Chapter 1

Overview

I'm an explorer, okay? I like to find out.
Richard Feynman (Sykes, 1994)

This chapter reviews C++: what it is, its history and its future. Alternative object-oriented programming languages are reviewed and we examine what is actually meant by object-oriented programming and what we should expect from an object-oriented programming language.

1.1 Why C++?

In the words of the original designer of C++, B. Stroustrup (1991, Preface to the first Edition): 'C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types.... When used well, these techniques result in shorter, easier to understand, and easier to maintain programs.' Note the emphasis placed on the programmer and the program.

C++'s *major* features are:

- A superset of the C language.
- Stronger type checking than C.
- Support for data abstraction and object-oriented programming.
- Classes and abstract classes that encapsulate data and functions which operate on a **`class`**'s data, defining a given structure and behaviour.
- Inheritance and multiple inheritance, enabling the creation of hierarchies of classes.
- Support for run-time polymorphism via **`virtual`** functions, allowing a **`class`** in an arbitrary inheritance hierarchy to redefine its parent/s member functions.
- Overloading. Overloaded functions and member functions allow a function with the same name but with a different number or type of arguments, or both, to be overloaded. A **`virtual`** function of a base **`class`** can be overridden by a function in a derived **`class`** with the same name and the same number or type of arguments, or both, and return type. Overloaded operators allow you to give new functionality to existing operators when defining a **`class`**.

- Support for the creation of parametrised types, generics or templates and generic functions.
- Exception handling or error handling techniques.
- Run-time type information (RTTI) to obtain run-time identification of types and expressions.
- Namespaces to solve the problem of a single global namespace or scope.
- The Standard Template Library (STL) which provides a collection of generic data structures and algorithms.

Why choose C++ rather than C? C++ is built on C and is thus a superset of C. The C++ language adds more than just object-oriented programming capabilities to C. For example, compare the C++ **new** and **delete** operators with 'equivalent' C user-defined macros *NEW()* and *DELETE()*. The C version is:

```
#include <stdio.h>      // printf()
#include <stdlib.h>     // malloc(), free(), exit()

#define NEW(p, ptype)\
  if ((p=ptype*) malloc (sizeof (ptype)) == NULL)\
    {\
    printf ("Out of memory\n") ;\
    exit (0) ;\
    }
#define DELETE(p)
  if (p)\
    {\
    free ((char*)p ) ;\
    p = NULL ;\
    }
//...
struct X
  { /* ... */ };
//...
void main ()
  {
  struct X* xptr ;
  //...
  NEW (xptr, X);
  //...
  DELETE (xptr) ;
  //...
  }
```

requiring the inclusion of two header files (STDIO.H and STDLIB.H) and four function calls *malloc()*, *printf()*, *exit()* and *free()* in the definition of *NEW()* and *DELETE()*. *NEW()* and *DELETE()* are defined as macros rather than functions because C will not permit the manipulation of a variable without strict regard to type. Thus, it would be necessary to define separate functions for each different data type used, e.g. *New_Int(p,* ***int****)*, *New_X(p X)*,.... Therefore, to avoid this, macros (as opposed to functions) are used, which are expanded inline prior to compilation, which can be prone to errors.

The equivalent C++ version is:

```
class X
  { /* ... */ };
  //...
void main ()
  {
  X* xptr = new X ;
  //...
  delete xptr ;
  //...
  }
```

No header files or function calls are required and **new** incorporates an implicit memory allocation check. The **new** and **delete** operators allow a programmer to perform memory allocation/deallocation in a structured and consistent language-based manner. In addition, the **new** operator supports a *placement syntax* to allow an object to be placed at a particular location in memory rather than relying on the operating system to determine where an object is to be stored. Thus, the **new** and **delete** operators should be viewed as part of a more general and comprehensive memory management system and not simply allocation/deallocation mechanisms. This example clearly illustrates the compactness and elegance of C++. Although certain features in C are still available in C++, they are seldom used.

The object-oriented programming features of C++ will be discussed shortly, but first let us take a look at the history of C++.

1.2 C++'s History

For a unique insight to the history of C++ refer to *The Design and Evolution of C++* by Bjarne Stroustrup (1994)[1]. In addition, this book is an invaluable text for the reader who wishes to develop a greater understanding of the C++ language, and I would certainly recommend reading it after you have developed a reasonable knowledge of C++.

The origin of C++ begins with the languages ALGOL 60, CPL and BCPL. In 1970, Ken Thompson of AT&T's Bell Laboratories began developing a language called B, which was based on an existing typeless language, BCPL, originally developed by Martin Richards in the mid-1960s (Richards and Whitney-Strevens, 1979). BCPL (Basic CPL) was a derivative of the CPL programming language developed at both Cambridge and London Universities. Shortly afterwards, it became apparent that B was not suitable for implementing the Unix operating system, also originally designed and implemented by Ken Thompson. At the same time, Dennis Ritchie[2], also of Bell Laboratories, was developing a successor to B that was to be a compact and robust language called C. The C language was so successful that approximately 90% of the Unix operating system was rewritten in C. Today, most of Unix is written in C.

The C language quickly became very popular, particularly in the university environment, due to the availability of inexpensive compilers and to the success of Unix. In 1978, Kernighan and Ritchie published the book *The C Programming Language* (frequently referred to as the *white book*) which contained a C reference manual as an appendix. Following the publication of this key book, many compilers were referred to as 'K&R compliant'. Incidentally, the Borland C++

1 See also the articles by Ritchie (1993) and Stroustrup (1993a) on the development of C and C++ in the ACM SIGPLAN Second History of Programming Languages Conference.
2 Dennis Ritchie's home page is at `http://www.cs.bell-labs.com/who/dmr/` and contains some fascinating articles about the development of the C language.

Table 1.1 Key events in C++'s history.

