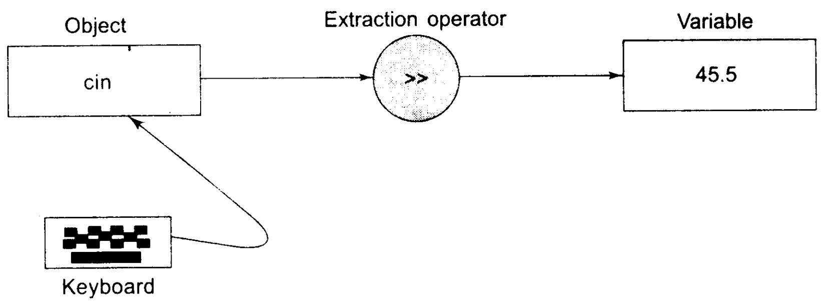

Fig. 2.2 ⇔ ***Input using extraction operator***

Cascading of I/O Operators

We have used the *insertion operator* << repeatedly in the last two statements for printing results.

The statement

```
cout << "Sum = " << sum << "\n";
```

first sends the string "Sum =" to cout and then sends the value of sum. Finally, it sends the newline character so that the next output will be in the new line. The multiple use of << in one statement is called *cascading*. When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

```
cout << "Sum = " << sum << "\n"
     << "Average = " << average << "\n";
```

This is one statement but provides two lines of output. If you want only one line of output, the statement will be:

```
cout << "Sum = " << sum << ","
     << "Average = " << average << "\n";
```

The output will be:

```
Sum = 14, Average = 7
```

We can also cascade input operator >> as shown below:

```
cin >> numberl >> number2;
```

The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to numberl and 20 to number2.

2.5 An Example with Class

One of the major features of C++ is classes. They provide a method of binding together data and functions which operate on them. Like structures in C, classes are user-defined data types.

Program 2.3 shows the use of class in a C++ program.

USE OF CLASS

```
#include <iostream>
using namespace std;
class person
{
        char name[30];
        int  age;

        public:
            void getdata(void);
            void display(void);
};
void person :: getdata(void)
{
        cout << "Enter name: ";
        cin  >> name;
        cout << "Enter age: ";
        cin  >> age;
}
void person :: display(void)
{
    cout << "\nName: " << name;
    cout << "\nAge:  " << age;
}

int main()
{
    person p;

    p.getdata();
    p.display();

    return 0;
}
```

PROGRAM 2.3

The output of Program 2.3 is:

```
Enter Name: Ravinder
Enter Age:  30

Name: Ravinder
Age:  30
```

note

cin can read only one word and therefore we cannot use names with blank spaces.

The program defines **person** as a new data of type class. The class person includes two basic data type items and two functions to operate on that data. These functions are called **member functions**. The main program uses **person** to declare variables of its type. As pointed out earlier, class variables are known as *objects*. Here, p is an object of type **person**. Class objects are used to invoke the functions defined in that class. More about classes and objects is discussed in Chapter 5.

2.6 Structure of C++ Program

As it can be seen from the Program 2.3, a typical C++ program would contain four sections as shown in Fig. 2.3. These sections may be placed in separate code files and then compiled independently or jointly.

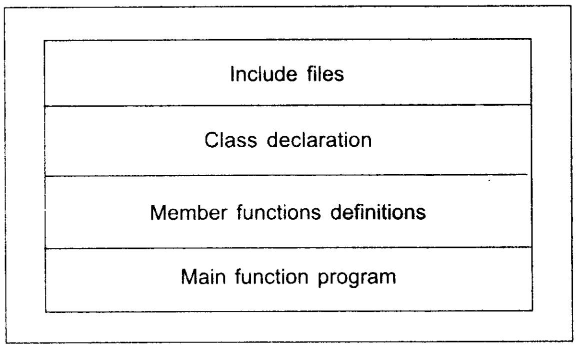

Fig. 2.3 ⇔ *Structure of a C ++ program*

It is a common practice to organize a program into three separate files. The class declarations are placed in a header file and the definitions of member functions go into another file. This approach enables the programmer to separate the abstract specification of the interface (class definition) from the implementation details (member functions definition). Finally, the main program that uses the class is placed in a third file which "includes" the previous two files as well as any other files required.

