19

19 Beginners' Class

Chapter 17 illustrated a few applications of structs. The first was a reworking of an earlier example, from Section 13.2, where data on pupils and their marks had to be sorted. In the original example, the names and marks for pupils were in separate arrays. Logically, a pupil's name and mark should be kept together; the example in Section 17.1 showed how this could be done using a struct. The marks data in these structs were only examined by the sort function, and about the only other thing that happened to the structs was that they got copied in assignment statements. In this example, the structs were indeed simply things that kept together related data.

More than just a data collection

In all the other examples from Chapter 17, the structs were used by many functions. Thus the `Points` could be combined with an `AddPoint()` function, while the `Customer` records in 17.3 were updated, transferred to and from disk, and printed. Although they used arrays rather than structs, the examples in Chapter 18 also had data structures that had many associated functions. Thus, the different forms of "hash table" all had functions for initialization, searching for a key, inserting a key, along with associated support functions (like the check for a "null" hash table entry). Similarly, the "bit maps" for the information retrieval system in Section 18.3 had a number of associated functions that could be used to do things like set specific bits.

Data and associated functions

You would have difficulties if you were asked "Show me how this code represents a 'Customer' (or a 'Hash Table', or a 'Bit map')". The program's representation of these concepts includes the code manipulating the structures as well as the structures themselves.

This information is scattered throughout the program code. There is nothing to group the functions that manipulate `Customer` records. In fact, any function can manipulate `Customer` records. There may have been a `PrintDetails(Customer&)` function for displaying the contents of a `Customer` record but there was nothing to stop individual data members of a `Customer` record being printed from `main()`.

When programs are at most a couple of hundred lines, like the little examples in Chapters 17 and 18, it doesn't much matter whether the data structures and associated functions are well defined. Such programs are so small that they are easy to "hack out" using any ad hoc approach that comes to mind.

As you get on to larger programs, it becomes more and more essential to provide well defined groupings of data and associated functionality. This is particularly important if several people must collaborate to build a program. It isn't practical to have several people trying to develop the same piece of code, the code has to be split up so that each programmer gets specific parts to implement.

Functional abstraction?

You *may* be able to split a program into separately implementable parts by considering the top level of a top-down functional decomposition. But usually, such parts are far from independent. They will need to share many different data structures. Often lower levels of the decomposition process identify requirements for similar routines for processing these structures; this can lead to duplication of code or to inconsistencies. In practice, the individuals implementing different "top level functions" wouldn't be able to go off and proceed independently.

or "data abstraction"

An alternative decomposition that focussed on the data might work better. Consider the information retrieval example (Section 18.3), you could give someone the task of building a "bit map component". This person would agree to provide a component that transferred bit maps between memory and disk, checked equality of bit maps, counted common bits in two maps, and performed any other necessary functions related to bit maps. Another component could have been a "text record component". This could have been responsible for displaying its contents, working with files, and (by curtesy of the bit map component) building a bit map that encoded the key words that were present in that text record.

The individuals developing these two components could work substantially independently. Some parts of a component might not be testable until all other components are complete and brought together but, provided that the implementors start by agreeing to the component interfaces, it is possible to proceed with the detailed design and coding of each component in isolation.

Data abstraction: its more than just a header file!

The code given in Chapter 18 had an interface (header) file that had a typedef for the "bit map" data and function prototypes for the routines that did things like set specific bits. A separate code file provided the implementations of those routines. As a coding style, this is definitely helpful; but it doesn't go far enough. After all, a header file basically describes the form of a data structure and informs other programmers of some useful functions, already implemented, that can be used to manipulate such structures. There is nothing to enforce controls; nothing to make programmers use the suggested interface.

Building a wall around data

Of course, if implementors are to work on separate parts of a program, they must stick to the agreed interfaces. Consider the example of programmers, A, B, C, and D who were working on a geometry modelling system. Programmer A was responsible for the coordinate component; this involved points (`struct pt { double x, y; };`) and functions like `offset_pt(double dx, double dy)`. Programmer B was uncooperative and did not want to wait for A to complete the points component. Instead of using calls to `offset_pt()` B wrote code that manipulated the data fields directly (`pt p; … p.x += d1;`). B then left the project and was replaced by D.

Programmer C needed polar coordinates and negotiated with A for functions `radius(const pt p)` and `theta(const pt p)`. After implementing these functions, A and C did a timing analysis of their code and found that these new functions were being called very often and represented a measurable percentage of the run time.

Programmer A, who after all was officially responsible for the definition of points and their behaviour, updated things so that the struct contained polar as well as cartesian coordinates (`struct pt { double x, y, r, theta; };`). A changed the new `radius()` and `theta()` functions so that these simply returned the values of the extra data members and then changed all the functions like `offset_pt()` so that these updated the polar coordinates as well as the cartesian coordinates.

Programmers C and D duly recompiled their parts to reflect the fact that the definition of `struct pt` had changed with the addition of the extra data members.

Programmer A was fired because the code didn't work. C and D wasted more than a week on A's code before they found the incorrect, direct usage of the struct in the code originally written by B.

The problem here was that A couldn't define a "wall" around the data of a point structure. The header file had to describe the structure as well as the prototypes of A's routines. Since the structure was known, uncooperative programmers like B could access it.

Things would have been different if A could have specified things so that point structures could only be used via A's own routines. (In particular, A would still have a job, while B would probably have quit before the project really started.)

The need for components

Modern programs typically require many "components". These components consist of the declaration of new type of data object and of the functions that are to be used to manipulate instances of this new type. You do require "walls around the data". When you define a new data type, you should be able to specify how it is to be used. If you can define components like this, then you can expect to be able to put a program together by combining separately developed components that work in accord with specified interfaces.

Language support for reliable data types

It is possible use the older C language to implement components. Possible. But it is not always easy and it does depend on the discipline of the programmers. C doesn't enforce compliance with an interface. (One could argue that C doesn't enforce anything!) In contrast, the C++ language has extensive support for the programmer who needs to define and work with new data types.

Classes

The new language feature of C++ that supports a component oriented style of programming is "class". A "class" is meant to describe a conceptual entity; that is describe something that owns data and provides services related to its data.

A class declaration defines a new data type. The declaration does specify the form of a data structure and in this respect it is like a struct declaration. Once the compiler has seen the class declaration, it permits the definition of variables of this new type; these variables are normally referred to as "*instances of a class*" or "*objects*".

In addition to describing the data members, a class declaration allows a programmer to specify how objects that are instances of the class can be used. This is done by

identifying those functions that are allowed to access and change data members in an object. These functions are mainly "*member functions*". Member functions form part of the definition of the new data type.

The code for member functions is not normally included in the class declaration. Usually, a class declaration is in a header file; the code implementing the member functions will be in a separate code file (or files). However, at the point where they are defined, member functions clearly identify the class of which they are a part. Consequently, although the description of a class may be spread over a header file and several code files, it is actually quite easy to gather it all together in response to a request like "Show me how this code represents a 'Customer'."

Classes and design

Classes facilitate program design

The major benefit of classes is that they encourage and support a "component oriented" view of program development. Program development is all about repeatedly reanalyzing a problem, breaking it down into smaller more manageable pieces that can be thought about independently. The decomposition process is repeated until you know you have identified parts that are simple enough to be coded, or which may already exist. We have seen one version of this overall approach with the "top down functional decomposition" designs presented in Part III. Components (and the classes that describe them) give you an additional way of breaking up a problem.

Finding components

In fact, for large programs you should start by trying to identify the "components" needed in a program. You try to partition the problem into components that each own some parts of the overall data and perform all the operations needed on the data that they own. So you get "bit map" components, "hash table" components, "word" components and "text record" components. You characterize how the program works in terms of interactions among these components – e.g. "if the `hash_table` identifies a `vocab_item` that matches the `word`, then get the bit number from that `vocab_item` and ask the `bit_map` to set that bit". You have to identify all uses of a component. This allows you to determine what data that component owns and what functions it performs. Given this information, you can define the C++ class that corresponds to the identified component.

Designing the classes

Once you have completed an initial decomposition of the programming problem by identifying the required classes, you have to complete a detailed design for each one. But each class is essentially a separate problem. Instead of one big complex programming problem, you now have lots little isolated subproblems. Once the data and member functions of a class have been identified, most of the work on the design of a class relates to the design and implementation of its individual member functions.

Designing the member functions

When you have reached the level of designing an individual member function of a class you are actually back in the same sort of situation as you were with the examples in Parts II and III. You have a tiny program to write. If a member function is simple, just assignments, loops, and selection statements, it is going to be like the examples in

Part II where you had to code a single `main()`. If a member function has a more complex task to perform, you may have to identify additional auxiliary functions that do part of its work. So, the techniques of top down functional decomposition come into play. You consider coding the member function as a problem similar to those in Part III. You break the member function down into extra functions (each of which becomes an extra member function of the class) until you have simplified things to the point where you can implement directly using basic programming constructs.

Reusing components

Breaking a problem into separately analyzable component parts gives you a way of handling harder problems. But there are additional benefits. Once you start looking for component parts, you tend to find that your new program requires components that are similar to or identical to components that you built for other programs. Often you can simply reuse the previously developed components.

This is a bit like reusing algorithms by getting functions from a function library, such as those described in Chapter 13. However, the scale of reuse is greater. If you get a `sort()` function from a function library, you are reusing one function. If you get a ready built bitmap class, you get an integrated set of functions along with a model for a basic data structure that you require in your program. Increasingly, software developers are relying on "class libraries" that contain classes that define ready made versions of many commonly required components.

Issues relating to the decomposition of problems into manageable parts, and reuse, exist as an underlying leitmotif or theme behind all the rest of the materials in the part of the text.

Topics in this chapter

Section 19.1 introduces C++ classes. It covers the declaration of a simple class and the definition of its member functions. Classes can be complex. Many aspects are deferred to later chapters (and some aspects aren't covered in this text at all).

The next two sections present simple examples. Class Bitmap, section 19.2, is a more complete representation of the concept of a bit map data object similar to that introduced in the information retrieval example in section 19.3. Class Number, section 19.3, represents an integer. You might think that with shorts, longs, (and long longs) there are already enough integers. Those represented as instances of class Number do however have some rather special properties.

19.1 CLASS DECLARATIONS AND DEFINITIONS

19.1.1 Form of a class declaration

A class declaration introduces the name of a new data type, identifies the data members present in each variable of this type (i.e. each "class instance" or "object"), identifies the

member functions that may be used to manipulate such variables, and describes the access (security) controls that apply. A declaration has the form:

```
class Name  {
    details …
    more details …
};
```

The declaration starts with the keyword `class`. The name of the class is then given. The usual naming rules apply, the name starts with a letter and contains letters, digits and underscore characters. (Sometimes, there may be additional naming conventions. Thus you may encounter an environment where you are expected to start the name of a class with either 'T' or 'C'. Such conventions are not followed here.)

The body of the declaration, with all the details, comes between `{` and `}` brackets. The declaration ends with a semicolon. Compilers get really upset, and respond with incomprehensible error messages, if you omit that final semicolon.

The details have to include a list of the associated functions and data members. Initially the examples will be restricted slightly, the associated functions will all be "member functions", that is they all are functions that are inherently part of the definition of the concept represented by the class. (There are other ways of associating functions with classes, they get touched on later in Section 23.2.) These lists of member functions and data members also specify the access controls.