1979	C with Classes development commences
1982	First paper on C with Classes (Stroustrup, 1982)
1983	C++ implementation in use; C++ named
1984	First C++ manual
1985	First commercial release (Cfront Release 1.0)
1986	*The C++ Programming Language*, 1st edn (Stroustrup, 1986)
1987	First USENIX C++ conference
1988	Zortech C++ Release
1989	Cfront Release 2.0
1990	Borland C++ Release *The Annotated C++ Reference Manual* (Ellis and Stroustrup, 1990)
1991	Cfront Release 3.0 *The C++ Programming Language*, 2nd edn (Stroustrup, 1991)
1992	DEC, Microsoft and IBM C++ Releases
1994	Draft ANSI/ISO standard
1998	ANSI/ISO standard finalised *The C++ Programming Language*, 3rd edn (Stroustrup, 1997)

compiler (version 5.x) still supports K&R language compliance. By the mid-1980s, there were more than 20 C compilers for MS-DOS, and C was in danger of being fragmented into several dialects. As a result, in 1982 an American National Standards Institute (ANSI) standardisation committee was formed to produce a standard definition of the C language. Seven years later, the ANSI X3.159-1989 standard was produced. This standard is frequently referred to as the 'ANSI C' standard. The latest C standard is *Programming Languages-C*: ANSI/ISO/IEC 9899-1999.

As a consequence of languages such as Simula and Smalltalk[3], in the late 1970s object-oriented programming was becoming an increasingly popular style of programming. In 1979, Bjarne Stroustrup, of AT&T's Computing Science Research Centre of Bell Laboratories in Murray Hill, New Jersey, began developing a language called *C with Classes*, which essentially added Simula classes to C via a preprocessor called Cpre. Initially, C with Classes was used exclusively by employees of AT&T. By 1982, C with Classes was sufficiently successful for it to be redesigned in to a new language, which was to be called C++. Also, around this time C++ received a new compiler front-end implementation (Cfront). The first commercial release of C++ was in 1985 (Cfront Release 1.0). From 1989 onwards, the emphasis was placed by the ANSI/ISO standards committee, X3J16, on developing a C++ standard, and a draft standard was first issued in 1994. In 1998 the standard was finalised, details of which can be found in Stanrdard (1998). Refer to Table 1.1 for an overview of key events in C++'s history.

C++ got its name from Rick Mascitti in 1983. The C language provides the ++ unary operator for incrementing a variable or pointer. For example, a common operation is incrementing a variable, say i, by adding the constant 1; i.e. $i = i + 1$, or, using the ++ operator, i++. Thus, C++ indicates the 'next' C.

1.3 C++'s Future

It has been stated above and throughout the literature that C++ is a superset of C. As C++ continues to develop this is gradually becoming less and less the case. C++ is becoming a separate

3 For a fascinating discussion of the early history of Smalltalk refer to Kay (1993).

language, gradually abandoning its complete compatibility with C. C++ is a continually evolving language, a *living language*. Fortunately, it is evolving to meet the needs of users.

1.4 There are Other OOP Languages

C++ is not the only programming language that supports object-oriented programming features. For instance, two languages that support OOP which have greatly influenced the development of C++ are Simula and Smalltalk. Three good works covering Simula and Smalltalk are Kirkerud (1989), Goldberg and Robson (1983) and Pinson and Weiner (1988).

Other object-oriented programming languages are Common Lisp Object System (CLOS) (Bobrow *et al.*, 1988), Eiffel (Meyer, 1991), Ada (Ada Reference Manual, 1983) and Oberon-2 (Reiser and Wirth, 1992; Mössenböck, 1993). Figure 1.1 illustrates the development of several of the more well-known object-oriented programming languages to date. For the interested reader, Saunders (1989) provides a survey of more than 80 object-oriented programming languages.

Although not the first object-oriented language, Smalltalk was instrumental in the development of object-oriented programming by demonstrating that object-oriented programming is a feasible solution to software development. Smalltalk's strengths are its excellent user interface and class library. Smalltalk is a dynamically typed language, and as a result suffers from static typing. For an introduction to Smalltalk, refer to, for example, Budd (1987).

Oberon-2 evolved from Oberon, which in turn evolved from Pascal and Modula-2. In Oberon-2, classes are represented as records which encapsulate both data and procedures, rather than by introducing an additional class construct. Oberon-2 integrates with the Oberon operating system, which provides run-time support for Oberon-2 programs. Refer to Mössenböck (1993) for details of how to obtain the Oberon-2 compiler and Oberon system.

Eiffel, although not as well-known as C++, is nevertheless a very powerful and industrially popular language and is more of a *pure* object-oriented language than C++, having not evolved primarily from a non-object-oriented language. Even though Eiffel is an object-oriented programming language, it does not forsake the ability to generate efficient program code such

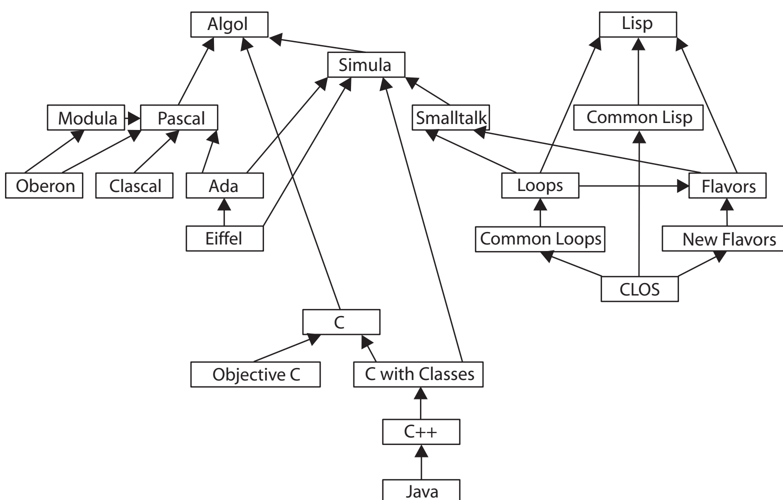

Fig. 1.1 Evolution of several object-oriented programming languages.

as that generated by C. Eiffel is a strong statically typed language and is particularly strong with regard to inheritance.

