaxAscentRate) {
                  fData.vz = kMaxAscentRate;
                  fData.az = 0.0;
                  }

           fTime--;
           if(fTime==0)
                  fData.az = fData.ay = fData.ax = 0.0;
           }
    fData.x += dx;
    fData.y += dy;
    fData.z += dz;
    fData.fuel--;
}
```

The `Update()` function performs a series of checks on the new acceleration values. If all checks are passed, the values in the data members are changed and an acknowledgment is printed:

```
void Aircraft::Update(double a1, double a2,
           double a3, short timer)
{
    /* validate input */
    if(timer < 1) return;
/* accelerations, x and y limited by Horizontal acceleration */
    if(fabs(a1) > kMaxHAcceleration) return;
    if(fabs(a2) > kMaxHAcceleration) return;

    /* Vertical bracketed in range. */
    if(a3 > kMaxVUpAcceleration) return;
    if(a3 < kMaxVDownAcceleration) return;

    /* check that new velocities not excessive */
    double newvertvelocity = fData.vz + a3;
    if(newvertvelocity > kMaxAscentRate) return;
    if(newvertvelocity < kMaxDescentRate) return;

    double newvx, newvy, newtotal;
    newvx = fData.vx + a1;
    newvy = fData.vy + a2;
    newtotal = sqrt(newvx*newvx + newvy*newvy);
    if(newtotal > kMaxSpeed) return;
    if(newtotal < kMinSpeed) return;

    cout  << fData.name << ": Roger\n";
    fData.ax = a1;
    fData.ay = a2;
    fData.az = a3;
    fTime = timer;
}
```

Function `MakingApproach()` involves a series of checks against the constraints that define the "auto landing" conditions:

```
Boolean Aircraft::MakingApproach() const
{
    /*   in pick up range of auto lander? */

    if((fData.x < kouterrange) ||
                  (fData.x > kinnerrange)) return false;

    /*  Aligned with runway? */
    if(fabs(fData.y) > koffline) return false;

    /* In height bracket? */
    if((fData.z < kminh) ||
                  (fData.z > kmaxh)) return false;

    /* In velocity bracket? */
    if((fData.vx < kxprimelow) ||
                  (fData.vx > kxprimehigh)) return false;

    …

    /* and no real accelerations? */
    if(fabs(fData.ax) > kxalphalimit) return false;
    …

    /* and sufficient fuel? */
    if(fData.fuel < kminfuel) return false;

    return true;
}
```

Function `Terminated()` uses the various auxiliary member functions to check for terminating conditions:

```
Boolean Aircraft::Terminated() const
{
    return
           Stalled() || Pancaked() || NoFuel() ||
                  TooFast() || TooSlow() || CrashDiving();
}
```

The remaining functions of class `Aircraft` are all simple; representative examples are:

```
Boolean Aircraft::CheckName(const char str[]) const
{
    return (0 == strcmp(fData.name, str));
}

Boolean Aircraft::Stalled() const
{
    return fData.z > kMaxHeight;
}

double Aircraft::Speed() const
{
    return sqrt(fData.vx*fData.vx + fData.vy*fData.vy);
}
```

The program runs OK but as a game it is a little slow (and directing an aircraft to land safely is surprisingly hard). The following is a fragment from a recording of the game:

```
Airspace is empty
New Aircraft:
BA009  (-149,12,25000) (7,0,-400)  (0,0,0) fuel: 120
Enter Flight Commands:
BA009 is west of the runway, a little to the north and coming
in much too fast; slow it down, make it edge south a bit
BA009 -1 -0.2 0 3
```