Keywords 'public' and 'private'

There are only two kinds of access that need be considered at this stage. They are defined by the keywords `public` and `private`. A `public` member function (or data member) is one that can be used anywhere in the code of the program. A `private` data member (or member function) is one that can only be used within the code of the functions identified in the class declaration itself. Note the wording of the explanation. Generally, data members are all `private`, there may also be a few `private` member functions. The main member functions are all `public` (sometimes, but very rarely, there may be `public` data members).

You can specify the appropriate access control for each data member and member function individually, or you can have groups of `public` members interspersed with groups of `private` members. But most often, the following style is used:

```
class Name  {
public:
    details of the "public interface" of the class
private:
    private implementation details
    and description of data members of each instance
    of the class…
};
```

In older text books, you may find the order reversed with the `private` section defined first. It is better to have the `public` interface first. Someone who wants to know how

to use objects of this class need only read the `public` interface and can stop reading at the keyword `private`.

The following examples are just to make things a little more concrete (they are simplified, refinements and extensions will be introduced later). First, there is a class defining the concept of a point in two dimensional space (such as programmer A, from the earlier example, might have wished to define):

```
class Point {
public:
    …
    // Get cartesian coords
    double X();
    double Y();
    // Get polar coords
    double Radius();
    double Theta();
    // Test functions
    int ZeroPoint();
    int InFirstQuad();
    …
    // Modify
    void Offset(double deltaX, double deltaY);
    …
    void SetX(double newXval);
    …
    // Comparisons
    int Equal(const Point& other);
    …
private:
    void FixUpPolarCoords();
    …
    double fX, fY, fR, fTheta;
};
```

(The ellipses, "…", indicate places where additional member functions would appear in actual class declaration; e.g. there would be several other test functions, hence the ellipsis after function `InFirstQuad()`. The ellipsis after the `public` keyword marks the place where some special initialization functions would normally appear; these "constructor" functions are described in section 19.1.4 below.)

The public role of Points

The public interface specifies what other programmers can do with variables that are instances of this type (class). Points can be asked where they are, either in cartesian or polar coordinate form. Points can be asked whether they are at the coordinate origin or whether they are in a specified quadrant (functions like `ZeroPoint()` and `InFirstQuad()`). Points can be asked to modify their data members using functions like `Offset()`. They may also be asked to compare themselves with other points through functions like `Equal()`.

The private lives of Points

The private section of the class declaration specifies things that are concern of the programmer who implemented the class and of no one else. The data members

generally appear here. In this case there are the duplicated coordinate details – both cartesian and polar versions. For the class to work correctly, these data values must be kept consistent at all times.

There could be several functions that change the cartesian coordinate values (`Offset()`, `SetX()` etc). Every time such a change is made, the polar coordinates must also be updated. During the detailed design of the class, the code that adjusts the polar coordinates would be abstracted out of the functions like `SetX()`. The code becomes the body for the distinct member function `FixUpPolarCoords()`.

This function is private. It is an implementation detail. It should only be called from within those member functions of the class that change the values of the cartesian coordinates. Other programmers using points should never call this function; as far as they are concerned, a point is something that always has consistent values for its polar and cartesian coordinates.

For a second example, how about class `Customer` (based loosely on the example problem in Section 17.3):

```
class Customer {
public:
    …
    // Disk transfers
    int    ReadFrom(ifstream& input);
    int    WriteTo(ofstream& output);
    // Display
    void   PrintDetails(ofstream& out);
    // Query functions
    int    Debtor();
    …
    // Changes
    void   GetCustomerDetails();
    void   GetOrder();
    …
private:
    UT_Word       fName;
    …
    UT_Word       fOrders[kMAXITEMS];
    int           fNumOrder;
    double        fAmountOwing;
    Date          fLastOrder;
};
```

Class `Customer` should define all the code that manipulates customer records. The program in Section 17.3 had to i) transfer customer records to/from disk (hence the `WriteTo()` and `ReadFrom()` functions, ii) display customer details (`PrintDetails()`), iii) check details of customers (e.g. when listing the records of all who owed money, hence the `Debtor()` function), and had to get customer details updated. A `Customer` object should be responsible for updating its own details, hence the member functions like `GetOrder()`.

The data members in this example include arrays and a struct (in the example in Section 17.3, `Date` was defined as a struct with three integer fields) as well as simple data types like int and double. This is normal; class `Point` is atypical in having all its data members being variables of built in types.

Minor point on naming conventions

There is no requirement that the data members have names starting with 'f'. It is simply a useful convention that makes code easier to read and understand. A name starting with 'g' signifies a global variable, a name with 's' is a "file scope" static variable, 'e' implies an enumerator, 'c' or 'k' is a constant, and 'f' is a data member of a class or struct. There are conventions for naming functions but these are less frequently adhered to. Functions that ask for yes/no responses from an object (e.g. `Debtor()` or `InFirstQuad()`) are sometimes given names that end with '_p' (e.g `Debtor_p()`); the 'p' stands for "predicate" (dictionary: 'predicate' assert or affirm as true or existent). `ReadFrom()` and `WriteTo()` are commonly used as the names of the functions that transfer objects between memory and disk. Procedures (void functions) that perform actions may all have names that involve or start with "Do" (e.g. `DoPrintDetails()`). You will eventually convince yourself of the value of such naming conventions; just try maintaining some code where no conventions applied.

19.1.2 Defining the member functions

A class declaration promises that the named functions will be defined somewhere. Actually, you don't have to define all the functions, you only have to define those that get used. This allows you to develop and test a class incrementally. You will have specified the class interface, with all its member functions listed, before you start coding; but you can begin testing the code as soon as you have a reasonable subset of the functions defined.

Usually, class declarations go in header files, functions in code files. If the class represents a single major component of the program, e.g. class `Customer`, you will probably have a header file Customer.h and an implementation file Customer.cp. Something less important, like class `Point`, would probably be declared in a header file ("geom.h") along with several related classes (e.g. `Point3D`, `Rectangle`, `Arc`, …) and the definitions of the member functions of all these classes would be in a corresponding geom.cp file.

A member function definition has the general form:

```
return type
class_name::function_name(arguments)
{
    body of function
}
```

e.g.

```
void Customer::PrintDetails(ofstream& out)
{
    …
}

double
Point::Radius()
{
    …
}
```

(Some programmers like the layout style where the return type is on a separate line so that the class name comes at the beginning of the line.)

Scope qualifier operator

The double colon :: is the "scope qualifier" operator. Although there are other uses, the main use of this operator is to associate a class name with a function name when a member function is being defined. The definitions of all functions of class `Point` will appear in the form `Point::`*`function_name`*, making them easy to identify even if they occur in a file along with other definitions like `Point3D::Radius()` or `Arc::ArcAngle()`).

Some examples of function definitions are:

```
double Point::Radius()
{
    return fR;
}

void Point::SetX(double newXval)
{
    fX = newXval;
    FixUpPolarCoords();
}

void Point::Offset(double deltaX, double deltaY)
{
    fX += deltaX;
    fY += deltaY;
    FixUpPolarCoords();
}
```

Now you may find something slightly odd about these definitions. The code is simple enough. `Offset()` just changes the `fX` and `fY` fields of the point and then calls the private member function `FixUpPolarCoords()` to make the `fR`, and `fTheta` members consistent.

The oddity is that it isn't clear which `Point` object is being manipulated. After all, a program will have hundreds of `Points` as individual variables or elements of arrays of `Points`. Each one of these points has its own individual `fX` and `fY` data members.

The code even looks as if it ought to be incorrect, something that a compiler should stomp on. The names `fX` and `fY` are names of data members of something that is like a

struct, yet they are being used on their own while in all previous examples we have only used member names as parts of fully qualified names (e.g. `thePt.fX`).

If you had been using the constructs illustrated in Part III, you could have had a struct `Pt` and function `OffsetPt()` and `FixPolars()`:

```
struct Pt { double x, y, r, t; };

void OffsetPt(Pt& thePoint, double dx, double dy)
{
    thePoint.x += dx;
    thePoint.y += dy;
    FixPolars(thePoint);
}

void FixPolars(Pt& thePt)
{
    …
}
```

The functions that change the `Pt` struct have a `Pt&` (reference to `Pt`) argument; the value of this argument determines which `Pt` struct gets changed.

This is obviously essential. These functions change structs (or class instances). The functions have to have an argument identifying the one that is to be changed. However, no such argument is apparent in the class member function definitions.

Implicit argument identifying the object being manipulated

The class member functions do have an *implicit* extra argument that identifies the object being manipulated. The compiler adds the address of the object to the list of arguments defined in the function prototype. The compiler also modifies all references to data members so that they become fully qualified names.

A "pointer parameter" identifies the object

Rather than a reference parameter like that in the functions `OffsetPt()` and `FixPolars()` just shown, this extra parameter is a "pointer". Pointers are covered in the next chapter. They are basically just addresses (as are reference parameters). However, the syntax of code using pointers is slightly different. In particular a new operator, ->, is introduced. This "pointer operator" is used when accessing a data member of a structure identified by a pointer variable. A few examples appear in the following code. More detailed explanations are given in the next chapter.

You can imagine the compiler as systematically changing the function prototypes and definitions. So, for example:

```
void Point::SetX(double newXval)
{
    fX = newXval;
    FixUpPolarCoords();
}
```

gets converted to

```
void __Point__SetX_ptsd(Point* this, double newXval)
{
    this->fX = newXval;
    this->FixUpPolarCoords();
}
```

while

```
int Customer::Debtor()
{
    return (fAmountOwing > 0.0);
}
```

gets converted to

```
int __Customer__Debtor_ptsv(Customer* this)
{
    return (this->fAmountOwing > 0.0);
}
```

The `this` pointer argument will contain the address of the data, so the compiler can generate code that modifies the appropriate `Point` or checks the appropriate `Customer`.

Normally, you leave it to the compiler to put in "`this->`" in front of every reference to a data member or member function; but you can write the code with the "`this->`" already present. Sometimes, the code is easier to understand if the pointer is explicitly present.

19.1.3 Using class instances

Once a class declaration has been read by the compiler (usually as the result of processing a #include directive for a header file with the declaration), variables of that class type can be defined:

```
#include "Customer.h"
    …

void RunTheShop()
{
    Customer theCustomer;
    …
}
```

or

```
#include "geom.h"
```

```
void DrawLine(Point& p1, Point& p2)
{
    Point   MidPoint;
    …
}
```

Just like structs, variables of class types can be defined, get passed to functions by value or by reference, get returned as a result of a function, and be copied in assignment statements.

Most code that deals with structs consists of statements that access and manipulate individual data members, using qualified names built up with the "." operator. Thus in previous examples we have had things like:

```
cout << theVocabItem.fWord;
```

```
strcpy(theTable[ndx].fWord, aWord);
```

With variables of class types you can't go around happily accessing and changing their data; the data are walled off! You have to ask a class instance to change its data or tell you about the current data values:

```
Point       thePoint;
…
thePoint.SetX(5.0);
…
if(thePoint.X() < 0.0) …
```

These requests, calls to member functions, use much the same syntax as the code that accessed data members of structures. A call uses the name of the object qualified by the name of the function (and its argument list).