New object-oriented programming languages are continually emerging. Of the more popular ones to recently emerge are Java and Python. Java was developed by Sun Microsystems and derived from C++. Java is a simple, compact implementation of C++, but with several C++ features removed and with a few important additional features. Some of the major features removed are pointers, templates, multiple inheritance and operator overloading. Java also eliminates the preprocessor and header files. Multithreading is integral in the Java language, so that applications can be developed with multithreads; this is an important feature when developing applications for the Internet, where Java has been particularly successful. A Java language compiler generates architecture-neutral object files by generating bytecode instructions which are architecture-independent. This means that a single Java application can be developed which will execute on several operating systems. Java also supports automatic memory management. For further details of the Java programming language refer to the Java web site, Sun (2000), and Anuff (1996). Python (2000) is an interpreted, interactive, object-oriented programming language that runs on a variety of operating systems. Like Java, Python is copyrighted but freely available.

The language you choose (if you have this choice) to work with is ultimately up to you. However, I would like to add that C++ is a relatively new language and still very much growing. Personally, one of the most exciting things about the language is watching the language grow and mature. That's not to say that because the language is young it has the quirky characteristics of early versions of software. Remember that C++ has been built on C, with numerous object-oriented programming features *borrowed* from existing languages.

1.5 Programming Paradigms

There are a number of different programming styles that programmers generally adopt. These styles are mainly a result of the programming language used, just as different people trained in different disciplines, such as mathematics, physics, computer science or engineering, will tackle a given problem completely differently according to their own 'subject language', with which they are familiar or feel most comfortable. There are five main categories of programming style (Booch, 1994):

- procedure-oriented: algorithms
- object-oriented: classes and objects
- logic-oriented: goals
- rule-oriented: if–then rules
- constraint-oriented: invariant relationships

The following sections examine the current two most popular styles of programming adopted, namely procedural and data abstraction/object-oriented.

1.6 Procedural and Modular Programming

Examples of procedural (sometimes referred to as structured) programming languages are COBOL, Fortran 77/90, Pascal and C. An example of procedural programming in C is:

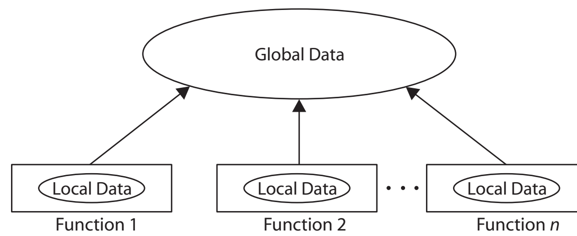

Fig. 1.2 Procedural approach.

```
#include <stdio.h>
/* get/print your name */

void main ()
  {
  char name[21] ;
  printf ("enter your name: ") ;
  gets (name) ;
  printf ("\nhello, %s", name) ;
  }
```

which simply gets a person's name and displays a message based on the name entered. A procedural program is a sequential list of statements or instructions, focusing on processing. Provided the program size remains fairly small, this approach is bearable. However, for large programs a list of statements becomes incomprehensible. Thus the use of functions came about. Each function performs a specified operation, is clearly defined and interfaces with other functions and the program; see Fig. 1.2. In the above example, `gets()` and `printf()` are functions, and each performs a different input/output operation. Consider the following hypothetical header and implementation files, which declare and define several related mathematical functions.

```
/* my_math.h, declarations of floating point math routines */
//...
double cos  (double x) ;
double sin  (double x) ;
double sqrt (double x) ;
//...

/* my_math.c, implementation of floating point math routines */
#include "my_math.h"
//...
double cos (double x)
  {
  /* code for calculating the cosine */
  }
//...
```

This grouping together of a number of related functions is often referred to as a *module*. It is worth mentioning that the concept of a module enables data, variables and functions to be hidden within the module.

A typical use of the module could be:

```
#include <stdio.h>
#include <stdlib.h>
#include <my_math.h>
/* print the square root of your age */

void main ()
  {
  char   buffer[9] ;
  double sqrt_age ;
  printf ("enter your age: ") ;
  sqrt_age = sqrt (atoi (gets (buffer)) ) ;
  printf ("\nsquare root of your age: %f", sqrt_age) ;
  }
```

In general, functions perform operations on supplied data. The emphasis is placed on doing things: do this, do that, did it work? Thus, the idea of functional abstraction enhances an algorithmic problem-solving design approach rather than a feature-based abstraction approach.

1.7 Data Abstraction

Data abstraction is the programming technique of being able to create new data types. The word *abstract* is used to reflect how a programmer abstracts features and concepts from a complex system into new data types. A programmer-defined data type is often referred to as an Abstract Data Type (ADT) to distinguish it from a built-in Fundamental Data Type (FDT), such as characters and integers. Apart from making a programmer think in terms of features, design and program organisation, data abstraction also makes programs more flexible, shorter and conceptually easier to understand. ADTs are implemented in C++ by declaring *classes*. Classes allow both data and functions to be associated with the `class` name. A `class` is similar to the C `struct` and the `RECORD` (combined with procedures) of Pascal and Oberon-2.

As an example of data abstraction, consider a complex system such as the Sierpinski gasket (also known as Sierpinski's sieve or carpet, after Vaclav Sierpinski) shown in Fig. 1.3. We could model the gasket solely in terms of integer and floating fundamental data types. Alternatively, we take the data abstraction route by observing that the gasket is composed purely of triangles placed in some kind of self-replicating fractal order. Similarly, each triangle can be decomposed into vertices or points, edges and a face. A face is composed of three edges, an edge consists of two end-points and a point is three floating-point coordinates.

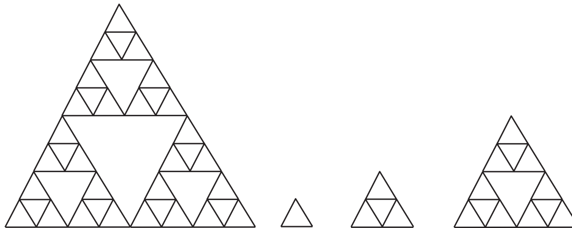

Fig. 1.3 Sierpinski's gasket and the first three stages of construction.