```
BA009: Roger
OK
BA009  (-142.5,11.9,24600) (6,-0.2,-400)  (-1,-0.2,0)  fuel:
119
…
BA009   (-124.5,9.9,23000)      (4,-0.6,-400)   (0,0,0) fuel:
115
Enter Flight Commands:
Slow up a little more, increase rate of descent
BA009 -0.5 0 -100 2
BA009: Roger
OK
BA009 (-120.75,9.3,22550 (3.5,-0.6,-500) (-0.5,0,-100) fuel:
114
…
BA009  (-15.5,0,3350) (3,-0,-500)  (0,0,0) fuel: 78
QF040  (-52.77,0.18,11600) (3.01,-0.05,-600) (0,0,0) fuel: 100
NZ164  (-71,-45.95,20500) (1,5.3,-500)  (0,0,0) fuel: 70
CO1102 (116,-87,32500) (-5,1,-500) (0,0,0) fuel: 73
Enter Flight Commands:
BA009 0 0 150 1
BA009: Roger
OK
BA009 (-12.5,0,2925) (3,-0,-350) (0,0,0) fuel: 76
QF040 (-49.76,0.13,11000) (3.01,-0.05,-600)  (0,0,0) fuel: 99
NZ164   (-70,-40.65,20000) (1,5.3,-500) (0,0,0) fuel: 69
CO1102  (111,-86,32000) (-5,1,-500)  (0,0,0) fuel: 72
Enter Flight Commands:
OK
BA009  (-9.5,0,2575) (3,-0,-350) (0,0,0) fuel: 75
QF040  (-46.75,0.08,10400) (3.01,-0.05,-600) (0,0,0) fuel: 98
NZ164   (-69,-35.35,19500) (1,5.3,-500) (0,0,0) fuel: 68
CO1102  (106,-85,31500) (-5,1,-500) (0,0,0) fuel: 71
BA009 transferred to airport traffic control, bye!
One landed safely, QF040 on course as well
Enter Flight Commands:
```

20.4 & : THE "ADDRESS OF" OPERATOR

The primary reason for having pointers is to hold addresses of data objects that have been dynamically allocated in the heap, i.e. for values returned by the new operator.

You need pointers to heap based structures when representing simple data objects with variable lifetimes like the `Aircraft` of the last example. You also need pointers when building up complex data structures that represent networks of different kinds. Networks are used for all sorts of purposes; they can represent electrical circuits, road maps, the structure of a program, kinship relations in families, the structure of a molecule, and thousands of other things. Networks are built up at run-time by using pointers to thread together different dynamically created component parts.

Why would you want pointers to data in the static data segment or to automatics on the stack? After all, if such variables are "in scope" you can use them directly, you don't have to work through pointer intermediaries. You should NEVER incorporate the address of an automatic (stack based) data item in an elaborate network structure. It is rare to want to incorporate the address of a static data item.

Really, you shouldn't be writing code that needs addresses of things other than heap-based objects. Usually you just want the address of an entire heap based object (the value returned by `new`), though sometimes you may need the address of a specific data member within a heap based object.

But C and C++ have the `&` "address of" operator. This allows you to get the address of any data element. Once an address has been obtained it can be used as the value to be stored in a pointer of the appropriate type. If you look at almost any C program, and most C++ programs, you will see `&` being used all over the place to get addresses of automatics and statics. These addresses are assigned to pointers. Pointers are passed as arguments. Functions with pointer arguments have code that has numerous expression using pointer dereferencing (`*ptr`). Why are all these addresses needed?

Reference arguments and pointer arguments for functions

No "pass by reference" in C

For the most part, the addresses are needed because the C language does not support pass by reference. In C, arguments for functions are passed by value. So, simple variables and struct instances are copied onto the stack (ignore arrays for now, they are discussed in the next section). The called function works with a copy of the original data. There are sound reasons for designing a language that way. If you are trying to model the mathematical concept of a function, it is something that simply computes a value. An idealized, mathematical style function should not have side effects; it shouldn't go changing the values of its arguments.

Uses of "pass by reference"

Function arguments that are "passed by reference" have already been used for a number of reasons. Thus, in the example in section 12.8.3, there was a function that had to return a set of values (actually, an int and a double) rather than a single value. Since a function can only return a single value, it was necessary to have reference arguments. The example function had an integer reference and a double reference as arguments; these allowed the function to change the values of the caller's variables.

Other examples have used pass by reference for structures. In the case of const reference arguments, this was done to avoid unnecessary copying of the struct that would occur if it were passed by value. In other cases, the structs are passed by reference to allow the called function to modify the caller's data.

Implementation of pass by value and pass by reference

As previously explained, a compiler generates different code for value and reference arguments. When a function needs a value argument, the compiler generates code that, at the point of call, copies the value onto the stack; within the function, the code using the "value" variable will have addresses that identify it by its position in the current stack frame. Reference arguments are treated differently. At the point of call their addresses are loaded onto the stack. Within the function, the address of a reference argument is taken from the stack and loaded into an address register. When the code needs to access the actual data variable, it uses indirect addressing via this address register; which is the same way that code uses pointers.

Consider the `Exchange()` function in the following little program:

```
void Exchange(double& data1, double& data2)
{
    double temp;
    temp = data1;
    data1 = data2;
    data2 = temp;
}

int main()
{
    cout << "Enter two numbers : ";
    double a, b;
    cin >> a >> b;
    if(b < a)
           Exchange(a, b);
    cout << "In ascending order: " << a << ", " << b << endl;
    return 0;
}
```

Function `Exchange()` swaps the values in the two variables it is given as arguments; since they are reference variables, the function changes data in the calling environment. The code generated for this program passes the addresses of variables `a` and `b`. The value in `a` gets copied into `temp`, replaced by the value in `b`, then the value of `temp` is stored in `b`. These data movements are achieved using indirect addressing through address registers loaded with the address of `a` and `b`.

Compiler uses addresses and pointers for pass by reference

Pass by reference really involves working with addresses and pointers. But the compiler takes care of the details. The compiler arranges to get the addresses of the arguments at the point of call. The compiler generates code that uses indirection, pointer style, for the body of the function.

Programmer uses addresses and pointers

Programs need the "pass by reference" mechanism. The C language didn't have it. But C allowed programmers to hand code the mechanism everywhere it was needed.

C has pass by value. An address is a value and can be passed as an argument if the function specifies that it wants a pointer as an argument. It is easy to use the `&` operator to get an address, so at the point of call the addresses can be determined and their values put onto the stack.

Faking pass by reference using explicit addresses and pointers

The code of the function has to use pointers. So, for example, a function that needs to change a `double` argument will need a `double*` pointer (i.e. its argument list will have to include something like `double *dptr`). When the code of the function needs the value of that `double`, it will have to access it indirectly via the pointer, i.e. it will have to use the value `*dptr`.

Using explicit pointers and address, the previous program becomes:

```
void Exchange(double *dptr1, double *dptr2)
{
    double temp;
    temp = *dptr1;
    *dptr1 = *dptr2;
    *dptr2 = temp;
}

int main()
{
    cout << "Enter two numbers : ";
    double a, b;
    cin >> a >> b;
    if(b < a)
           Exchange(&a, &b);
    cout << "In ascending order: " << a << ", " << b << endl;
    return 0;
}
```

The function prototype specified pointer arguments, `Exchange(double *dptr1, double *dptr2)`. The call, `Exchange(&a, &b)`, has the appropriate "address of operations" getting the addresses of the actual arguments `a` and `b`.

The statement:

```
temp = *dptr1;
```

gets the double value accessed indirectly via `dptr1` and saves it in the double `temp`. The statement

```
    *dptr1 = *dptr2;
```

gets the value accessed indirectly via `dptr2` and uses it to overwrite the double in the memory location referenced via `dptr1`. The final statement changes the value in the location reference by `dptr2` to be the value stored temporarily in `temp`.

Explicit pointers and addresses represent the underlying mechanism

High level source code using explicit pointers and addresses corresponds almost directly to the actual machine instruction sequences generated by the compiler. Identical instruction sequences are generated for the high level code that uses references; it is just that the code with references leaves the mechanism implicit. There are programmers who prefer the pointer style, arguing that "it is better to see what is really going on". (Of course, if you follow such arguments to their logical conclusion, all programs should be written in assembly language because in assembler you can see the instructions and really truly know what is going on.)

Pointer style or reference style?

There are a few situations where the pointer style is more natural. The low level `read()` and `write()` functions are good examples. These need to be given the address of a block of bytes; the code of the function uses a byte pointer to access the first byte, then the second, and so on as each byte gets transferred to/from file.

In most other cases, reference style is easier to read. The reference style leaves the mechanisms implicit, allowing the reader to focus on the data transformations being performed. The use of explicit pointers, and the continual pointer dereferencing that this necessitates, makes it harder to read the code of functions written using the pointer style.

While you can adopt the style you prefer for the functions that you write for yourself, you are often constrained to using a particular style. All the old C libraries still used from C++ will use pointers and will need to be passed addresses. Because of the C heritage, many C++ programmers write code with pointer arguments; consequently, many of the newer C++ libraries use pointers and pointer dereferencing.

Getting the addresses of program elements

You can get the address of any program element. You automatically get the address of any data object you create in the heap (the return value from `new` is the address); in most other cases you need to use the `&` operator. The following contrived code fragment illustrates how addresses of a variety of different data elements can be obtained and used in calls to `write()`. The prototype for function `write()` specifies either a `char*` or a `void*` (depending on the particular version of the iostream library used in your IDE); so at the point of call, a value representing an address has to be specified.