You can guess how the compiler deals with things. You already know that the compiler has built a function:

```
void __Point__SetX_ptsd(Point* this, double newXval);
```

from the original `void Point::SetX(double)`. A call like

```
thePoint.SetX(5.0);
```

gets converted into:

```
__Point__SetX_ptsd(&thePoint, 5.0);
```

(The `&` "get the address of" operator is again used here to obtain the address of the point that is to be changed. This address is then passed as the value of the implicit `Point* this` parameter.)

19.1.4 Initialization and "constructor" functions

Variables should be initialized. Previous examples have shown initialization of simple variables, arrays of simple variables, structs, and arrays of structs. Class instances also need to be initialized.

For classes like class `Point` and class `Customer`, initialization would mean setting values in data fields. Later, Chapter 23, we meet more complex "resource manager" classes. Instances of these classes can be responsible for much more than a small collection of data; there are classes whose instances own opened files, or that control Internet connections to programs running on other computers, or which manage large collections of other data objects either in memory or on disk. Initialization for instances of these more complex structures may involve performing actions in addition to setting a few values in data members.

constructors

So as to accommodate these more complex requirements, C++ classes define special initialization functions. These special functions are called "constructors" because they are invoked as an instance of the class is created (or constructed).

There are some special rules for contructors. The name of the function is the same as the name of the class. A constructor function does not have any return type.

We could have:

```
class Point {
public:
    Point(double initalX = 0.0, double initialY = 0.0);
    …
};

class Customer {
public:
    Customer();
    …
};
```

The constructor for class `Point` has two arguments, these are the initial values for the X, Y coordinates. Default values have been specified for these in the declaration. The constructor for class `Customer` takes no arguments.

These functions have to have definitions:

```
Point::Point(double initialX, double initialY)
{
    fX = initialX;
    fY = initialY;
    FixUpPolarCoords();
}
```

```
Customer::Customer()
{
    strcpy(fName, "New customer");
    strcpy(fAddress, "Delivery address");
    …
    fAmountOwing = 0.0;
}
```

Here, the constructors do involve calls to functions. The constructor for class `Point` initializes the `fX`, `fY` data members but must also make the polar coordinates consistent; that can be accomplished by a call to the member function `FixUpPolar-Coords()`. The constructor for `Customer` makes calls to `strcpy` from the string library.

Overloaded constructors

A class can have an overloaded constructor; that is there can be more than one constructor function, with the different versions having different argument lists. The class `Number`, presented in Section 19.3, illustrates the use of multiple constructors.

Constructors can get quite elaborate and can acquire many specialized forms. Rather than introduce these now in artificial examples, these more advanced features will be illustrated later in contexts where the need arises.

When you need to define intialized instances of a class, you specify the arguments for the constructor function in the definition of the variable. For example:

```
Point      p1(17.0, 0.5);
Point      p2(6.4);
Point      p3;
```

Variable `p1` is a `Point` initially located at 17.0, 0.5; `p2` is located at 6.4, 0.0 (using the default 0.0 value for `initialY`). `Point p3` is located at 0.0, 0.0 (using the defaults for both `initialX` and `initialY`).

19.1.5 const member functions

Some member functions change objects, others don't. For example:

```
double Point::Y()
{
    return fY;
}
```

doesn't change the `Point` for which this function is invoked; whereas the `Point` is changed by:

```
void Point::SetY(double newYval)
{
    fY = newYval;
    FixUpPolarCoords();
```

```
}
```

It is worthwhile making this clear in the class declaration:

```
class Point {
public:
    …
    // Get cartesian coords
    double X() const;
    double Y() const;
    // Get polar coords
    double Radius() const;
    double Theta() const;
    // Test functions
    int ZeroPoint() const;
    int InFirstQuad() const;
    …
    // Modify
    void Offset(double deltaX, double deltaY);
    void SetX(double newXval);
    …
    // Comparisons
    int Equal(const Point& other) const;
```

Functions that don't change the object should be qualified by the keyword `const` both in the declaration, and again in their definitions:

```
double Point::Radius() const
{
    return fR;
}
```

const class instances

Apart from acting as an additional documentation cue that helps users understand a class, `const` functions are necessary if the program requires `const` instances of a class.

Now constancy isn't something you often want. After all, a `const Customer` is one who never places an order, never sets an address, never does anything much that is useful. But there are classes where `const` instances are meaningful. You can imagine uses for `const Points`:

```
const Point MinWindowSize(100.0, 100.0);
```

But there is a catch. The compiler is supposed to enforce constancy. So what is the compiler to do with code like:

```
// I need a bigger window
MinWindowSize.SetX(150.0);
// Test against window limits
if(xv < MinWindowSize.X()) …
```

The second is a legitimate use of the variable `MinWindowSize`, the first changes it. The first should be disallowed, the second is permitted.

The compiler can't know the rules unless you specify them. After all, the code for class `Point` may have been compiled years ago, all the compiler has to go on is the description in the header file. The compiler has to assume the worst. If you don't tell it that a member function leaves an object unchanged, the compiler must assume that it is changed. So, by default, a compiler refuses to let you use class member functions to manipulate `const` instances that class.

If your class declaration identifies some members as `const` functions, the compiler will let you use these with `const` instances.

19.1.6 inline member functions

Because the data in a class instance are protected, access functions have to be provided, e.g.:

```
double Point::X() const
{
    return fX;
}
```

Code needing the x-coordinate of a point must call this access function:

```
Point p1;
…
while(p1.X() > kMIN) { …; p1.Offset(delta, 0.0); }
```

Calling a function simply to peek at a data member of an object can be a bit costly. In practice, it won't matter much except in cases where the access function is invoked in some deeply nested inner loop where every microsecond counts.

Now inline functions were invented in C++ to deal with situations where you want function semantics but you don't want to pay for function overheads. Inlines can be used to solve this particular version of the problem.

As always, the definition of an inline function has to have been read so that the compiler can expand a function call into actual code. Since classes are going to be declared in header files that get #included in all the code files that use class instances, the inline functions definitions must be in the header file.

The following is the preferred style:

```
#ifndef __MYGEOM__
#define __MYGEOM__

class Point {
public:
```

```
    …
    double X();
    …
private:
    …
    double fX, fY, fR, fTheta;
};

other class declarations if any, e.g. class Arc { };

inline double Point::X() { return fX; }

#endif
```

Any inline functions get defined at the end of the file.

An alternative style is:

```
#ifndef __MYGEOM__
#define __MYGEOM__

class Point {
public:
    …
    double X() { return this->fX; }
    …
private:
    …
    double fX, fY, fR, fTheta;
};

other class declarations if any, e.g. class Arc { };
#endif
```

Here, the body of the inline function appears in the function declaration (note, an explicit `this->` qualifier on the data members is highly advisable when this style is used). The disadvantage of this style is that it makes the class declaration harder to read. Remember, your class declaration is going to be read by your colleagues. They want to know what your class can do, they don't really care how the work gets done. If you put the function definitions in the class declaration itself, you force them to look at the gory details. In contrast, your colleagues can simply ignore any inlines defined at the end of a header file, they know that these aren't really their concern.

19.2 EXAMPLE: BITMAPS (SETS)

The information retrieval example, Section 18.3, illustrated one simple use of a bitmap in an actual application The bitmaps in that example used '1/0' bits to indicate whether a document contained a specific keyword.

Really, the example illustrated a kind of "set". The bitmap for a document represented the "set of keywords" in that document. The system had a finite number of keywords, so there was a defined maximum set size. Each possible keyword was associated with a specific bit position in the set.

Such sets turn up quite often. So a "bit map" component is something that you might expect to "reuse" in many applications.

This section illustrates the design and implementation of a class that represents such bitmaps and provides the functions necessary to implement them. The implementation is checked out with a small test program.

Designing a general component for reuse is actually quite hard. Normally, you have a specific application in mind and the functionality that you identify is consequently biased to the needs of that application. When you later attempt to reuse the class, you find some changes and extensions are necessary. It has been suggested that a reusable class is typically created, and then reworked completely a couple of times before it becomes a really reusable component. So, the class developed here may not exactly match the needs of your next application, but it should be a good starting point.

Designing a simple class

This example serves as an introduction to the design of a class. Remember a class is a C++ description of a particular type of component that will be used in programs. Objects that are instances of the class will own some of the program's data and will provide all the services related to the owned data. The place to start designing a class is deciding what data its instances own and what services they provide.

The data owned

What data does an individual bitmap own? Each instance of class `Bitmap` will have an array of unsigned long integers. There shouldn't be any other data members. To make things simple, we can have the `Bitmap` class use a fixed size array, with the size defined by a #define constant.

Services provided by a bitmap

The next step is to decide what "services" a bitmap object should provide. You make up a list of useful functions. The list doesn't have to be complete, you can add further functionality later. Some useful services of a bitmap would be:

- Zeroing; reset all the bits to zero (the constructor function should call this, bit maps should start with all their bits set to 0).

- Set bit i; and Clear bit i;
  Need functions to set (make '1') and clear (make '0') a specified bit. This could be done by having separate set() and clear() functions, or a set_as() function that had two arguments – the bit number and the setting. Actually, it would probably be convenient if the class interface offered both forms because sometimes one form is more convenient than the other.

- Test bit i;

Will often want to test the status of an individual bit.

- Flip bit i;
  Change setting of a specified bit, if it was '1' it becomes '0' and vice versa.

- ReadFrom and WriteTo
  Will want to transfer the data as a block of binary information.

- PrintOn
  Produce a (semi)-readable printout of a bitmap as a series of hex values.

- Count
  Get Bitmap to report how many bits it has set to '1'.

- Invert
  Not clear whether this function would be that useful, but it might. The invert function would flip all the bits; all the '1's become '0's, all the '0's become '1's (the bitmap applies a "Not" operation to itself).

The next group of functions combine two bitmaps to produce a third as a result. Having a class instance as a function's result is just the same as having a struct as a result. As explained for structs, you must always consider whether this is wise. It can involve the use of lots of space on the stack and lots of copying operations. In this case it seems reasonable. Bitmap objects aren't that large and the function semantics are sensible.

- "And" with bit map "other"
  Returns the bitmap that represents the And of the bitmap with a second bitmap.

- "Inclusive Or" with bitmap "other".

- "Exclusive Or" with bitmap "other".

and for consistency

- "Not"
  Returns a bit map that is the inverse of this bitmap.

Finally, we should probably include an "Equals" function that checks equality with a second bitmap.

The discussion of design based on top down functional decomposition (Chapter 15) favoured textual rather than diagrammatic representation. Diagrams tend to be more useful in programs that use classes. Simple diagrams like that shown in Figure 19.1 can provide a concise summary of a class.