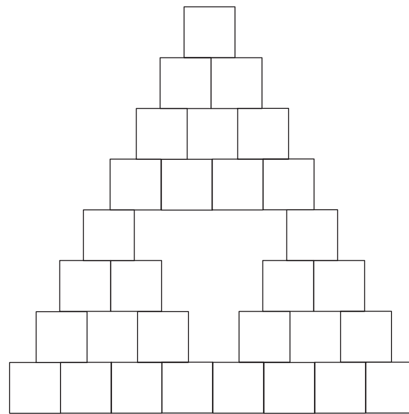

Fig. 1.4 Alternative Sierpinski gasket.

An advantage of the abstract data approach is that our point, edge and face data types can now be applied to alternative geometrical constructions, such as the similar construction shown in Fig. 1.4, which comprises squares rather than triangles. Thus, data abstraction has allowed us to generate generic types that possess common features that can be realised and utilised by a whole class of problems. It is worth pointing out that all abstract data types, however complex and abstract, can be degenerated into fundamental data types.

Data abstraction forms the foundation of object-oriented programming, and as a result a more detailed discussion of data abstraction is postponed till the next section.

1.8 Object-Oriented Programming

There are many different and diverse approaches to program development and object-oriented programming is just one of them. Object-oriented programming is not a new concept. The key ideas of object-oriented programming date back to as early as 1967, from the Simula programming language, and the early 1970s, from the Smalltalk language. Simula was developed by Dahl, Myrhaug and Nygaard (1970), whereas Smalltalk was developed at Xerox Palo Alto Research Center. Simula was based on the Algol language, with the addition of encapsulation and inheritance. C++ is an object-oriented programming language and has inherited many of the object-oriented features of its predecessors.

So what actually is object-oriented programming and how do we put this programming approach in to practice in C++? By way of a typical example we shall now examine object-oriented programming in C++. An object-oriented program consists of a number of different objects, as shown in Fig. 1.5. Objects are literally everywhere, including ourselves: see Fig. 1.6. Each object 'carries' its own set of characteristics or data and is continually sending messages to and receiving messages from other objects.

Initially, the best way to think of the object-oriented paradigm is via 'real' objects: everyday objects around us. Therefore, consider the development of a program to characterise a variety of different shapes. At this point you will probably be unfamiliar with the C++ implementation details, so try to concentrate on the object-oriented approach. At a later stage you may want to refer back to this chapter in order to fully understand the ideas presented.

If you are familiar with the procedural way of thinking about program construction it is tempting at this stage to be focusing your attention more on the details and algorithmics of

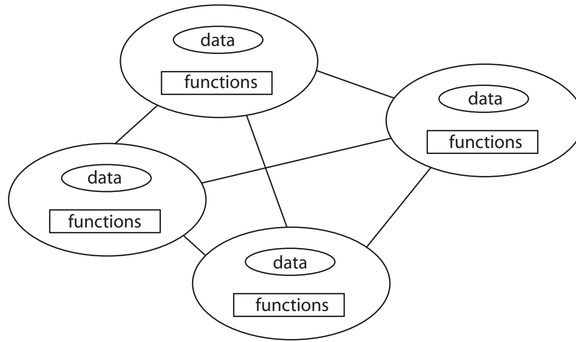

Fig. 1.5 Object approach. Objects talk to each other.

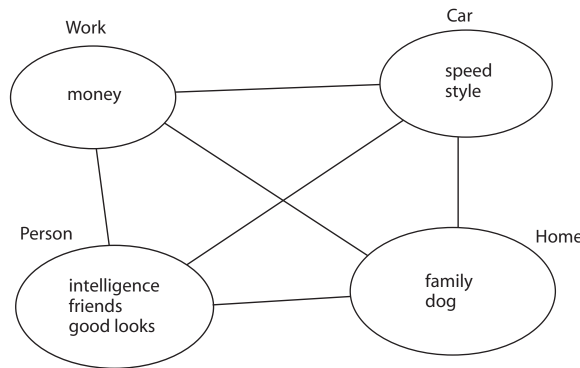

Fig. 1.6 World objects.

generating individual shapes, the division of such details into functions, and the flow of information through a shapes program into a series of steps. Let us take an alternative approach of decomposing the problem into key abstractions which embody their own unique structure and behaviour, each abstraction modelling an object in the real world.

What general properties do *all* objects around us have? They have colour, size, position relative to a given space, orientation and possibly taste, smell and so on. What kinds of object are there, or more importantly which of the infinite number of objects in the real world do we want to model? Let us limit our discussion to, say, a rectangle, a circle and a hexahedron (a three-dimensional object composed of quadrilateral faces, such as a cube). What specific properties do objects have? A circle has a centre, radius, line thickness and so on. These features are *our* abstractions, or the essential characteristics of objects as we perceive them to be. Of course[4], there is no right or wrong set of abstractions. Different sets of people choose different sets of abstractions. However, if there is a general rule – and there isn't – try to choose abstractions that are as general as possible so that they do not become too specialised and known only to a small group of people.

A **`class`**, `FuzzyShape`, that defines the general properties of all shapes is:

4 Bob Reuben

```
class FuzzyShape
  {
  public:
    Colour   colour ;
    Size     size ;
    Position position ;
    //...
    FuzzyShape () ;
    //...
  };
```

assuming that we already have classes which define position, colour and size:

```
class Position
  {
  public:
    double x, y, z ;       // (x, y, z) coordinates in space
    Position () ;
    //...
  };

class Colour
  {
  public:
    int red, green, blue ; // 3 primary components of colour
    //..
  };

class Size
  { /* ... */ };
```

The `FuzzyShape` **`class`** captures our abstraction of the characteristic features of all shapes. The **`class`** is called `FuzzyShape` rather than `Shape` to emphasise that the **`class`** does not give a clear or *crisp* definition of any particular shape.