```
struct demo { int f1; double f2; };

// Some data in the static data segment
long array[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
demo d1 = { 1, 2.2};
demo d2 = { 2, 4.4};

void WriteJunk(ofstream& out)
{
    double auto1 = 3.142;
    demo *demoptrA = new demo;
    demoptrA->f1 = -7; demoptrA->f2 = 6.8;
    demo *demoptrB = new demo;
    demoptrB->f1 = 11; demoptrB->f2 = 81.8;
    …
    // The version of iostream and the compiler on your IDE
    // may require an explicit (char*) cast in all these calls
    // e.g. the first might have to be
    //     out.write((char*) &auto1, sizeof(double));
    out.write(&auto1, sizeof(double));
    out.write(&(array[4]), sizeof(long));
    out.write(&d1, sizeof(demo));
```

```
        out.write(&(d2.f1), sizeof(int));
        out.write(demoptrA, sizeof(demo)); // No & needed!
        out.write(&(demoptrB->f2), sizeof(double));
    }
```

As shown in this fragment, you can get the addresses of:

- local, automatic (stack-based) variables (`&auto1`);
- specifically chosen array elements (automatic or global/filescope) (`&(array[4])`);
- global/filescope variables (`&d1`);
- a chosen data member of an auto or static struct (`&(d2.f1)`).
- a chosen data member of a struct accessed by a pointer (`&(demoptrB->f2)`).

The call

```
        out.write(demoptrA, sizeof(demo));
```

doesn't need an `&` "get address of"; the pointer `demoptrA` contains the address needed, so it is the value from `demoptrA` has to be passed as the argument.

Addresses of functions and arrays

There are two kinds of program element whose address you can get by just using their names. These are functions and arrays. You shouldn't use the `&` address of operator with these.

The use of function addresses, held in variables of type "pointer to function", is beyond the scope of this book. You will learn about these later.

As explained more in the next section, C (and hence C++) regards an array name as having an address value; in effect, an array name is a pointer to the start of the array. So, if you needed to pass an array to the `write()` function, your code would be:

```
    long Array2[] = { 100, -100, 200, 300, 567 };
    …
    out.write(Array2, sizeof(Array2));
```

Pointers as results from functions

A function that has a pointer return type will return an address value. This address value will get used after exit from the function. The address is going to be that of some data element; this data element had better still be in existence when the address gets used!

This should be obvious. Surprisingly often, beginners write functions that return addresses of things that will disappear at the time the functions exits. A simple example is:

buggy code!

```
char *GetName()
{
    char buff[100];
    cout << "Enter name";
    cin.getline(buff, 99, '\n');
    return buff;
}
```

This doesn't work. The array `buff` is an automatic, it occupies space on the stack while the function `GetName()` is running but ceases to exist when the function returns.

The compiler will let such code through, but it causes problems at run time. Later attempts to use the pointer will result in access to something on the stack, but that something won't be the character buffer.

A function should only return the address of an object that outlives it; which means an object created in the heap. The following code will work:

```
char *GetName()
{
    char buff[100];
    cout << "Enter name";
    cin.getline(buff, 99, '\n');
    char   *ptr = new char[strlen(buff) + 1];
    strcpy(ptr, buff);
    return ptr
}
```

20.5 POINTERS AND ARRAYS

When arrays were introduced in Chapter 11, it was noted that C's model for an array is relatively weak. This model, which of course is shared by C++, really regards an array as little more than a contiguous block of memory.

The instruction sequences generated for code that uses an array have the following typical form:

```
load address register with the address of the start of the
array
calculate offset for required element
add offset to address register
load data value using indirect address via address register
```

which is somewhat similar to the code for using a pointer to get at a simple data element

```
load address register from pointer variable
load data value using indirect address via address register
```

and really very similar to code using a pointer to access a data member of a struct:

```
load address register with the address of the start of the struct
calculate offset for required data member
add offset to address register
load data value using indirect addressing via address register
```

If you think about things largely in terms of the instruction sequences generated, you will tend to regard arrays as similar to pointer based structs. Hence you get the idea of the name of the array being a pointer to the start of that array.

Once you start thinking mainly about instructions sequences, you do tend to focus on different issues. For example, suppose that you have an array of characters that you need to process in a loop, you know that if you use the following high level code:

```
char msg[50];
…
for(int i=0; i < len; i++) {
    …
    … = msg[i];
    …
    }
```

then the instruction sequence for accessing the ith character will be something like

```
load an address register with the address of msg[0]
add contents of integer register that holds i to address register
load character indirectly using address in address register
```

which is a little involved. (Particularly as on the very first machine used to implement C, a PDP-7, there was only one register so doing subscripting involved a lot of shuffling of data between the CPU and memory.)

Still thinking about instructions, you would know that the following would be a more efficient way of representing the entire loop construct:

```
    load address register with address of msg[0]
    load integer register with 0
loop
    compare contents of integer register and value len
    jump if greater to end_loop
    …
    load character indirectly using address in address register
    add 1 to address register // ready for next time
    …
```

This instruction sequence, which implements a pointer based style for accessing the array elements, is more "efficient". This approach needs at most two instructions to get a character from the array where the other scheme required at least three instructions.

(On many machines, the two operations "load character indirectly using address in address register" and "add 1 to address register" can be combined into a single instruction.)

C was meant to compile to efficient instruction sequences. So language constructs were adapted to make it possible to write C source code that would be easy to compile into the more efficient instruction sequence:

```
char msg[50];
char *mptr;
…
mptr = msg;
for(int i=0; i < len; i++) {
    …
    … = *mptr;
    mptr++;
    …
    }
```

This initializes a pointer with the address of the first array element:

```
mptr = msg;
```

(which could also be written as `mptr = &(msg[0])`). When the character is needed, pointer dereferencing is used:

```
    … = *mptr;
```

Finally, "pointer arithmetic" is performed so as to make the pointer hold the address of the next character from the array.

Pointer arithmetic

C (and C++) have to allow arithmetic operations to be done on pointers. Once you've allowed operations like ++, you might as well allow other addition and subtraction operations (I don't think anyone has ever found much application for pointer multiplication or division). For example, if you didn't have the `strlen()` function, you could use the following:

```
int StrLen(char *mptr)
{
    char *tmp;
    tmp = mptr;
    while( *tmp) tmp++;
    return tmp - mptr;
}
```

This function would be called with a character array as an argument (e.g. `StrLen("Hello");`). The `while` loop moves the pointer `tmp` through the array until it is pointing to a null character. The final statement:

```
    return tmp - mptr;
```

subtracts the address of the start of the array from the address where the '\0' character is located.

Arithmetic operations on pointers became a core part of the language. Many of the C libraries, still used from C++, are written in the expectation that you will be performing pointer arithmetic. For example, the string library contains many functions in addition to `strlen()`, `strcmp()`, and `strcpy()`; one of the other functions is `strchr()`:

```
char *strchr(const char *s, int c)
```

this finds the first occurrence of a character `c` in a string `s`. It returns a pointer to the position where the character occurs (or `NULL` if it isn't there). Usually, you would want to know which array element of the character array contained the character, but instead of an integer index you get a pointer. Of course, you can use pointer arithmetic to get the index:

```
char word[] = "Hello";
char *ptr = strchr(word, 'e');
int pos = ptr - word;
cout << "e occurred at position " << pos << endl;
```

Obviously, character arrays can not be a special case. What works for characters has to work for other data types. This requirement has ramifications. If `++` changes a `char*` pointer so that it refers to the next element of a character array, what should ++ do for an array of doubles?

The following program fragment illustrates the working of `++` with different data types (all pointer values are printed as long integers using decimal output to make things clearer):