This approach is based on the concept of client-server model as shown in Fig. 2.4. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

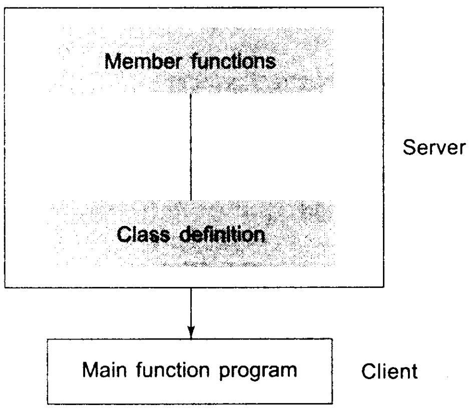

Fig. 2.4 ⇔ *The client-server model*

2.7 Creating the Source File

Like C programs, C++ programs can be created using any text editor. For example, on the UNIX, we can use vi or *ed* text editor for creating and editing the source code. On the DOS system, we can use *edlin* or any other editor available or a word processor system under non-document mode.

Some systems such as Turbo C++ provide an integrated environment for developing and editing programs. Appropriate manuals should be consulted for complete details.

The file name should have a proper file extension to indicate that it is a C++ program file. C++ implementations use extensions such as .c, .C, .cc, .cpp and .cxx. Turbo C++ and Borland C++ use .c for C programs and .cpp (C plus plus) for C++ programs. Zortech C++ system uses .cxx while UNIX AT&T version uses .C (capital C) and .cc. The operating system manuals should be consulted to determine the proper file name extensions to be used.

2.8 Compiling and Linking

The process of compiling and linking again depends upon the operating system. A few popular systems are discussed in this section.

Unix AT&T C++

The process of implementation of a C++ program under UNIX is similar to that of a C program.We should use the "CC" (uppercase) command to compile the program. Remember, we use lowercase "cc" for compiling C programs. The command

```
CC example.C
```

at the UNIX prompt would compile the C++ program source code contained in the file **example.C**. The compiler would produce an object file **example.o** and then automatically link with the library functions to produce an executable file. The default executable filename is **a.out**.

A program spread over multiple files can be compiled as follows:

```
CC file1.C file2.o
```

The statement compiles only the file **file1.C** and links it with the previously compiled **file2.o** file.This is useful when only one of the files needs to be modified. The files that are not modified need not be compiled again.

Turbo C++ and Borland C++

Turbo C++ and Borland C++ provide an integrated program development environment under MS DOS. They provide a built-in editor and a menu bar which includes options such as File, Edit, Compile and Run.

We can create and save the source files under the **File option**, and edit them under the **Edit option**. We can then compile the program under the **Compile option** and execute it under the **Run option**. The **Run option** can be used without compiling the source code. In this case, the **RUN** command causes the system to compile, link and run the program in one step. Turbo C++ being the most popular compiler, creation and execution of programs under Turbo C++ system are discussed in detail in Appendix B.

Visual C++

It is a Microsoft application development system for C++ that runs under Windows. Visual C++ is a visual programming environment in which basic program components can be selected through menu choices, buttons, icons, and other predetermined methods. Development and execution of C++ programs under Windows are briefly explained in Appendix C.