To create an object or an instance of `FuzzyShape` so that we can do things to the object we typically write:

```
FuzzyShape fuzzy_object ;
//...
cout << "fuzzy object's position: " << fuzzy_object.position ;
cout << "fuzzy object's colour:  " << fuzzy_object.colour ;
//...
```

just as you might define a variable `var` of type **`int`**:

```
int var ;
//...
var = 7 ;
cout << "value of var: " << var ;
//...
```

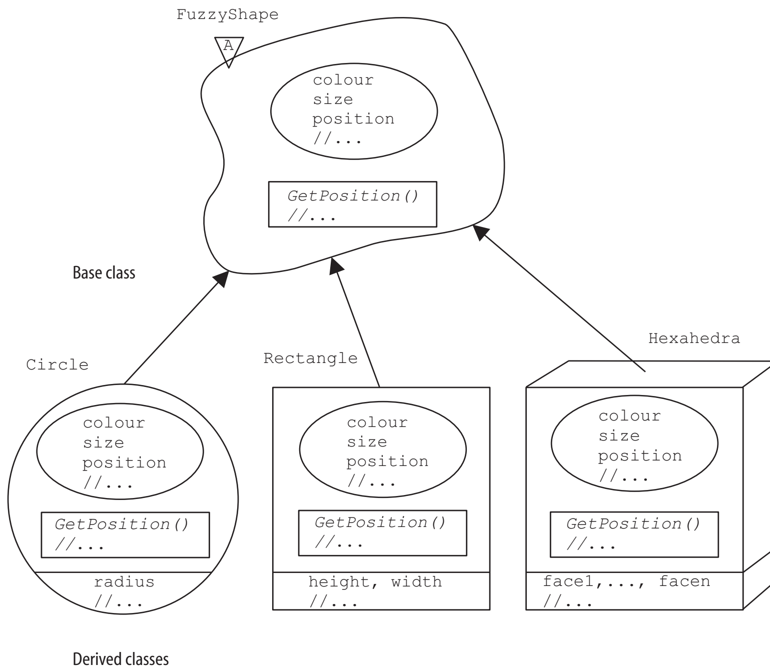

Fig. 1.7 Inheritance of shapes. The inverted triangle encompassing the uppercase letter A indicates an abstract base class in accordance with the Booch (1994) notation.

At present, our `FuzzyShape` **`class`** is nothing more than an elaborate list of properties which are difficult to operate on and are of limited use. Also, you may have noticed that the `colour`, `size` and `position` data members were declared with **`public`** access, which allows access to all clients. Generally, we are interested in separating an object's data from its client interface by *data-hiding*. Data hiding allows an object complete control in specifying what data a client needs to have access to and what data is, or should be, of no interest to a client. Hiding the internal details of an object can greatly reduce the possibility of accidental object-data corruption.

In the above example we were able to create an object of `FuzzyShape` **`class`** when such an object is meaningless in the real world, since `FuzzyShape` models general properties of shapes but is not an abstraction of any real shape. The inheritance mechanism of C++ enables us to define `FuzzyShape` as a base **`class`** and shape classes such as `Rectangle`, `Circle` and `Hexahedra` to be *derived* from **`class`** `FuzzyShape`; see Fig. 1.7[5]. In fact, `FuzzyShape` is made an *abstract* base **`class`**, since we don't want to define any objects of this **`class`**. In other words `FuzzyShape` acts only as a base **`class`** for other classes. Also, in the following modified example the data members of `FuzzyShape` are now **`protected`** to enforce data-hiding and restrict access to the **`class`** itself, derived classes and **`friend`**s:

```
class FuzzyShape
  {
  protected:
    Colour   colour ;
    Size     size ;
```

5 Where possible, the Booch (1994) notation has been adopted throughout this book for diagrammatically illustrating classes, objects and their interactions.

```
    Position position ;
    //...
  public:
    FuzzyShape () ;
    // member functions
    Position GetPosition () const ;
    void     SetPosition (const Position& p) ;
    //...
    // pure virtual member functions
    virtual void Draw () = 0 ;
    //...
  };

class Rectangle : public FuzzyShape
  {
  protected:
    double height, width ;
  public:
    Rectangle () ;
    void Draw () ;
    //...
  };

class Circle : public FuzzyShape
  {
  protected:
    double radius ;
  public:
    Circle () ;
    //...
  };

class Hexahedra : public FuzzyShape
  {
  public:
    Hexahedra () ;
  //...
  };
```

The implementation file might be:

```
FuzzyShape::FuzzyShape ()
  {
  position = 0.0 ;
  //..
  }

Position FuzzyShape::GetPosition ()
  { return position ; }

void FuzzyShape::SetPosition (const Position& p)
```

```
    { position = p ; }
  //...
```

A typical use of the above implementation could be:

```
// error: can't create an instance of an abstract class
FuzzyShape fuzzy_object ;

Rectangle rectangle_object ;
Circle    circle_object ;
//...
// error: not accessible
cout << "circle position: " << circle_object.position ;

// O.K., access thro' access member function
cout << "circle position: " << circle_object.GetPosition () ;
//...

FuzzyShape* fs_array[N] ;          // array of pointers to
                                   // FuzzyShapes
//...
fs_array[0] = &circle_object ;     // place addresses in
                                   // pointer array
fs_array[1] = &rectangle_object ;
//...
// draw all shapes using a single function call!
for (int i=0; i<N; i++)
      fs_array[i]->Draw () ;
//...
```

noting that `position` now has restricted access.

The code above illustrates that after an array of pointers to `FuzzyShape` is created and the object addresses are assigned to the pointer array we are then able to draw a variety of shapes using a single function call. This example of calling completely different functions with a single function call is an example of *polymorphism*, made possible with the aid of inheritance and **`virtual`** functions.

It should be clear from the discussion above that object-oriented programming takes a great deal more planning than traditional procedural or modular programming, but is a more natural way of solving problems.

To be considered an object-oriented programming language it is often stated that a language must support the following key elements: *abstraction*, *encapsulation*, *modularity*, *inheritance* and *polymorphism*; see, for example, Meyer's ten key OO concepts (Meyer, 1995). A brief description of each follows.