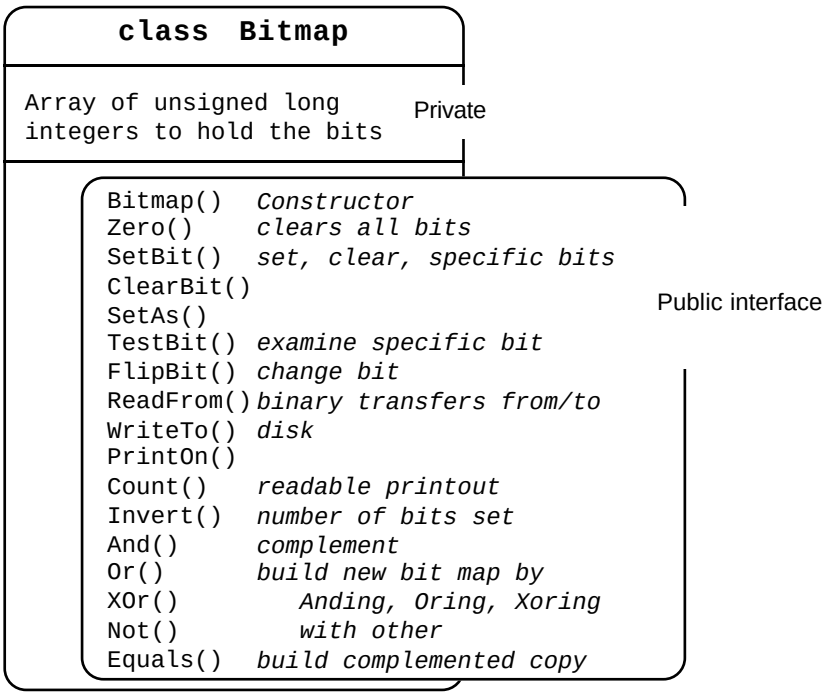

Figure 19.1 A simple preliminary design diagram for a class.

The figure shows the class with details of its private data and functions all enclosed within the class boundary. The public member functions stick outside the boundary. In this example, all the member functions are public.

The first iteration through the design of a class is complete when you have composed a diagram like Figure 19.1 that summarizes the resources owned by class instances and their responsibilities.

The next stage in class design involves firming up the function prototypes, deciding on the arguments, return types, and const status. (There may not be much use for const Bitmaps, but you should still make the distinction between access functions that merely look at the data of a class member and modifying procedures that change data.)

In general, it would also be necessary to produce pseudo-code outlines for each function. This isn't done here as the functions are either trivial or are similar to those explained in detail in Section 18.3.

The prototypes become:

```
            Bitmap();
void        Zero(void);
void        SetBit(int bitnum);
void        ClearBit(int bitnum);
void        SetAs(int bitnum, int setting);
```

```
int         TestBit(int bitnum) const;
void        FlipBit(int bitnum);
void        ReadFrom(fstream& in);
void        WriteTo(fstream& out) const;
void        PrintOn(ostream& printer) const;
int         Count(void) const;
void        Invert(void);

Bitmap      And(const Bitmap& other) const;
Bitmap      Or(const Bitmap& other) const;
Bitmap      XOr(const Bitmap& other) const;

Bitmap      Not(void) const;

int         Equals(const Bitmap& other) const;
```

About the only point to note are the `const Bitmap&` arguments in functions like `And()`. We don't want to have `Bitmaps` passed by value (too much copying of data) so pass by reference is appropriate. The "And" operation does not affect the two bitmaps it combines, so the argument is `const`. The i/o functions take `fstream` reference arguments.

Only 0 and 1 values are appropriate for argument setting in function `SetAs()`. It would be possible to define an enumerated type for this, but that seems overkill. The coding can use 0 to mean 0 (reasonable) and non-zero to mean 1.

Implementation

Header file for class declaration

The Bitmap class will be declared in a header file bitmap.h:

```
#ifndef __MYBITSCLASS__
#define __MYBITSCLASS__

// Code use iostream and fstream,
// #include these if necessary, fstream.h
// does a #include on iostream.h
#ifndef __FSTREAM_H
#include <fstream.h>
#endif

// Code assumes 32-bit unsigned long integers
#define MAXBITS 512
#define NUMWORDS 16

typedef unsigned long Bits;

class Bitmap {
public:
    Bitmap();
```

```
    void   Zero(void);
    void   SetBit(int bitnum);
    void   ClearBit(int bitnum);
    void   SetAs(int bitnum, int setting);
    int    TestBit(int bitnum) const;
    void   FlipBit(int bitnum);
    void   ReadFrom(fstream& in);
    void   WriteTo(fstream& out) const;
    void   PrintOn(ostream& printer) const;
    int    Count(void) const;
    void   Invert(void);

    Bitmap And(const Bitmap& other) const;
    Bitmap Or(const Bitmap& other) const;
    Bitmap XOr(const Bitmap& other) const;

    Bitmap Not(void) const;

    int    Equals(const Bitmap& other) const;
private:
    Bits   fBits[NUMWORDS];
};

#endif
```

This header file has a #include on fstream.h. Attempts to compile a file including bitmap.h without fstream.h will fail when the compiler reaches the i/o functions. Since there is this dependency, fstream.h is #included (note the use of conditional compilation directives, if fstream.h has already been included, it isn't read a second time). There is no need to #include iostream.h; the fstream.h header already checks this.

Function definitions in the separate code file

The functions implementing the Bitmap concept are defined in bitmap.c:

```
#include "bitmap.h"

Bitmap::Bitmap()
{
    Zero();
}

void Bitmap::Zero(void)
{
    for(int i=0;i<NUMWORDS; i++)
           fBits[i] = 0;
}
```

The constructor, `Bitmap::Bitmap()`, can use the `Zero()` member function to initialize a bitmap. Function `Zero()` just needs to loop zeroing out each array element.

```
void Bitmap::SetBit(int bitnum)
```

```
{
    if((bitnum < 0) || (bitnum >= MAXBITS))
           return;
    int word = bitnum / 32;
    int pos = bitnum % 32;

    Bits mask = 1 << pos;
    fBits[word] |= mask;
}
```

Function `SetBit()` uses the mechanism explained in 18.3. Function `ClearBit()` has to remove a particular bit. It identifies the array element with the bit. Then, it sets up a mask with the specified bit set. Next it complements the mask so that every bit except the specified bit is set. Finally, this mask is "Anded" with the array element; this preserves all bits except the one that was to be cleared.

```
void Bitmap::ClearBit(int bitnum)
{
    if((bitnum < 0) || (bitnum >= MAXBITS))
           return;
    int word = bitnum / 32;
    int pos = bitnum % 32;

    Bits mask = 1 << pos;
    mask = ~mask;
    fBits[word] &= mask;
}
```

Function `SetAs()` can rely on the implementation of `ClearBit()` and `SetBit()`:

```
void Bitmap::SetAs(int bitnum, int setting)
{
    if(setting == 0)
           ClearBit(bitnum);
    else SetBit(bitnum);
}
```

Function `TestBit()` uses the same mechanism for identifying the array element and bit and creating a mask with the chosen bit set. This mask is "Anded" with the array element. If the result is non-zero, the chosen bit must be set.

```
int Bitmap::TestBit(int bitnum) const
{
    if((bitnum < 0) || (bitnum >= MAXBITS))
           return 0;
    int word = bitnum / 32;
    int pos = bitnum % 32;

    Bits mask = 1 << pos;
```

```
    return (fBits[word] & mask);
}
```

Function `FlipBit()` uses an exclusive or operation to change the appropriate bit in the correct word (check that you understand how the xor operation achieves the desired result):

```
void Bitmap::FlipBit(int bitnum)
{
    if((bitnum < 0) || (bitnum >= MAXBITS))
           return;
    int word = bitnum / 32;
    int pos = bitnum % 32;

    Bits mask = 1 << pos;
    fBits[word] ^= mask;
}
```

Functions `ReadFrom()` and `WriteTo()` perform the binary transfers that copy the entire between file and memory:

```
void Bitmap::ReadFrom(fstream& in)
{
    in.read(&fBits, sizeof(fBits));
}

void Bitmap::WriteTo(fstream& out) const
{
    out.write(&fBits, sizeof(fBits));
}
```

These functions do not check for transfer errors. That can be done in the calling environment. When coding class `Bitmap`, you don't know what should be done if a transfer fails. (Your IDE's version of the iostream library, and its compiler, may differ slightly; the calls to read and write may need "`(char*)`" inserted before `&fBits`.)

The files produced using `WriteTo()` are unreadable by humans because they contain the "raw" binary data. Function `PrintOn()` produces a readable output:

```
void Bitmap::PrintOn(ostream& printer) const
{
    long savedformat = printer.flags();
    printer.setf(ios::showbase);
    printer.setf(ios::hex,ios::basefield);

    for(int i = 0; i <NUMWORDS; i++) {
           printer.width(12);
           printer << fBits[i];
           if((i % 4) == 3) cout << endl;
```

```
    }
    printer.flags(savedformat);
}
```

Function `PrintOn()` has to change the output stream so that numbers are printed in hex. It would be somewhat rude to leave the output stream in the changed state! So `PrintOn()` first uses the `flags()` member function of `ostream` to get the current format information. Before returning, `PrintOn()` uses another overloaded version of the `flags()` function to set the format controls back to their original state.

```
int Bitmap::Count(void) const
{
    int count = 0;
    for(int n=0; n < NUMWORDS; n++) {
           Bits x = fBits[n];
           int j = 1;
           for(int i=0;i<32;i++) {
                  if(x & j)
                         count++;
                  j = j << 1;
                  }
           }
    return count;
}
```

The `Count()` function uses a double loop, the outer loop steps through the array elements, the inner loop checks bits in the current element. It might be worth changing the code so that the inner loop was represented as a separate `CountBitsInElement()` function. This would be a private member function of class `Bitmap`.

Functions `Invert()`, `Not()` and `Equals()` all have similar loops that check, change, or copy and change successive array elements from `fBits`:

```
void Bitmap::Invert(void)
{
    for(int i=0; i < NUMWORDS; i++)
           fBits[i] = ~fBits[i];
}

Bitmap Bitmap::   Not(void) const
{
    Bitmap b;
    for(int i = 0; i < NUMWORDS; i++)
           b.fBits[i] = ~this->fBits[i];
    return b;

}

int Bitmap::Equals(const Bitmap& other) const
{
```

```
    for(int I = 0; I < NUMWORDS; i++)
           if(this->fBits[I] != other.fBits[i]) return 0;
    return 1;

}
```

Note the explicit use of the `this->` qualifier in `Not()` and `Equals()`. These functions manipulate more than one `Bitmap`, and hence more than one `fBits` array. There are the `fBits` data member of the object executing the function, and that of some second object. The `this->` qualifier isn't needed but sometimes it makes things clearer.

Accessing data members of another object of the same class

You will also note that the `Bitmap` object that is performing the `Equals()` operation is looking at the `fBits` data member of the second `Bitmap` (`other`). What about the "walls" around `other's` data?

Classes basically define families of objects and there are no secrets within families. An object executing code of a member function of the class is permitted to look in the private data areas of any other object of that class.

The functions `And()`, `Or()`, and `XOr()` are very similar in coding. The only difference is the operator used to combine array elements:

```
Bitmap Bitmap::And(const Bitmap& other) const
{
    Bitmap b;
    for(int i = 0; i < NUMWORDS; i++)
           b.fBits[i] = this->fBits[i] & other.fBits[i];
    return b;
}

Bitmap Bitmap::Or(const Bitmap& other) const
{
    Bitmap b;
    for(int i = 0; i < NUMWORDS; i++)
           b.fBits[i] = this->fBits[i] | other.fBits[i];
    return b;
}

Bitmap Bitmap::XOr(const Bitmap& other) const
{
    Bitmap b;
    for(int i = 0; i < NUMWORDS; i++)
           b.fBits[i] = this->fBits[i] ^ other.fBits[i];
    return b;
}
```

Test program

It isn't sufficient to write the class, you must also test it! The test program should automatically exercise all aspects of the class. This test program becomes part of the class documentation. It has to be available to other programmers who may need to extend or modify the existing class and who will need to retest the code after their modifications.