SUMMARY

- ⇔ C++ is a superset of C language.
- ⇔ C++ adds a number of object-oriented features such as objects, inheritance, function overloading and operator overloading to C. These features enable building of programs with clarity, extensibility and ease of maintenance.
- ⇔ C++ can be used to build a variety of systems such as editors, compilers, databases, communication systems, and many more complex real-life application systems.
- ⇔ C++ supports interactive input and output features and introduces a new comment symbol // that can be used for single line comments. It also supports C-style comments.
- ⇔ Like C programs, execution of all C++ programs begins at **main( )** function and ends at **return( )** statement. The header file **iostream** should be included at the beginning of all programs that use input/output operations.

⇔ All ANSI C++ programs must include **using namespace std** directive.

⇔ A typical C++ program would contain four basic sections, namely, include files section, class declaration section, member function section and main function section.

⇔ Like C programs, C++ programs can be created using any text editor.

⇔ Most compiler systems provide an integrated environment for developing and executing programs. Popular systems are UNIX AT&T C++, Turbo C++ and Microsoft Visual C++.

Key Terms

➤ #include	➤ **main()**
➤ **a.out**	➤ member functions
➤ Borland C++	➤ MS-DOS
➤ cascading	➤ **namespace**
➤ **cin**	➤ object
➤ class	➤ operating systems
➤ client	➤ operator overloading
➤ comments	➤ output operator
➤ **cout**	➤ put to operator
➤ edlin	➤ return ()
➤ extraction operator	➤ screen
➤ **float**	➤ server
➤ free-form	➤ Simula67
➤ get from operator	➤ text editor
➤ input operator	➤ Turbo C++
➤ insertion operator	➤ Unix AT&T C++
➤ **int**	➤ **using**
➤ **iostream**	➤ Visual C++
➤ **iostream.h**	➤ Windows
➤ keyboard	➤ Zortech C++

Review Questions

2.1 *State whether the following statements are TRUE or FALSE.*

(a) *Since C is a subset of C++, all C programs will run under C++ compilers.*

(b) *In C++, a function contained within a class is called a member function.*

(c) *Looking at one or two lines of code, we can easily recognize whether a program is written in C or C++.*

(d) *In C++, it is very easy to add new features to the existing structure of an object.*

(e) *The concept of using one operator for different purposes is known as oerator overloading.*

(f) *The output function printf() cannot be used in C++ programs.*

2.2 *Why do we need the preprocessor directive #include <iostream> ?*

2.3 *How does a main() function in C++ differ from main() in C?*

2.4 *What do you think is the main advantage of the comment / / in C++ as compared to the old C type comment?*

2.5 *Describe the major parts of a C++ program.*

Debugging Exercises

2.1 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
        int i = 0;
        i = i + 1;
        cout << i << " ";
        /*comment\*//i = i + 1;
        cout << i;
}
```

2.2 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
        short i=2500, j=3000;
        cout >> "i + j = " >> -(i+j);
}
```

2.3 What will happen when you run the following program?

```
#include <iostream.h>
void main()
{
        int i=10, j=5;
        int modResult=0;
        int divResult=0;

        modResult = i%j;
        cout << modResult << " ";

        divResult = i/modResult;
        cout << divResult;
}
```

2.4 Find errors, if any, in the following C++ statements.

(a) cout << "x=" x;

(b) m = 5; // n = 10; // s = m + n;

(c) cin >>x; >>y;

(d) cout << \n "Name:" << name;

(e) cout <<"Enter value:"; cin >> x;

(f) /*Addition*/ z = x + y;

Programming Exercises

2.1 *Write a program to display the following output using a single cout statement.*

```
Maths     = 90
Physics   = 77
Chemistry = 69
```

2.2 *Write a program to read two numbers from the keyboard and display the larger value on the screen.*

2.3 *Write a program to input an integer value from keyboard and display on screen "WELL DONE" that many times.*

2.4 *Write a program to read the values of a, b and c and display the value of x, where*

$$x = a / b - c$$

Test your program for the following values:

(a) $a = 250, b = 85, c = 25$

(b) $a = 300, b = 70, c = 70$

2.5 *Write a C++ program that will ask for a temperature in Fahrenheit and display it in Celsius.*