1.8.1 Abstraction

Abstraction is concerned with formulating the *essential* characteristics of an object. In the above example we characterised a circle purely in terms of a radius as well as the inherited characteristics of colour, size and position. Real objects have an infinite number of features, and it is simply not possible to model and comprehend all of them and the complex interactions between different objects. Alternatively, a complex object is broken down into a finite set

of simplified, yet essential, characteristics. Because of the infinite number of object characteristics it is important to focus on the essential characteristics when formulating an object abstraction. Formulating an abstraction is generally not as straightforward a process as it might at first appear. Different people inherently view the same object differently. What is required is a set of characteristics which are considered essential by the majority of users for a given problem domain.

1.8.2 Encapsulation

While abstraction addresses the characteristics of an object from an external viewer's perspective, encapsulation focuses on the implementation of an object. The implementation encapsulates the details about an abstraction into separate elements. The implementation should be considered a secret of the abstraction and hidden from most clients. For instance, a car is composed of literally thousands of different components, but can be viewed as an engine, chassis, body, controls etc., with the specific details concerning the implementation and operation of these different abstractions hidden from the driver. The driver is supplied with *just enough* information about the different abstractions to be able to drive the car.

Encapsulation is another word for the *black box* paradigm. Given an input, the black box generates a respective output – this is all the information we know about the box. The black box can perform any number of other operations, but we are given no clues as to the nature of the internal details of the black box.

Note that the abstraction of an object should precede its implementation. The abstraction of an object should not comprise any implementation details.

1.8.3 Modularity

Modularity is simply the decomposition of an abstraction into separate, discrete cells. It is an intuitive process to break a large complex system problem down into a manageable number of components. When programming in the large, organising a program into separate coherent modules is essential to the design of an application. Individual modules can be tested reasonably independently of the overall application. Modularity helps in designing an application to be both extendable and reusable.

1.8.4 Inheritance

Inheritance is concerned with placing separate but related abstractions into a structured hierarchy which will enable the passing of shared characteristics. Figure 1.8 illustrates a schematic `class` hierarchy in C++. A *derived* `class` inherits properties of a *base* `class`. Figure 1.8 also illustrates the concept of multiple inheritance, in which a `class` is derived from more than one base `class`. Multiple inheritance is plausible when you remember that we all have two parents.

The most important property of inheritance from a programming perspective is that it allows the reuse of program code, and consequently reduces the repetition of code.

1.8.5 Polymorphism

Polymorphism is the ability of a child object in an inheritance hierarchy to exhibit different behaviour based on the type of the object. This feature enables an object to respond to a common set of operations in different ways.

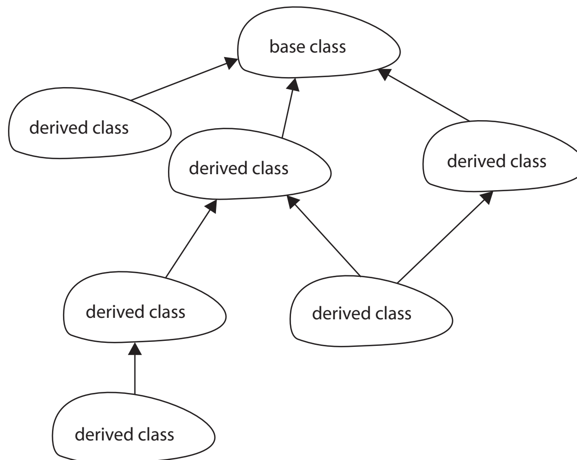

Fig. 1.8 Class hierarchy illustrating single and multiple inheritance.

In addition to the above five key elements, there are some minor elements of object-oriented programming which warrant mentioning: *typing*, *automatic memory management*, *concurrency*, *persistence* and *operator overloading*.

1.8.6 Typing

The distinction between type and **class** is very confusing, and we will assume that the two terms are equivalent. It suffices to say that *a class implements a type*.

Programming languages are referred to as untyped, weakly typed or strongly typed. A strongly typed language requires that a strict conformance of type is enforced at all times. Operations cannot be performed on an object unless the object's **class** possesses an exact signature of the operation or member function. For strongly typed languages such problems can be dealt with during compilation. If violations of type are detected at compilation then a language is referred to as *strongly statically typed*. An untyped language, such as Smalltalk, allows an object to send any message to the object's **class** even if the **class** does not know how to respond to the message. Such errors generally go unnoticed until program execution. If violations of type are allowed to pass compilation and type checking is postponed until run-time then a language is referred to as *dynamically typed*.

Regarding classes, C++ is strongly typed. Consider the shapes program (introduced above) once more:

```
class FuzzyShape
  { /* ... */ };

class Rectangle : public FuzzyShape
  {
  protected:
    double height, width ;
    //...
  public:
    //...
```

```
    double Area () ;          // compute rectangle's area
    //...
  };

class Hexahedra : public FuzzyShape
  {
  protected:
    Rectangle* face_array ;
    //...
  public:
    //...
    double SurfaceArea () ;  // compute surface area
    //...
  };
```

The following assignment is illegal in C++:

```
Hexahedra hex ;
double    area ;
//...
area = hex.Area () ;
```

because the member function `Area()` is not defined for the **`class`** or any base classes of object `hex`. However, the following assignments are legal:

```
Rectangle rect ;
Hexahedra hex ;
double    r_area, h_area ;
Position  h_pos ;
//...
r_area = rect.Area () ;
h_area = hex.SurfaceArea () ;
h_pos  = hex.GetPosition () ;
```

The more static typing performed and the more bugs found and fixed at the time of compilation the better. However, static typing alone is too constricting for object-oriented programming. The ideal solution to typing is a mixture of static typing, dynamic binding and polymorphism, which are so essential to object-oriented programming. C++ supports both strong static typing and dynamic binding, but in a controlled manner. By categorising similar abstractions into class hierarchies with a common base class and using polymorphism, a base class object is allowed to point to objects of classes within its own class hierarchy, but is excluded from pointing to objects outside of its respective hierarchy.