The test program for class Bitmap is:

```
int main()
{
    Test1();
    Test2();
    Test3();
    return EXIT_SUCCESS;
}
```

There are three Test functions (and an auxiliary function). As the names imply, the functions were developed and implemented in sequence. Basic operations were checked out using `Test1()` before the code of `Test2()` and `Test3()` was implemented.

The first test function checks out simple aspects like setting and clearing bits and getting a `Bitmap` printed:

```
void Test1()
{
    // Try setting some bits and just printing the
    // resulting Bitmap
    int somebits[] = { 0, 3, 4, 6, 9, 14, 21, 31,
                       32, 40, 48, 56,
                       64, 91, 92, 93, 94, 95,
                       500, 501
                                    };
    int n = sizeof(somebits) / sizeof(int);
    Bitmap b1;
    SetBits(b1, somebits, n);

    cout << "Test 1" << endl;
    cout << "b1:" << endl;
    b1.PrintOn(cout);

    cout << "Number of bits set in b1 is " << b1.Count()
                << endl;
    cout << "(that should have said " << n << ")" << endl;

    if(b1.TestBit(20))
           cout << "Strange I didn't think I set bit 20" << endl;

    if(!b1.TestBit(3))
           cout << "Something just ate my bitmap" << endl;

    b1.Invert();

    cout << "Look at b1, it's flipped" << endl;
    b1.PrintOn(cout);

    b1.Zero();
```

Set bits

Print

Check the count function

and the test bit function

Inversion

```
    SetBits(b1, somebits, n);
    cout << "sanity restored" << endl;

    b1.ClearBit(4);                                          Check other bit flips
    b1.FlipBit(6);                                           etc
    b1.FlipBit(7);

    cout << "Check for three small changes:" << endl;
    b1.PrintOn(cout);
}
```

An auxiliary function `SetBits()` was defined to make it easier to set several bits:

```
void SetBits(Bitmap& theBitmap, int bitstoset[], int num)
{
    for(int i=0; i<num; i++) {
           int b = bitstoset[i];
           theBitmap.SetBit(b);
           }
}
```

Function `Test2()` checks the transfers to/from files, and also the `Equals()` function:

```
void Test2()
{
    int somebits[] = { 0, 1, 2, 3, 5, 7, 11, 13,
                       17, 19, 23, 29, 31, 37,
                       41, 47
                                 };
    int n = sizeof(somebits) / sizeof(int);
    Bitmap b1;
    SetBits(b1, somebits, n);
    fstream tempout("tempbits", ios::out);
    if(!tempout.good()) {
           cout << "Sorry, Couldn't open temporary output file"
                         << endl;
           exit(1);
           }
    b1.WriteTo(tempout);                                     Write a bit map to file
    if(!tempout.good()) {
           cout << "Seem to have had problems writing to file"
                         << endl;
           exit(1);
           }
    tempout.close();
    Bitmap b2;                                               Read a bitmap from
    fstream tempin("tempbits", ios::in | ios::nocreate);     file
    if(!tempin.good()) {
           cout << "Sorry, couldn't open temp file with"
```

```
                                " the bits" << endl;
                exit(1);
                }
        b2.ReadFrom(tempin);
        if(!tempin.good()) {
                cout << "Seem to have had problems reading from"
                                " file" << endl;
                exit(1);
                }
        tempin.close();

        cout << "I wrote the bitmap :" << endl;
        b1.PrintOn(cout);
        cout << "I read the bitmap : " << endl;
        b2.PrintOn(cout);
```

Check equality

```
        if(b1.Equals(b2)) cout << "so i got as good as i gave" <<
                                endl;
        else cout << "which is sad" << endl;

}
```

The final function checks whether the implementations of `And()`, `Or()`, `XOr()` and `Not()` are mutually consistent. (It doesn't actually check whether they are right, just that they are consistent). It relies on relationships like: "A Or B" is the same as "Not (Not A And Not B)".

```
void Test3()
{
    int somebits[] = {
            2, 4, 6, 8, 16,
            33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55,
            68, 74, 80, 86,
            102, 112, 122, 132,
            145, 154, 415, 451
            };
    int n = sizeof(somebits) / sizeof(int);
    Bitmap b1;
    SetBits(b1, somebits, n);

    int otherbits[] = {
            2, 3, 6, 9, 16,
            23, 25, 27, 49, 52, 63, 75, 87, 99,
            102, 113, 132, 143,
            145, 241, 246, 362, 408, 422, 428, 429, 500, 508
            };
    int m = sizeof(otherbits) / sizeof(int);
    Bitmap b2;
    SetBits(b2, otherbits, m);
```

Evaluate Or directly

```
    Bitmap b3 = b1.Or(b2);
```

```
    Bitmap b4 = b1.Not();                                    // And by using a
    Bitmap b5 = b2.Not();                                    // sequence of And and
    Bitmap b6 = b4.And(b5);                                  // Not operations
    b6.Invert();

    if(b6.Equals(b3)) cout << "The ands, ors and nots are "  // Check consistency of
                          "consistent" << endl;              // results
    else cout << "back to the drawing board" << endl;

    b3 = b1.XOr(b2);                                         // Don't forget to check
    b4 = b1.And(b2);                                         // XOr
    b5 = b4.Not();
    b6 = b1.Or(b2);

    b6 = b6.And(b5);
    if(b3.Equals(b6)) cout << "XOr kd" << endl;
    else cout << "XOr what?" << endl;
}
```

A class isn't complete until you have written a testing program and checked it out. Class `Bitmap` passed these tests. It may not be perfect but it appears useable.

19.3 NUMBERS – A MORE COMPLEX FORM OF DATA

Bitmaps are all very well, but you don't use them that often. So, we need an example of a class that might get used more widely.

How about *integers* . You can add them, subtract them, multiply, divide, compare for equality, assign them, ... You may object "*C ++ already have integers*". This is true, but not integers like

```
9653186171569671450061340657561557651348765510 9
```

or

```
-33444555566666677777777888888888999999
```

or

```
176890346568912029856350812764539706367893255789655634373681547
453896650877195434788809984225547868696887365466149987349874216
935058327356843785265432789062357232564348 56
```

The integers we want to have represented by a class are LARGE integers; though just for our examples, we will stick to mediumish integers that can be represented in less than 100 decimal digits.

Why bother with such large numbers? Some people do need them. Casual users include: journalists working out the costs of pre-election political promises, economists estimating national debts, demographers predicting human population. The frequent users are the cryptographers.

There are simple approaches to encrypting messages, but most of these suffer from the problem that the encryption/decryption key has to be transmitted, before the message, by some separate secure route. If you have a secure route for transmitting the key, then why not use it for the message and forget about encryption?

Various more elaborate schemes have been devised that don't require a separate secure route for key transmission. Several of these schemes depend in strange ways on properties of large prime numbers, primes with 60...100 digits. So cryptographers often do get involved in performing arithmetic with large numbers.

Representing large integers

The largest integer that can be represented using hardware capabilities will be defined by the constant `INT_MAX` (in the file limits.h):

```
#define INT_MAX         2147483647
```

but that is tiny, just ten decimal digits. How might you represent something larger?

One possibility would be to use a string:

```
"123346753245098754645661"
```

Strings make some things easy (e.g. input and output) but it would be hard to do the arithmetic. The individual digits would have to be accessed and converted into numeric values in the range 0…9 before they could be processed. The index needed to get a digit would depend on the length of the string. It would all be somewhat inconvenient.

A better representation might be an array of numbers, each in the range 0…9, like that shown in Figure 19.2. The array dimension defines largest number that can be represented, so if we feel ambitious and want 1000 digit numbers then

```
fDigits[1000];
```

should make this possible.

Each digit would be an unsigned integer. As the range for the digits is only 0…9 they can be stored using "`unsigned char`". So, a large integer could be represented using an array of unsigned characters.

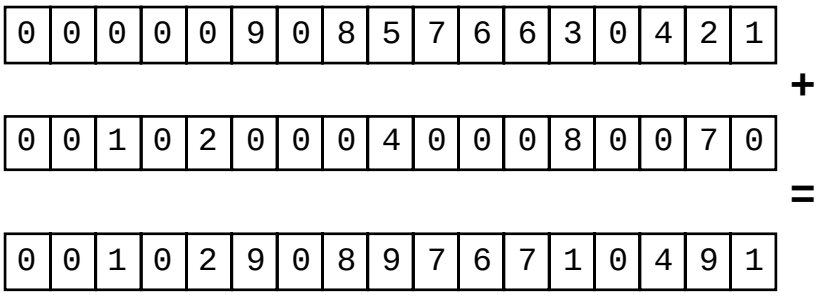

Figure 19.2 Large integers represented as arrays.

Arithmetic operations would involve loops. We could simply have the loops check all one hundred (or one thousand) digits. Generally the numbers are going to have fewer than the maximum number of digits. Performance will improve if the loops only process the necessary number of digits. Consequently, it will be useful if along with the array each "Number" object has details of its current number of digits.

Of course there is a problem with using a fixed size array. Any fixed sized number can *overflow* . Overflow was explained in Part I where binary hardware representations of numbers were discussed. Figure 19.3 illustrates the problem in the context of large integers. The implementation of arithmetic operations will need to include checks for overflow.

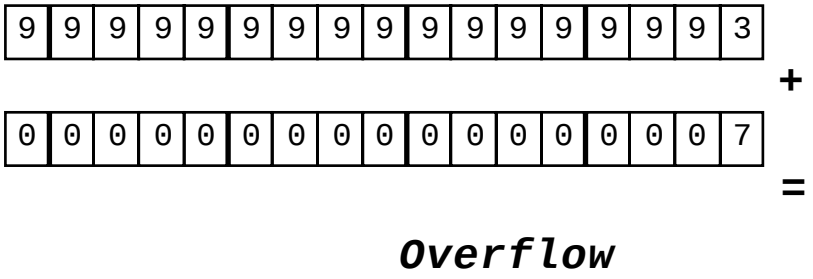

Figure 19.3 "Overflow" may occur with any fixed size number representation.

Signed numbers

There has to be a way of representing signed numbers. The smart way would be to use a representation know as "tens-complement". But that is getting really way out of our depth. So instead, we can use "sign and magnitude". The array of digits will represent the size (magnitude) of the number; there will be a separate data member that represents the sign.

Sign and magnitude representations do have a problem. They complicate some arithmetic operations. For example, you have to check the signs of both numbers that

you are combining in order to determine whether you should be doing addition or subtraction of the magnitude parts:

```
+ve number      PLUS      +ve number   =>   Do addition
+ve number      PLUS      -ve number   =>   Do subtraction
-ve number      PLUS      +ve number   =>   Do subtraction
-ve number      PLUS      -ve number   =>   Do addition
+ve number      MINUS     +ve number   =>   Do subtraction
+ve number      MINUS     -ve number   =>   Do addition
...
```

As well as performing the operation, you have to determine the sign of the result.

Doing the arithmetic

How does the arithmetic get done?