2.6 *Redo Exercise 2.5 using a class called* **temp** *and member functions.*

3

Tokens, Expressions and Control Structures

Key Concepts

- Tokens
- Keywords
- Identifiers
- Data types
- User-defined types
- Derived types
- Symbolic constants
- Declaration of variables
- Initialization
- Reference variables
- Type compatibility
- Scope resolution
- Dereferencing
- Memory management
- Formatting the output
- Type casting
- Constructing expressions
- Special assignment expressions
- Implicit conversion
- Operator overloading
- Control structures

3.1 Introduction

As mentioned earlier, C++ is a superset of C and therefore most constructs of C are legal in C++ with their meaning unchanged. However, there are some exceptions and additions. In this chapter, we shall discuss these exceptions and additions with respect to tokens and control structures.

3.2 Tokens

As we know, the smallest individual units in a program are known as tokens. C++ has the following tokens:

- Keywords
- Identifiers
- Constants
- Strings
- Operators

A C++ program is written using these tokens, white spaces, and the syntax of the language. Most of the C++ tokens are basically similar to the C tokens with the exception of some additions and minor modifications.

3.3 Keywords

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements.

Table 3.1 gives the complete set of C++ keywords. Many of them are common to both C and C++. The ANSI C keywords are shown in boldface. Additional keywords have been added to the ANSI C keywords in order to enhance its features and make it an object-oriented language. ANSI C++ standards committee has added some more keywords to make the language more versatile. These are shown separately. Meaning and purpose of all C++ keywords are given in Appendix D.

3.4 Identifiers and Constants

Identifiers refer to the names of variables, functions, arrays, classes, etc. created by the programmer. They are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

- Only alphabetic characters, digits and underscores are permitted.
- The name cannot start with a digit.
- Uppercase and lowercase letters are distinct.
- A declared keyword cannot be used as a variable name.

Table 3.1 *C++ keywords*

asm	**double**	new	**switch**
auto	**else**	operator	template
break	**enum**	private	this
case	**extern**	protected	throw
catch	**float**	public	try
char	**for**	**register**	**typedef**
class	friend	**return**	**union**
const	**goto**	**short**	**unsigned**
continue	**if**	**signed**	virtual
default	inline	**sizeof**	**void**
delete	**int**	**static**	**volatile**
do	**long**	**struct**	**while**
Added by ANSI C++			
bool	export	reinterpret_cast	typename
const_cast	false	static_cast	using
dynamic_cast	mutable	true	wchar_t
explicit	namespace	typeid	

Note: The ANSI C keywords are shown in bold face.

A major difference between C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name, ANSI C++ places no limit on its length and, therefore, all the characters in a name are significant.

Care should be exercised while naming a variable which is being shared by more than one file containing C and C++ programs. Some operating systems impose a restriction on the length of such a variable name.

Constants refer to fixed values that do not change during the execution of a program.

Like C, C++ supports several kinds of literal constants. They include integers, characters, floating point numbers and strings. Literal constant do not have memory locations. Examples:

```
123          // decimal integer
12.34        // floating point integer
037          // octal integer
0X2          // hexadecimal integer
"C++"        // string constant
'A'          // character constant
L'ab'        // wide-character constant
```

The **wchar_t** type is a wide-character literal introduced by ANSI C++ and is intended for character sets that cannot fit a character into a single byte. Wide-character literals begin with the letter L.

C++ also recognizes all the backslash character constants available in C.

note

C++ supports two types of string representation — the C-style character string and the string class type introduced with Standard C++. Although the use of the string class type is recommended, it is advisable to understand and use C-style strings in some situations. The string class type strings support many features and are discussed in detail in Chapter 15.

3.5 Basic Data Types

Data types in C++ can be classified under various categories as shown in Fig. 3.1.