1.8.7 Automatic Memory Management

A large object-oriented application will create and destroy numerous objects. The manual task of allocating and freeing memory is a difficult task for a programmer involved in a large object-oriented application. Automatic memory management mechanisms track down memory used by objects that are no longer accessible to an application. Automatic memory management is an important support feature for object-oriented programming, which not only makes a pro-

grammer's life easier but can also reduce programmer errors associated with manual memory management.

C++ does not support automatic memory management, although optional memory management is being considered (Stroustrup, 1994). Several other object-oriented programming languages, such as Oberon-2 and Java (which is a derivative of C++), do support automatic memory management.

1.8.8 Concurrency

Concurrency allows objects to operate simultaneously. The Microsoft Windows NT non-preemptive multi-tasking operating system is a good example of different objects being *alive* at the same time and interacting with one another.

1.8.9 Persistence

Objects in a program require a certain amount of space in memory. Objects also exist for a certain length of time. For example, the lifetime of temporary objects in C++ can be very short, whereas certain objects can outlive the execution lifetime of a program.

An object is referred to as persistent if it outlives its creator or program execution and/or an object survives a move from its original memory address.

1.8.10 Operator Overloading

Although C++ is frequently referred to as an object-oriented programming language because of its support for classes, abstraction, encapsulation, inheritance and polymorphism, an equally important feature of C++ is operator overloading. Having created classes and abstract data types, C++ allows us to overload operators to manipulate objects just as if they were objects of the fundamental data types. Operator overloading is a particularly attractive feature when performing operations on numeric objects. For example, consider objects `m1`, `m2` and `m3` of a `Matrix` **`class`** which models mathematical matrices. If we are required to add the two `Matrix` objects `m1` and `m2` and assign the result to object `m3`, then using **`class`** member functions we could write:

```
//...
Matrix m1, m2, m3 ;
//...
m3 = Add (m1, m2) ;
```

However, operator overloading allows us to overload the addition operator (+) specifically for objects of the `Matrix` **`class`**. Then it is possible to write our `Matrix` object addition in a more natural form:

```
//...
Matrix m1, m2, m3 ;
//...
m3 = m1 + m2 ;
```

1.8.11 The Unified Modeling Language

The Unified Modeling Language (UML) is a standard object-oriented design language for specifying, visualising and documenting the components of software systems. Over the past

few years the UML has been standardised by the Object Management Group (OMG) and has now gained universal acceptance among software developers. The UML represents a collection of best engineering practices that have proven successful in the modelling of large and complex systems. The primary objectives of the UML are to (1) encourage and integrate best practices via a formal, expressive visual modelling language so they can develop and exchange meaningful models, (2) be independent of particular programming languages and development processes and (3) support higher-level development concepts such as frameworks and components. The UML is not part of the C++ language but it is worth being aware of the UML, particularly as you become a more experienced programmer. The best Web site to keep up to date on the UML is that of Rational Rose (`http://www.rational.com/uml/`) who were pioneers in the development of the UML. Three key texts on the UML are Booch *et al.* (1999), Jacobson *et al.* (1999) and Rumbaugh *et al.* (1999).

1.9 Objects

It can be seen from the discussions in the previous sections that an object is a single entity with a well-defined structure and behaviour defined in the object's **`class`**. An object does things and we do things to an object by *sending messages*, via member functions. Thus an object has a state and an identity which can change with the lifetime of the object. You should be aware that an object is also referred to as an *instance*, and member functions are called *methods* or *operations* in other object-oriented programming languages.

Thinking in terms of messages, consider again an object, `c`, of the geometric **`class`** `Circle`, which describes a circle. To determine the area of `c` we send an area message (in the form of a call to the `Circle::`*`Area()`* member function) to object `c`:

```
Circle c ;
//...
double area = c.Area () ;   // compute area of object c
```

The object itself processes the area message. If we send the same area message to other objects, such as `Rectangle` and `Triangle`, then the *message–object–process* is identical and independent of the type of object. Each object processes the area message appropriately. This is contrary to procedural programming, which would place emphasis on the area procedure rather than the objects.

1.10 Syntax, Semantics and Pragmatics

Frequently in programming we hear the words *syntax*, *semantics* and *pragmatics*. The following three sections describe what these buzzwords mean.

1.10.1 Syntax

The syntax of a programming language consists of the rules for the correct use of the language. These involve the correct grammatical construction and arrangement of the language, correct spelling, hyphenation, inflection and so on. Although the syntactical rules of a programming

language can be difficult at first, they are far simpler than the syntactical rules of a natural language. The syntax of a programming language has to be strictly adhered to.

1.10.2 Semantics

The semantics of a programming language deal with the meanings given to syntactically correct constructs of the language. Semantics also deal with the relationships between symbols and the ideas given to the symbols.

1.10.3 Pragmatics

The pragmatics of a programming language deal with the practical application of the language, not the theory. Computer programs are designed and written to solve problems, and their ultimate success depends on how they solve the problem to which they were designed. Large, complex programs are continually changing according to the changing demands placed upon them. Thus, pragmatics also deals with the evolution of a program.

1.11 Obtaining a C++ Compiler

Several commercial C++ compilers exist. To name a few of the more popular compilers there are Borland C++/C++ Builder (`http://www.borland.com/`), Microsoft Visual C++ (`http://www.microsoft.com/`), Symantec C++ (`http://www.symantec.com/`) and KAI C++ (`http://www.kai.com/`). At the time of going to press a free compiler (version 5.5) can be downloaded from the Borland Web site. Furthermore, the free GNU C/C++ compiler (gcc/g++) for the Linux operating system can be obtained from the Free Software Foundation (`http://www.gnu.org/` or specifically `http://gcc.gnu.org/`).

1.12 Applications of C++

Although my suggestions of typical applications of C++ are biased and some what limited, here are a few books which illustrate the diverse fields to which C++ is now applied.