The algorithms that manipulate these large integers are going to be the same as those you learnt way back in primary school. We are going to have to have

```
addition with "carry" between columns,
subtract with "borrow" between columns,
long multiplication,
long division.
```

Figure 19.4 illustrates the mechanism of addition with "carry". This can be encoded using a simple loop:

```
carry = 0;
for(int i=0; i < lim; i++) {
    int temp;
    temp = digit[i] of first number +
                  digit[i] of second number +
                        Carry;
    if(temp>=10) {
           Carry = 1; temp -= 10;
           }
    else Carry = 0;
    digit [i] of result = temp;
    }
```

The limit for the loop will be determined by the number of digits in the larger magnitude value being added. The code has to check for overflow; this is easy, if `Carry` is non-zero at the end of the loop then overflow has occurred.

The process of subtraction with "borrow" is generally similar. Of course, you must subtract the smaller magnitude number from the larger (and reverse the sign if necessary).

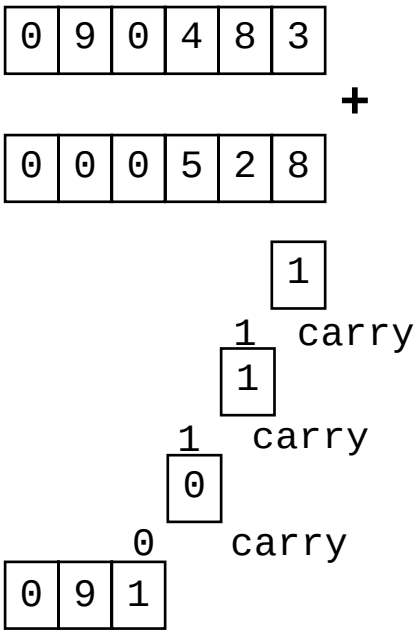

Figure 19.4 Mechanism of addition with carry.

One could do multiplication by repeated addition, but that is a bit slow. Instead a combination of "shifts", simple products, and addition steps should be used. The "shifts" handle multiplying by powers of ten, as shown in Figure 19.5. The process would have to be coded using several separate functions. An overall driver routine could step through the digits of one of the numbers, using a combination of a "times ten" and a "product" routine, and employing the functions written to handle addition.

```
for(int i=0;i<limit; i++)
    if(digit[i] != 0) {
            temp = other number * 10^i
            use product function to multiply temp
                   by digits[i]
            add product into result
            }
```

(
Check the outline multiplication code; you should find that this is the algorithm that you use to do "long multiplication".)

The hardest operation is division. A form of "long division" has to be coded. This is actually quite a hard algorithm to code. But you don't have to. It is standard, so versions are available in subroutine libraries.

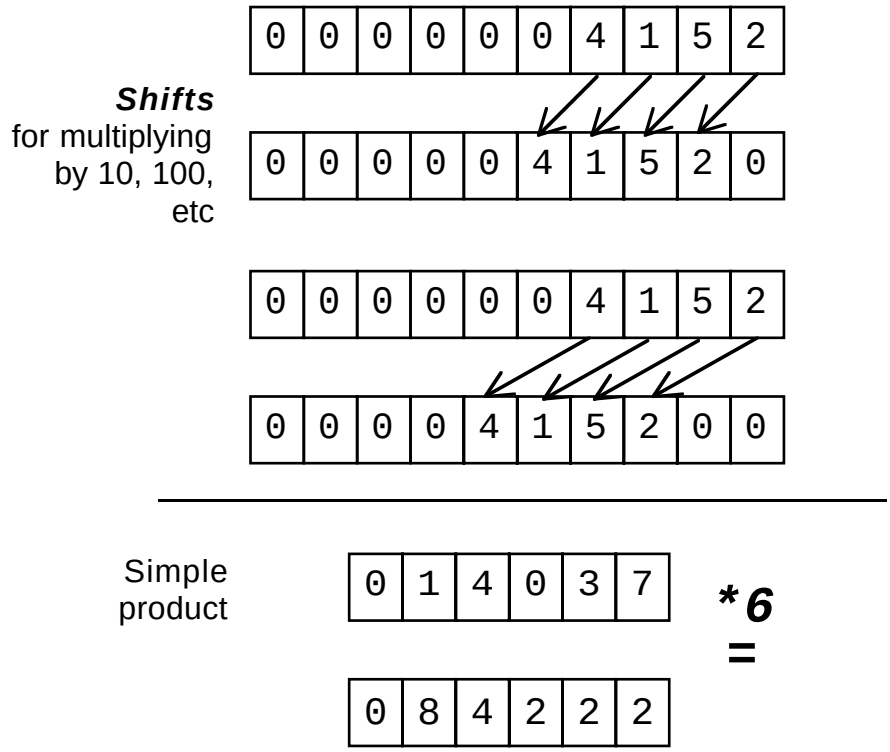

Figure 19.5 Shift and simple product sub-operations in multiplication.

P. Brinch Hansen recently published a version of the long division algorithm("Software Practice and Experience", June 1994). His paper gives a model implementation (in Pascal) with the long division process broken down into a large number of separate routines that individually are fairly simple. (The implementation of the `Number` class includes a C++ version of Brinch Hansen's code. You should read the original paper if you need a detailed explanation of the algorithm.)

Specification

Implement a class that represents multi digit integer numbers. This "Numbers" class should support the basic arithmetic operations of addition, subtraction, multiplication, and division. It should be possible to transfer Numbers to/from disk, and to print readable representations of numbers. Comparison functions for numbers should be defined. It should be possible to initialize a Number from an ordinary long integer value, or from a given string of digits.

Design

The data owned

First, consider the data. What data does an individual `Number` own? Each instance of class `Number` will have an array of unsigned long characters to store the digits. In addition, there has to be a "sign" data member (another unsigned character) and a length data member (a short integer). The size of the array of unsigned characters can be defined by a #define constant.

```
unsigned char     fDigits[kMAXDIGITS+1];  // +1 explained later
unsigned char     fPosNeg;
short             fLength;
```

Services provided by a Number

The next step is to decide what "services" a `Number` object should provide. As in the previous example, you make up a list of useful functions – things that a `Number` might be expected to do. The list is definitely not going to be the complete list of member functions. As already noted, functions like "Multiply" and "Divide" are complex and will involve auxiliary routines (like the "times 10" and "product" needed by "Multiply"). These extra auxiliary routines will be added to the list of member functions as they are identified during the design process. These extra functions will almost always be private; they will be simply an implementation detail.

When writing a test program to exercise a class, you may identify the need for additional member functions. Getting a class right does tend to be an iterative process.

The following functions form a reasonable starting point:

- Constructors.
  Several of these. The simplest should just initialize a `Number` to zero. The specification required other constructors that could initialize a `Number` to a given long and to a given string.

  Another constructor that would probably be useful would initialize a Number given an existing Number whose value is to be copied.

- Add, Subtract, Multiply, Divide.
  These combine two numbers. It seems reasonable to make them functions that return a Number as a result. A Number isn't too large (around 100-110 bytes) so returning a Number in the stack is not too unreasonable. A function giving a Number as a result would also be convenient for programs using Numbers.

- Zero.
  Probably useful, to have a function that tests whether a Number is zero, and another function that zeros out an existing Number.

- ReadFrom, WriteTo.
  Binary transfer to files.

- PrintOn.
  Output. Probably just print digit sequence with no other formatting. One hundred digit numbers just about fit on a line. If `Numbers` were significantly larger, it would be necessary to work out an effective way of printing them out over several lines.

- Compare.
  Can return a -1, 0, or 1 result depending on whether a `Number` is less than, equal to, or bigger than the `Number` that it was asked to compare itself with.

- ChangeSign.
  Probably useful.

The functions that combine `Numbers` to get a `Number` as a result will take the second `Number` as a reference argument.

Prototypes

We can jump immediately to function prototypes and complete a partial declaration of the class:

```
#define kMAXDIGITS = 100;

class Number {
public:
    Number();
    Number(long);
    Number(char numstr[]);
    …

    void   ReadFrom(ifstream& in);
    void   WriteTo(ofstream& out) const;
    void   PrintOn(ostream& printer) const;

    Number Add(const Number& other) const;
    Number Subtract(const Number& other) const;
    Number Multiply(const Number& other) const;
    Number Divide(const Number& other) const;
    int    Zero_p() const;
    void   MakeZero();

    int    Equal(const Number& other) const;
    void   ChangeSign();
    int    Compare(const Number& other) const;
private:
    // private functions still to be defined
    …
    …


    unsigned char fDigits[kMAXDIGITS+1];
    unsigned char fPosNeg; // 0 => positive, 1 => negative
    short         fLength;
};
```

It is worthwhile coming up with this kind of partial class outline fairly early in the design process, because the outline can then provide a context in which other design aspects can be considered. (The extra digit in the `fDigits` data member simplifies one part of the division routines.)

Error handling

There aren't many general design issues remaining in this class. One that should be considered is error handling. There are several errors that we know may occur. We can ignore i/o errors, they can be checked by the calling environment that transfers `Numbers` to and from disk files. But we have to deal with things like overflow. Another problem is dealing with initialization of a `Number` from a string. What should we do if we are told to initialize a `Number` with the string "Hello World" (or, more plausibly, "1203o5l7" where there are letters in the digit string)?

Chapter 26 introduces the C++ exception mechanism. Exceptions provide a means whereby you can throw error reports back to the calling code. If you can't see a general way of dealing with an error when implementing code of your class, you arrange to pass responsibility back to the caller (giving the caller sufficient information so that they know what kind of error occurred). If the caller doesn't know what to do, the program can terminate. But often, the caller will be able to take action that clears the error condition.

For now, we use a simpler mechanism in which an error report is printed and the program terminates.

Assignment of Numbers

When defining a class, you always need to think about "assignment":

```
Number a:
Number b;
…
b = a;
```

Just as in the case of `Bitmaps`, assignment of `Numbers` presents no problems. A `Number` is simply a block of bytes; the compiler will generate "memory copy" instructions to implement the assignment. Later examples will illustrate cases where you may wish to prohibit assignment operations or you may need to change the compiler's default implementation.

Detailed design and implementation

Some of the member functions of class `Number` are simple, and for these it is possible to sketch pseudo-code outlines, or even jump straight to implementation. These member functions include the default constructor:

```
Number::Number()
{
    fPosNeg = 0;
    for(int i=0; i<= kMAXDIGITS; i++) fDigits[i] = 0;
    fLength = 0;
```

```
}
```

The next two constructors are slightly more elaborate:

```
Number::Number(long lval)
{
    fPosNeg = 0;
    if(lval<0) { fPosNeg = 1; lval = -lval; }
    for(int i=0; i<= kMAXDIGITS; i++) fDigits[i] = 0;
    fLength = 0;
    while(lval) {
           fDigits[fLength++] = lval % 10;
           lval = lval / 10;
           }
}

Number::Number(char numstr[])
{
    for(int i=0;i<= kMAXDIGITS; i++) fDigits[i] = 0;
    fPosNeg = 0; /* Positive */
    i = 0;
    fLength = strlen(numstr);
```

Check for ± sign at start of string

```
    if(numstr[i] =='+') { i++; fLength--; }
    else
    if(numstr[i] =='-') { fPosNeg = 1; i++; fLength--; }

    int    pos = fLength - 1;
```

Loop to process all characters in string

```
    while(numstr[i] != '\0') {
           if(!isdigit(numstr[i])) {
                  cout << "Bad data in number input\n";
                  exit(1);
                  }
           fDigits[pos] = numstr[i] - '0';
           i++;
           pos--;
           }
```

Fix up any case where given leading zeros

```
    while((fLength>0) && (fDigits[fLength-1] == 0)) fLength--;
}
```

You shouldn't have any difficulty in understanding the constructor that initializes a `Number` from a long integer value. The code just has a loop that fills in digits starting with the units, then the tens and so forth.