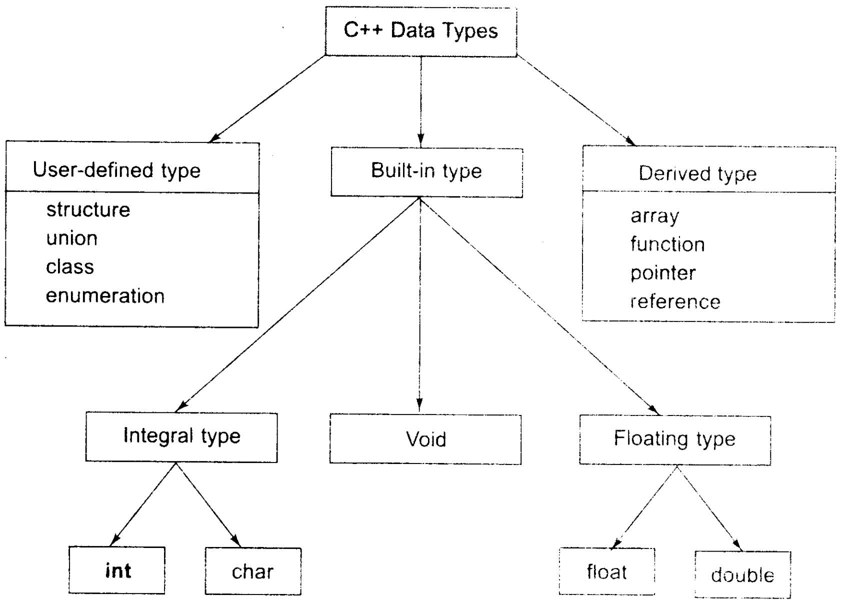

Fig. 3.1 ⇔ *Hierarchy of C++ data types*

Both C and C++ compilers support all the built-in (also known as *basic* or *fundamental*) data types. With the exception of **void**, the basic data types may have several *modifiers* preceding them to serve the needs of various situations. The modifiers **signed, unsigned, long,** and **short** may be applied to character and integer basic data types. However, the modifier **long** may also be applied to **double**. Data type representation is machine specific in C++. Table 3.2 lists all combinations of the basic data types and modifiers along with their size and range for a 16-bit word machine.

Table 3.2 *Size and range of C++ basic data types*

Type	*Bytes*	*Range*
char	1	–128 to 127
unsigned char	1	0 to 255
signed char	1	– 128 to 127
int	2	– 32768 to 32767
unsigned int	2	0 to 65535
signed int	2	– 31768 to 32767
short int	2	– 31768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	–32768 to 32767
long int	4	–2147483648 to 2147483647
signed long int	4	–2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E–38 to 3.4E+38
double	8	1.7E–308 to 1.7E+308
long double	10	3.4E–4932 to 1.1E+4932

ANSI C++ committee has added two more data types, **bool** and **wchar_t**. They are discussed in Chapter 16.

The type **void** was introduced in ANSI C. Two normal uses of **void** are (1) to specify the return type of a function when it is not returning any value, and (2) to indicate an empty argument list to a function. Example:

```
void functl(void);
```

Another interesting use of **void** is in the declaration of generic pointers. Example:

```
void *gp;        // gp becomes generic pointer
```

A generic pointer can be assigned a pointer value of any basic data type, but it may not be dereferenced. For example,

```
int *ip;         // int pointer
gp = ip;         // assign int pointer to void pointer
```

are valid statements. But, the statement,

```
*ip = *gp;
```

is illegal. It would not make sense to dereference a pointer to a **void** value.