Masters, T. (1993) *Practical Neural Network Recipes in C++*, Academic Press, New York.
Porter, A. (1993) *C++ Programming for Windows*, Osborne–McGraw-Hill, New York.
Rao, V.B. and Rao, H.V. (1993) *C++ Neural Networks and Fuzzy Logic*, MIS Press, New York.
Stevens, R.T. (1992) *Fractal Programming and Ray Tracing with C++*, Prentice Hall, Englewood Cliffs NJ.
Wilt, N. (1994) *Object-Oriented Ray Tracing in C++*, John Wiley & Sons, New York.

1.13 References for C++

If you are interested in referring to alternative or additional published work on C++, then a glance through the references at the end of the book should supply you with more than enough material to get going. Of the works referenced, my favourites are:

Lafore, R. (1991) *Object-Oriented Programming in Turbo C++*, Waite Group Press, Mill Valley CA.

You would be hard pushed to find a better introductory text to C++ than this, which is now in its second edition. Robert Lafore has an excellent style of writing and presentation, typical of authors for the Waite Group Press.

Adams, J., Leestma, S. and Nyhoff, N. (1995) *C++: An Introduction to Computing*, Prentice Hall, Englewood Cliffs NJ.

This book simultaneously introduces the reader to computing and C++. This book is an excellent text for student learning and well supported with numerous examples and programming tips.

Schildt, H. (1994) *C++ from the Ground Up* and Schildt, H. (1995) *C++: The Complete Reference*, both published by Osborne–McGraw-Hill, New York.

A list of recommended reading on C++ would not be complete without a couple of references to Schildt's many C and C++ programming books. These two, of his more recent books, are excellent first books to learn C++. When it comes to writing C++ books, Schildt is simply a magician!

Lippman, S.B. (1991) *C++ Primer*, Addison-Wesley, Reading MA.

For a more in depth study of C++, this book is a must.

Barton, J.J. and Nackman, L.R. (1994) *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*, Addison-Wesley, Reading MA.

For those of you interested in the scientific and engineering applications of C++, this well-written book is essential reading. It helps scientific and engineering programmers in the transition from Fortran and C to C++, covering a wide range of applications of C++.

Booch, G. (1994) *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, New York.

Although not specifically a text on C++ programming, this book provides an invaluable text on object-oriented programming, design and analysis. All example code in the book is implemented in C++.

Swan, T. (1999) *Tom Swan's GNU C++ for Linux*, MacMillan.

If your interests are specifically C++ for the GNU/Linux operating system then Swan's book is a must.

If you require a standard on the C++ language:

Ellis, M.A. and Stroustrup, B. (1990) *The Annotated C++ Reference Manual*, Addison-Wesley, Reading MA.

The first reference manual on C++ including expected future extensions of the language.

Stroustrup, B. (1997) *The C++ Programming Language*, 3rd edn, Addison-Wesley, Reading MA.

Although not primarily a reference manual, one is included.

Details of the latest working paper for the ANSI/ISO C++ standard can be found at Standard (1998).

If your interests are more up to date then some key journals that feature C++ are:

The C++ Report
The C++ Journal
The C/C++ Users' Journal
The Journal of Object-Oriented Programming (JOOP)
Object-Oriented Systems
Computer Language

USENIX (Unix Users' association) conference proceedings: the first C++ Workshop conference in November 1987 marked the start of a series of conferences held by USENIX in October 1988, April 1990, April 1991, August 1992 and July 1993. Refer to USENIX in the references section for further details. Also refer to the proceedings of the OOPSLA, ECOOP and C++ At Work conferences.

C++ Internet newsgroups: `comp.lang.c++`, `comp.std.c++` and `comp.object` for discussions on the C++ language, the Standard Template Library (STL) and object-oriented programming respectively. For example, the C++ newsgroup can be accessed by entering `news:comp.lang.c++` in the Location field of an Internet browser.

1.14 References for Graphics

Mortenson, M.E. (1989) *Computer Graphics: An Introduction to the Mathematics and Geometry*, Industrial Press.

This is a good first text on computer graphics. The book covers points, lines, planes, curves, surfaces, projections, coordinate systems and transformations.

I also recommend the works of Glaeser (1994), O'Rourke (1994), Preparata and Shamos (1985), Stevens (1994), Watt and Watt (1992) and Wilt (1994). More recently, Laszlo (1996) has addressed the topics of computational geometry and computer graphics in C++.

1.15 Notation

The notational convention used throughout this book is as follows:

`Courier`	User-defined identifiers, objects, classes, etc.
`Courier Italic`	User comments; C/C++ and user-defined macro and function names

Courier Bold	C++ keywords
CAPITALS	Disk directories and filenames; user-defined constants
File | Save	Save command from the File menu
.CPP, .H	Filename extensions for C++ implementation and header files respectively
//...	More program code considered not essential to the present discussion.

1.16 Summary

C++ is a superset of the C language. C++ not only adds object-oriented programming capabilities to C, but also adds numerous other features which make programming safer, easier and more enjoyable. Object-oriented programming helps focus a programmer's attention more on the design and organisation of a program than on the details. The fundamental concept of object-oriented programming is the object. Objects possess their own properties and operations – *objects do things*.

Exercises

1.1 What do you understand by object-oriented programming? Highlight the key features which define a programming language to be object-oriented.

1.2 What is meant by an *object*?

1.3 If you do not already possess or have access to the ANSI/ISO C++ standard, then try getting hold of a copy – it's an impressive document! For further details of how to obtain the C++ standard refer to Standard (1998).

1.4 What are the main differences between abstraction, encapsulation and inheritance?

1.5 Obtain a copy of and read the following paper: Parnas, D.L. (1972) On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, **15**(12), 1059–62. This paper was instrumental in introducing the concept of separating a program's implementation form its interface which is at the heart of data-hiding and encapsulation.

1.6 What are the essential abstractions of a triangle and a polygon? Identify those abstractions that are common to both objects.

1.7 Try to find out more about other object-oriented programming languages. Is C++ the best object-oriented programming language? What are the advantages and disadvantages of C++ compared with other object and non-object-oriented programming languages?