The constructor that takes a character string first checks for a + or - sign. Then it loops processes all remaining characters in the string. If any are non-digits, the program terminates with an error message. Digit characters are converted into numeric values in range 0…9 and placed in the correct locations of the array. (Note how the

length of the string is used to determine the number of digits in the number, and hence the digit position in the array to be filled in with the first digit taken from the string.) The final loop just corrects for any cases where there were leading zeros in the string.

Copy constructor

Very often you need to say something like "Give me a Number just like this one". This is achieved using a copy constructor. A copy constructor takes an existing class instance as a reference argument and copies the data fields:

```
Number::Number(const Number& other)
{
    fPosNeg = other.fPosNeg;
    fLength = other.fLength;
    for(int i=0; i<= kMAXDIGITS; i++)
           fDigits[i] = other.fDigits[i];
}
```

Other simple functions include the comparison function and the input/output functions:

```
int Number::Compare(const Number& other) const
{
    if(fPosNeg != other.fPosNeg) {
           /* The numbers have opposite signs.
           If this is positive, then return "Greater" else
           return "Less". */
           return (fPosNeg == 0) ? 1 : -1;
           }

    if(fLength > other.fLength)
                  return (fPosNeg == 0) ? 1 : -1;

    if(fLength < other.fLength)
                  return (fPosNeg == 0) ? -1 : 1;

    for(int i = fLength-1;i>=0;i--)
           if(fDigits[i] > other.fDigits[i])
                         return (fPosNeg == 0) ? 1 : -1;
           else
           if(fDigits[i] < other.fDigits[i])
                         return (fPosNeg == 0) ? -1 : 1;

    return 0;
}

void Number::WriteTo(ofstream& out) const
{   // Remember, some iostreams+compilers may require (char*)
    out.write(&fDigits, sizeof(fDigits));
    out.write(&fPosNeg, sizeof(fPosNeg));
    out.write((char*)&fLength, sizeof(fLength)); // like this
}
```

Can tell result from signs if these not the same

Otherwise by number of digits if they differ

But sometimes have to compare digit by digit from top

```
void Number::PrintOn(ostream& printer) const
{
    if(fLength==0) { printer << "0"; return; }

    if(fPosNeg) printer << "-";
    int i = kMAXDIGITS;
    i = fLength - 1;

    while(i>=0) {
           printer << char('0' + fDigits[i]);
           i--;
           }

}
```

The `WriteTo()` function uses a separate call to `write()` for each data member. (the `ReadFrom()` function uses equivalent calls to `read()`). Sometimes this gets a bit clumsy; it may be worth inventing a struct that simply groups all the data members so as to facilitate disk transfers.

A few of the member functions are sufficiently simple that they can be included as inlines:

```
inline int Number::Zero_p() const
{
    return (fLength == 0);
}

inline void Number::ChangeSign()
{
    if(fPosNeg) fPosNeg = 0;
    else fPosNeg = 1;
}
```

As noted previously, such functions have to be defined in the header file with the class declaration.

Designing little programs!

The detailed design of the harder functions, like `Subtract()`, `Multiply()` and `Divide()`, really becomes an exercise in "top down functional decomposition" as illustrated by the many examples in Part III. You have the same situation. You have a "program" to write (e.g. the "Multiply program") and the data used by this program are simple (the data members of two class instances).

The "Add Program" and the "Subtract Program"

The addition and subtraction functions are to be called in code like:

```
Number a("123987115437810017316546151515");
Number b("71757656755466753443546541431765765137654");
Number c;
Number d;
c = a.Add(b);
d = a.Subtract(b);
```

Sign and magnitude representation complicates the process

The `Add()` (and `Subtract()`) functions are to return, via the stack, a value that represents the sum (difference) of the `Numbers a` and `b`. As noted in the introduction, sign and magnitude representations make things a bit more complex. You require lower level "do add" and "do subtract" functions that work using just the magnitude data. The actual `Add()` function will use the lower level "do add" if it is called to combine two values with the same sign, but if the signs are different the "do subtract" function must be called. Similarly, the `Subtract()`function will combine the magnitudes of the two numbers using the lower level "do add" and "do subtract" functions; again, the function used to combine the magnitude values depends upon the signs of the values.

There is a further complication. The actual subtraction mechanism (using "borrows" etc) only works if you subtract the smaller magnitude number from the larger magnitude number. The calls must be set up to get this correct, with the sign of the result being changed if necessary.

The following are sketches for the top level routines:

```
Add
// Returns sum of "this" and other
    initialize result to value of "this"

    if(other is zero)
           return result;

    if("this" and other have same sign) {
           do addition operation combining result with other;
           return result;
           }
    else
           return DoSubtract(other);
```

The code starts by initializing the result to the value of "this" (so for `a.Add(b)`, the result is initialized to the value of `a`). If the other argument is zero, it is possible to return the result. Otherwise we must chose between using a "do add" function to combine the partial result with the other argument, or calling `DoSubtract()` which will sort out subtractions.

The code for the top-level `Subtract()` function is similar, the conditions for using the lower-level addition and subtraction functions are switched:

```
Subtract
// Returns difference of "this" and other
    initialize result to value of "this"

    if(other is zero)
           return result;

    if("this" and other have same sign)
           return DoSubtract(other);
    else {
```

```
            do addition operation combining result with other
            return result;
            }
}
```

New member functions identified

These sketches identify at least one new "private member function" that should be added to the class. It will be useful to have:

```
int SameSign(const Number& other) const;
```

This function checks the `fPosNeg` data members of an object and the second `Number` (`other`) passed as an argument (i.e. `return (this->fPosNeg == other.fPosNeg);`). This can again be an inline function.

We also need to initialize a variable to the same value as "this" (the object executing the function). This can actually be done by an assignment; but the syntax of that kind of assignment involves a deeper knowledge of pointers and so is deferred until after the next chapter. Instead we can use a small `CopyTo()` function:

```
void Number::CopyTo(Number& dest) const
{
    // This function is not needed.
    // Can simply have dest = *this; at point of call.
    // Function CopyTo introduced to delay discussion of *this
    // until pointers covered more fully.
    dest.fLength = this->fLength;
    dest.fPosNeg = this->fPosNeg;
    for(int i=0; i<= kMAXDIGITS; i++)
            dest.fDigits[i] = this->fDigits[i];
}
```

Both `Add()` and `Subtract()` invoked the auxiliary function `DoSubtract()`. This function has to sort out the call to the actual subtraction routine so that the smaller magnitude number is subtracted from the larger (and fix up the sign of the result):

```
DoSubtract
// Set up call to subtract smaller from larger magnitude

    if(this is larger in magnitude than other)
            Initialize result to "this"
            call actual subtraction routine to subtract other
                    from result
    else
            Initialize a temporary with value of this
            set result to value of other;

            call actual subtraction routine to subtract temp.
                    from result
            if(subtracting) change sign of result
```

```
    return result;
```

Yet another auxiliary function shows up. We have to compare the magnitude of numbers. The existing `Compare()` routine considers sign and magnitude but here we just want to know which has the larger digit string. The extra function, `LargerThan()`, becomes an additional private member function. Its code will be somewhat similar to that of the `Compare()` function involving checks on the lengths of the numbers, and they have the same number of digits then a loop checking the digits starting with the most significant digit.

An outline for the low level "do add" routine was given earlier when the "carry" mechanism was explained (see Figure 19.4). The lowest level subtraction routine has a rather similar structure with a loop that uses "borrows" between units and tens, tens and hundreds, etc.

Class handling additions and subtractions

With the auxiliary functions needed for addition and subtraction the class declaration becomes:

```
class Number {
public:
    Number();
    …
    // Public Interface unchanged
    …
    int    Compare(const Number& other) const;
private:
    int    SameSign(const Number& other) const;
    void   DoAdd(const Number& other);
    Number DoSubtract(const Number& other, int subop) const;
    void   SubtractSub(const Number& other);

    int    LargerThan(const Number& other) const;
    void   CopyTo(Number& dest) const;

    unsigned char fDigits[kMAXDIGITS+1];
    unsigned char fPosNeg;
    short         fLength;
};
```

The implementation for representative functions is as follows:

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

    if(other.Zero_p()) return result;

    if(this->SameSign(other))
           return DoSubtract(other, 1);
```

```
    else {
            result.DoAdd(other);
            return result;
            }
}
```

The `Add()` routine is very similar. It should be possible to code it, and the auxiliary `DoSubtract()` from the outlines given.

Both the `DoAdd()` and the `SubtractSub()` routines work by modifying a `Number`, combining its digits with the digits of another number. The implementation of `DoAdd()`, "add with carry", is as follows:

```
void Number::DoAdd(const Number& other)
{
    int    lim;
    int    Carry = 0;
    lim = (fLength >= other.fLength) ? fLength : other.fLength;
    for(int i=0;i<lim; i++) {
            int temp;
            temp = fDigits[i] + other.fDigits[i] + Carry;
            if(temp>=10) { Carry = 1; temp -= 10; }
            else Carry = 0;
            fDigits[i] = temp;
            }
    fLength = lim;
    if(Carry) {
            if(fLength == kMAXDIGITS) Overflow();
            fDigits[fLength] = 1;
            fLength++;
            }
}
```

Loops limited by magnitude of larger number

Deal with any final carry, check for Overflow

Handling overflow

A routine has to exist to deal with overflow (indicated by carry out of the end of the number). This would not need to be a member function. The implementation used a static filescope function defined in the same file as the Numbers code; the function printed a warning message and terminated.

The function `SubtractSub()` implements the "subtract with borrow" algorithm:

```
void Number::SubtractSub(const Number& other)
{
    int    Borrow = 0;
    int    newlen = 0;
    for(int i=0;i<fLength; i++) {
            int    temp;
            temp = fDigits[i] - other.fDigits[i] - Borrow;
            if(temp < 0) { temp += 10; Borrow = 1; }
            else Borrow = 0;
            fDigits[i] = temp;
            if(temp) newlen = i+1;
```

```
            }
        fLength = newlen;
        if(fLength==0) fPosNeg = 0;
    }
```

(If the result is ±0, it is made +0. It is confusing to have positive and negative version of zero.)

In a phased implementation of the class, this version with just addition and subtraction would be tested.

The "Multiply Program"

The usage of the multiplication routine is similar to that of the addition and subtraction routines:

```
Number a("123987115437810017316546151 5");
Number b("71757656755466753443546541431765765137654");
Number c;
c = a.Multiply(b);
```

Function `Multiply()` again returns a `Number` on the stack that represents the product of the `Number` executing the routine ('a') and a second `Number` ('b').

Sign and magnitude representation doesn't result in any problems. Getting the sign of the product correct is easy; it the two values combined have the same sign the product is positive otherwise it is negative. The basics of the algorithm was outlined in the discussion related to Figure 19.5. The code implementing the algorithm is:

```
Number Number::Multiply(const Number& other) const
{
    Number Result;  // Number defaults to zero

    if(other.Zero_p()) { return Result; }

    for(int i=0;i<fLength; i++)
           if(fDigits[i]) {
                  Number temp(other);
                  temp.Times10(i);
                  temp.Product(fDigits[i]);
                  Result = Result.Add(temp);
                  }

    if(SameSign(other))
           Result.fPosNeg = 0;
    else Result.fPosNeg = 1;

    return Result;
}
```

Note use of "Copy constructor"

The routine forms a product like:

```
19704 * 6381
```

by calculating

```
4         *       6381            25524       (4*6381*10^0)
0         *      63810                0       (0*6381*10^1)
7         *     638100          4466700       (7*6381*10^2)
9         *    6381000         57429000       (9*6381*10^3)
1         *   63810000         63810000       (1*6381*10^4)
Total =                       125731224
```

Further private member functions

Two more auxiliary functions are needed. One, `Times10()`, implements the shifts to multiply by a specified power of ten. The other, `Product()`, multiplies a `Number` by a value in the range 0…9. Both change the `Number` for which they are invoked. Partial results from the individual steps accumulate in `Result`.