Assigning any pointer type to a **void** pointer without using a cast is allowed in both C++ and ANSI C. In ANSI C, we can also assign a **void** pointer to a non-**void** pointer without using a cast to non-void pointer type. This is not allowed in C++. For example,

```
    void *ptr1;
    char *ptr2;
    ptr2 = ptr1;
```

are all valid statements in ANSI C but not in C++. A **void** pointer cannot be directly assigned to other type pointers in C++. We need to use a cast operator as shown below:

```
  ptr2 = (char *)ptr1;
```

3.6 User-Defined Data Types

Structures and Classes

We have used user-defined data types such as **struct** and **union** in C. While these data types are legal in C++, some more features have been added to make them suitable for object-oriented programming. C++ also permits us to define another user-defined data type known as **class** which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus of object-oriented programming. More about these data types is discussed later in Chapter 5.

Enumerated Data Type

An enumerated data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. The **enum** keyword (from C) automatically enumerates a list of words by assigning them values 0,1,2, and so on. This facility provides an alternative means for creating symbolic constants. The syntax of an **enum** statement is similar to that of the **struct** statement. Examples:

```
    enum shape{circle, square, triangle};
    enum colour{red, blue, green, yellow};
    enum position{off, on};
```

The enumerated data types differ slightly in C++ when compared with those in ANSI C. In C++, the tag names **shape, colour**, and **position** become new type names. By using these tag names, we can declare new variables. Examples:

```
    shape ellipse;          // ellipse is of type shape
    colour background;      // background is of type colour
```

ANSI C defines the types of **enums** to be **ints**. In C++, each enumerated data type retains its own separate type. This means that C++ does not permit an **int** value to be automatically converted to an **enum** value. Examples:

```
    colour background = blue;           // allowed
    colour background = 7;              // Error in C++
    colour background = (colour) 7;     // OK
```

However, an enumerated value can be used in place of an **int** value.

```
int c = red;      // valid, colour type promoted to int
```

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. We can over-ride the default by explicitly assigning integer values to the enumerators. For example,

```
enum colour{red, blue=4, green=8};
enum colour{red=5, blue, green};
```

are valid definitions. In the first case, **red** is 0 by default. In the second case, **blue** is 6 and **green** is 7. Note that the subsequent initialized enumerators are larger by one than their predecessors.

C++ also permits the creation of anonymous **enums** (i.e., **enums** without tag names). Example:

```
enum{off, on};
```

Here, **off** is 0 and **on** is 1. These constants may be referenced in the same manner as regular constants. Examples:

```
int switch_1 = off;
int switch_2 = on;
```

In practice, enumeration is used to define symbolic constants for a **switch** statement. Example:

```
enum shape
{
  circle,
  rectangle,
  triangle
};

int main()
{
  cout << "Enter shape code:";
  int code;
  cin >> code;
  while(code >= circle && code <= triangle)
  {
        switch(code)
```

constant expression, such as

```
const int size = 10;
char name[size];
```

This would be illegal in C. **const** allows us to create typed constants instead of having to use **#define** to create constants that have no type information.

As with **long** and **short**, if we use the **const** modifier alone, it defaults to **int**. For example,

```
const size = 10;
```

means

```
const int size = 10;
```

The *named constants* are just like variables except that their values cannot be changed.

C++ requires a **const** to be initialized. ANSI C does not require an initializer; if none is given, it initializes the **const** to 0.

The scoping of **const** values differs. A **const** in C++ defaults to the internal linkage and therefore it is local to the file where it is declared. In ANSI C, **const** values are global in nature. They are visible outside the file in which they are declared. However, they can be made local by declaring them as **static**. To give a **const** value an external linkage so that it can be referenced from another file, we must explicitly define it as an **extern** in C++. Example:

```
extern const total = 100;
```

Another method of naming integer constants is by enumeration as under;

```
enum {X,Y,Z};
```

This defines X, Y and Z as integer constants with values 0, 1, and 2 respectively. This is equivalent to:

```
const X = 0;
const Y = 1;
const Z = 2;
```

We can also assign values to X, Y, and Z explicitly. Example:

```
enum{X=100, Y=50, Z=200};
```

Such values can be any integer values. Enumerated data type has been discussed in detail in Section 3.6.