```
struct zdemo { int f1; char f2; double f3 ;};

int main()
{
    char cArray[10];
    short sArray[10];
    long lArray[10];
    double dArray[10];
    zdemo zArray[10];

    char *cptr = cArray;
    short *sptr = sArray;
    long *lptr = lArray;
    double *dptr = dArray;
    zdemo *zptr = zArray;
    for(int i=0; i < 5; i++) {
           cout << long(cptr) << ", " << long(sptr) << ", "
                  << long(lptr) << ", " << long(dptr) << ", " <<
                                 long(zptr) << endl;
           cptr++; sptr++, lptr++, dptr++, zptr++;
    }
    cout << cptr - cArray << ", ";
    cout << sptr - sArray << ", ";
    cout << lptr - lArray << ", ";
    cout << dptr - dArray << ", ";
    cout << zptr - zArray << endl;

    return 0;
}
```

Output

```
22465448, 22465460, 22465480, 22465520, 22465600
22465449, 22465462, 22465484, 22465528, 22465616
22465450, 22465464, 22465488, 22465536, 22465632
22465451, 22465466, 22465492, 22465544, 22465648
22465452, 22465468, 22465496, 22465552, 22465664
5, 5, 5, 5, 5
```

The arithmetic operations take account of the size of the data type to which the pointer refers. Increment a `char*` changes it by 1 (a char occupies one byte), incrementing a `double*` changes it by 8 (on the machine used, a double needs 8 bytes). Similarly the value `zptr (22465680) - zArray (22465600)` is 5 not 80, because this operation on `zdemo` pointers involves things whose unit size is 16 bytes not one byte.

A lot of C code uses pointer style for all operations on arrays. While there can be advantages in special cases like working through the successive characters in a string, in most cases there isn't much benefit. Generally, the pointer style leads to code that is much less easy to read (try writing a matrix multiply function that avoids the use of the `[]` operator).

Although "pointer style" with pointer dereferencing and pointer arithmetic is a well established style for C programs, it is not a style that you should adopt. You have arrays because you want to capture the concept of an indexable collection of data elements. You lost the benefit of this "indexable collection" abstraction as soon as you start writing code that works with pointers and pointer arithmetic.

20.6 BUILDING NETWORKS

The Air Traffic Controller program represented an extreme case with respect to dynamically created objects – its `Aircraft` were all completely independent, they didn't relate in a specific ways.

There are other programs where the structures created in the heap are components of a larger construct. Once created, the individual separate structures are linked together using pointers. This usage is probably the more typical. Real examples appear in the next chapter with standard data structures like "lists" and "trees".

In future years, you may get to build full scale "Macintosh" or "Windows" applications. In these you will have networks of collaborating objects with an "application" object linked (through pointers) to "document" objects; the documents will have links to "views" (which will have links back to their associated documents), and other links to "windows" that are used to frame the views.

The data manipulated by such programs are also likely to be represented as a network of components joined via pointers. For example, a word processor program will have a data object that has a list of paragraphs, tables, and pictures. Each paragraph will have links to things such as structures that represent fonts, as well as to lists of sentences.

Each of the components in one of these complex structures will be an instance of some struct type or a class. These various structs and classes will have data members that are of pointer types. The overall structure is built up by placing addresses in the pointer data members.

Figure 20.7 illustrates stages in building up a simple kind of list or queue. The overall structure is intended to keep a collection of structures representing individual data items. These structures would be called "list cells" or "list nodes". They would be instances of a struct or class with a form something like the following:

```
struct list_cell {
    data_type1    data_member_1;
    data_type2    data_member_2;
    …;
    list_cell     *fNext;
};
```

Pointer data member for link

Most of the data members would be application specific and are not of interest here. But one data member would be a pointer; its type would be "pointer to list cell". It is these pointer data members, the `fNext` in the example, that are used to link the parts together to form the overall structure.

Head pointer

A program using a list will need a pointer to the place where it starts. This will also be a pointer to `list_cell` It would be a static data segment variable (global or filescope) or might be a data member of yet another more complex structure. For the example in Figure 20.7, it is assumed that this "head pointer" is a global:

```
list_cell  *Head;
```

Initially, there would be no `list_cells` and the head pointer would be `NULL` (stage 1 in Figure 20.7). User input would somehow cause the program to create a new `list_cell`:

```
list_cell *MakeCell()
{
```

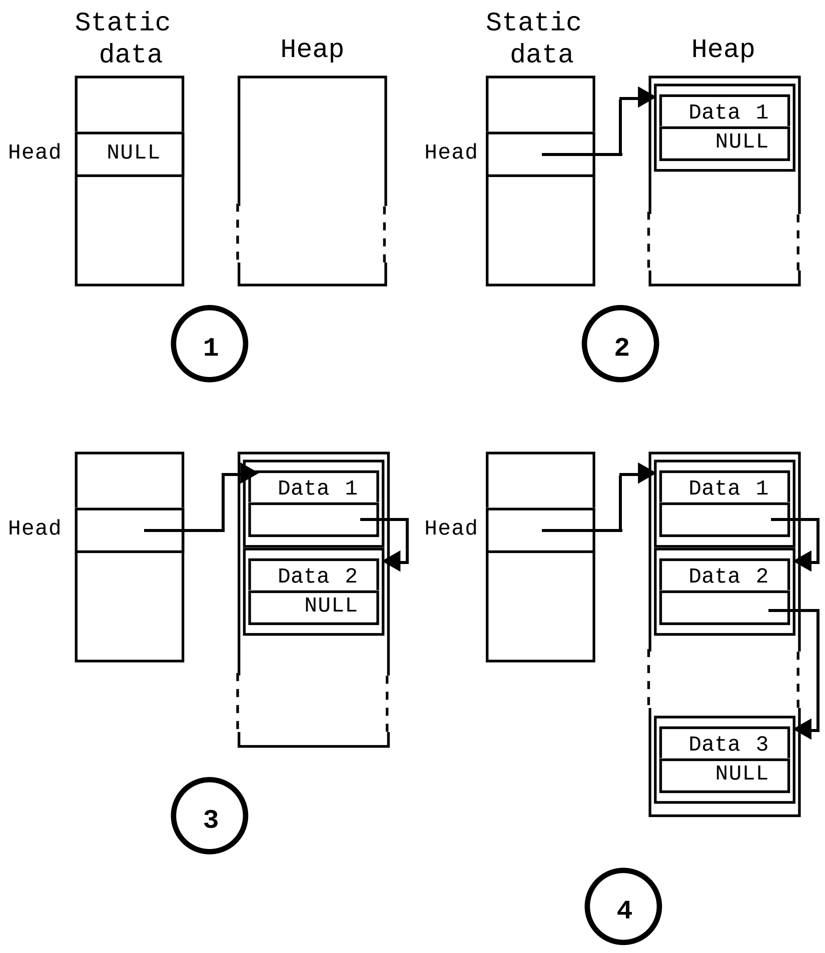

Figure 20.7 Building a "list structure" by threading together "list nodes" in the heap.

```
        list_cell *res = new list_cell;
        cout << "Enter data for …";
        …;
        list_cell->data_member_1 = something;
        …
        list_cell->fNext = NULL;
}
```

The `MakeCell()` function would obviously fill in all the various specialized data members; the only general one is the link data member `fNext` which has to be initialized to `NULL`.

The function that created the new `list_cell` would add it to the overall structure:

Setting the Head pointer to the first cell

```
…
list_cell *newdata = MakeCell();
if(Head == NULL)
    Head = newdata;
else …;
…
```

As this would be the first `list_cell`, the `Head` pointer is changed to point to it. This results in the situation shown as stage 2, in Figure 20.7.

When, subsequently, other `list_cells` get created, the situation is slightly different. The `Head` pointer already points to a `list_cell`. Any new `list_cell` has to be attached at the end of the chain of existing `list_cell(s)` whose start is identified by the `Head` pointer.

Finding the end of a list

The correct place to attach the new `list_cell` will be found using a loop like the following:

```
list_cell  *ptr = Head;
while(ptr->fNext != NULL)
    ptr = ptr->fNext;
```

Attaching a new cell at the end of the list

The `list_cell*` variable `ptr` starts pointing to the first `list_cell`. The while loop moves it from `list_cell` to `list_cell` until a `list_cell` with a `NULL fNext` link is found. Such a `list_cell` represents the end of the current list and is the place where the next `list_cell` should be attached. The new `list_cell` is attached by changing the `fNext` link of that end `list_cell`:

```
ptr->fNext = newdata;
```

Stages 3 and 4 shown in Figure 20.7 illustrate the overall structure after the addition of the second and third `list_cells`.

A lot of the code that you will be writing over the next few years will involve building up networks, and chasing along chains of pointers. Two of the examples in the next chapter look at standard cases of simple pointer based composite structures. The first of these is a simple `List` class that represents a slight generalization of the idea of the list as just presented. The other example illustrates a "binary search tree". The code for these examples provides a more detailed view of how such structures are manipulated.