```
void Number::Times10(int power)
{
    if((fLength+power)>kMAXDIGITS) Overflow();
    for(int i = fLength-1;i>=0;i--) fDigits[i+power] = fDigits[i];
    for(i = power-1;i>=0;i--) fDigits[i] = 0;
    fLength += power;
}
```

Function `Times10()` needs to check that it isn't going to cause overflow by shifting a value too far over. If the operation is valid, the digits are just moved leftwards and the number filled in at the right with zeros.

The `Product()` routine is a bit like "add with carry":

```
void Number::Product(int k)
{
    int lim;
    int    Carry = 0;
    if(k==0) {
           MakeZero();
           return;
           }

    lim = fLength;
    for(int i=0;i<lim; i++) {
           int temp;
           temp = fDigits[i]*k + Carry;
           Carry = temp / 10;
           temp = temp % 10;
           fDigits[i] = temp;
           }
    if(Carry) {
           if(fLength == kMAXDIGITS) Overflow();
           fDigits[fLength] = Carry;
```

```
            fLength++;
            }
}
```

The "Divide" Program

You must remember this:

```
          0000029------------------
49853    |14678890234517709865431 75
          99706
          470829
          448677
```

(unless your primary school teachers let you use a calculator). You found long division hard then; it still is.

As with multiplication, getting the sign right is easy. If the dividend and divisor have the same sign the result is positive, otherwise it is negative. The `Divide()` routine itself can deal with the sign, and also with some special cases. If the divisor is zero, you are going to get overflow. If the divisor is larger than the dividend the result is zero. A divisor that is a one digit number is another special case handled by an auxiliary routine. The `Divide()` function is:

```
Number Number::Divide(const Number& other) const
{
    if(other.Zero_p()) Overflow();
    Number Result; // Initialized to zero

    if(!LargerThan(other))
           return Result; // return zero

    CopyTo(Result);

    if(other.fLength == 1)
           Result.Quotient(other.fDigits[0]);
    else
           Result.LongDiv(other);

    if(other.fPosNeg == fPosNeg) Result.fPosNeg = 0;
    else Result.fPosNeg = 1;

    return Result;
}
```

The auxiliary (private member) function `Quotient()` deals with simple divisions (e.g. 77982451 ÷ 3 = 2599…). You start with the most significant digit, do the division to get the digit, transfer any remainder to the next less significant digit:

```
void Number::Quotient(int k)
{
```

```
    int    Carry = 0;
    int newlen = 0;
    for(int i= fLength-1;i>=0;i--) {
           int    temp;
           temp = 10*Carry + fDigits[i];
           fDigits[i] = temp / k;
           if((newlen==0)  && (fDigits[i] !=0)) newlen = i+1;
           Carry = temp % k;
           }
    fLength = newlen;
}
```

The main routine is `LongDiv()`. Brinch Hansen's algorithm is moderately complex and involves several auxiliary functions that once again become private member functions of the class. The code is given here without the explanation and analysis (available in Brinch Hansen's paper in Software Practice and Experience):

```
void Number::LongDiv(const Number& other)
{
    int    f;
    Number d(other);
    Number r;
    CopyTo(r);
    int    m = other.fLength;
    int    n = fLength;

    f = 10 / (other.fDigits[m-1] + 1);
    r.Product(f);
    d.Product(f);

    int newlen = 0;
    for(int k = n - m; k>=0; k--) {
           int    qt;
           qt = r.Trial(d,k,m);
           if(qt==0) {
                  fDigits[k] = 0;
                  continue;
                  }

           Number dq(d);
           dq.Product(qt);
           if(r.Smaller(dq,k,m)) { qt--; dq = dq.Subtract(d); }
           if((newlen==0) && (qt !=0)) newlen = k+1;
           fDigits[k] = qt;
           r.Difference(dq,k,m);
           }
    fLength = newlen;
}

int Number::Trial(const Number& other, int k, int m)
{
```

Using lengths of numbers to get idea of size of quotient

loop filling in successive digits of quotient

Taking a guess at next digit of quotient

```
    int    km = k + m;
    int    r3;
    int d2;
    km = k + m;
    r3 = (fDigits[km]*10 + fDigits[km-1])*10 + fDigits[km-2];
    d2 = other.fDigits[m-1]*10 + other.fDigits[m-2];
    int temp = r3 / d2;
    return (temp<9) ? temp : 9;
}

int Number::Smaller(const Number& other, int k ,int m)
{
    int    i;
    i = m;
    for(;i>0;i--)
           if(fDigits[i+k] != other.fDigits[i]) break;
    return fDigits[i+k] < other.fDigits[i];
}

void Number::Difference(const Number& other, int k, int m)
{
    int    borrow = 0;
    for(int i = 0; i <= m; i++) {
           int diff = fDigits[i+k] -
                  other.fDigits[i] - borrow + 10;
           fDigits[i+k] =  diff % 10;
           borrow = 1 - (diff / 10);
           }
    if(borrow) Overflow();
}
```

Final class declaration

With all these auxiliary private member functions, the final class declaration becomes:

```
class Number {
public:
    Number();
    Number(long);
    Number(const Number& other);
    Number(char numstr[]);
    Number(istream&);

    void   ReadFrom(ifstream& in);
    void   WriteTo(ofstream& out) const;
    void   PrintOn(ostream& printer) const;

    Number Add(const Number& other) const;
    Number Subtract(const Number& other) const;
    Number Multiply(const Number& other) const;
    Number Divide(const Number& other) const;
    int    Zero_p() const;
```

```
    void   MakeZero();
    int    Equal(const Number& other) const;
    void   ChangeSign();
    int    Compare(const Number& other) const;
private:
    int    SameSign(const Number& other) const;
    void   DoAdd(const Number& other);
    Number DoSubtract(const Number& other, int subop) const;
    void   SubtractSub(const Number& other);

    void   Product(int k);
    void   Quotient(int k);

    void   Times10(int power);
    int    LargerThan(const Number& other) const;

    void   LongDiv(const Number& other);
    int    Trial(const Number& other, int, int);
    int    Smaller(const Number& other, int, int);
    void   Difference(const Number& other, int, int);

    void   CopyTo(Number& dest) const;

    unsigned char fDigits[kMAXDIGITS+1];
    unsigned char fPosNeg;
    short         fLength;
};
```

Testing

The class has to be tested. Simple programs like:

```
int main()
{
    Number a("123456789");
    Number b("-987654");
    Number c;
    c = a.Add(b); cout << "Sum "; c.PrintOn(cout); cout <<
endl;
    c = a.Subtract(b); cout << "Difference ";
                  c.PrintOn(cout); cout  << endl;
    c = a.Multiply(b); cout << "Product ";
                  c.PrintOn(cout); cout  << endl;
    c = a.Divide(b); cout << "Quotient ";
           c.PrintOn(cout); cout  << endl;
    return 0;
}
```

check the basics. As long the values are less than ten digits you can verify them on a calculator or with a spreadsheet program (or even by hand).

Automatic test programs

You would then proceed to programs such as one calculating the expressions (a-b)*(c+d) and (ac-bc+ad-bd) for different randomly chosen values for the `Numbers` a, b, c, and d (the two expressions should have the same value). A similar test routine for the multiplication and divide operations would check that (a*b*c*d)/(a*c) did equal (b*d); again the expressions would be evaluated for hundreds of randomly selected values. (A small random value can be obtained for a `Number` as follows: `long lv; lv = rand() % 25000; lv -= 12500; Number a(lv);` …. Alternatively, an additional "Randomize()" function can be added to the class that will fill in `Numbers` with randomly chosen 40 digit sequences and a random sign. You limit the size of these random values so that you could multiply them without getting overflows). Such test programs need only generate outputs if they encounter cases where the pair of supposedly equal calculated values are not in fact equal.

Optimising the code

If the class were intended for serious use in an application, you would then use a profiling tool like that described in Chapter 14. This would identify where a test program spent most time. The test program has to use `Numbers` in the same manner as the intended application. If the application uses a lot of division so should the test program.

Such an analysis was done on the code and the `Product()` routine showed up as using a significant percentage of the run time. If you look back at the code of this routine you will see that it treats multiplication by zero as a special case (just zero out the result and return); other cases involve a loop that multiplies each digit in the `Number`, getting the carry etc. Multiplying by 1 could also be made a special case. If the multiplier is 1, the `Number` is unchanged. This special case would be easy to incorporate by an extra test with a return just before the main loop in the `Product()` routine. Retesting showed that this small change reduced the time per call to `Product()` from 0.054 to 0.049μs (i.e. worthwhile). Similar analysis of other member functions could produce a number of other speed ups; however, generally, such improvements would be minor, in the order of 5% to 10% at most.

19.4 A GLANCE AT THE "IOSTREAM" CLASSES

Now that you have seen example class declarations and functions, and seen objects (class instances) being asked to perform functions, the syntax of some of the input and output statements should be becoming a little clearer.

The iostream header file contains many class declarations, including:

```
class ostream … (some detail omitted here) … {
public:
    …
    ostream &flush();
    // Flush any characters queued for output.
```

```
        ostream &put(char c);
        // Inserts a character

        ostream &write(const void *data, size_t size);
        // writes data, your version of iostream may use
        // different argument types
        …
};
```

Functions like `put()`, and `write()` are public member functions of the class. So, if you have an ostream object `out`, you can ask that object to perform a write operation using the expression:

```
out.write(&dataval, sizeof(dataval));
```

In Chapter 23, the strange "takes from" `<<` and "gives to" `>>` operators will get explained.

EXERCISES

1 Implement class Numbers, filling in all the simple omitted routines and check that these Numbers do compute.

2 Add an "integer square root" member function that will use Newton's method to find the integer x such that x is the largest integer for which x^2 <= N for a given `Number N`.

3 Often, users of these large numbers want the remainder and not the quotient of a division operation. The `LongDiv()` routine actually computes but discards the remainder. Add an extra member function that will calculate the remainder. It should be based on the existing `Divide()` and `LongDiv()` functions.

4 Write a program that tests whether a given `Number` is prime. (The program will be somewhat slow, probably not worth testing any numbers bigger than 15 digits.)