Contents

Preface *iii*

1. Introduction to C++ **1**
- 1.1 A Review of Structures *2*
- 1.2 Procedure-Oriented Programming System *7*
- 1.3 Object-Oriented Programming System *9*
- 1.4 Comparison of C++ with C *11*
- 1.5 Console Input/Output in C++ *12*
- 1.6 Variables in C++ *18*
- 1.7 Reference Variables in C++ *19*
- 1.8 Function Prototyping *25*
- 1.9 Function Overloading *29*
- 1.10 Default Values for Formal Arguments of Functions *31*
- 1.11 Inline Functions *34*

2. Classes and Objects **41**
- 2.1 Introduction to Classes and Objects *42*
- 2.2 Member Functions and Member Data *64*
- 2.3 Objects and Functions *84*
- 2.4 Objects and Arrays *87*
- 2.5 Namespaces *88*
- 2.6 Nested Classes *92*

3. Dynamic Memory Management **103**
- 3.1 Introduction *104*
- 3.2 Dynamic Memory Allocation *105*
- 3.3 Dynamic Memory Deallocation *110*
- 3.4 The set_new_handler() function *116*

4. Constructors and Destructors **121**
- 4.1 Constructors *122*
- 4.2 Destructors *140*
- 4.3 The Philosophy of OOPS *145*

Chapter 1

Introduction to C++

OVERVIEW

This chapter introduces the reader to the fundamentals of object-oriented programming systems (OOPS).

The chapter begins with an overview of structures, the reasons for their inclusion as a language construct in C language, and their role in procedure-oriented programming systems. Use of structures for creating new data types is described.

Also, the drawbacks of structures and the development of OOPS are elucidated.

The middle section of the chapter explains OOPS, supplemented with suitable examples and analogies to help in understanding this tricky subject.

The concluding section of the chapter includes a study of a number of new features that are implemented by C++ compilers but do not fall under the category of object-oriented features. (Language constructs of C++ that implement object-oriented features are dealt with in the next chapter.)

1.1 A Review of Structures

In order to understand procedure-oriented programming systems, let us first recapitulate our understanding of structures in C. Let us review their necessity and use in creating new data types.

The Need for Structures

There are cases where the value of one variable depends upon that of another variable.

Take the example of date. A date can be programmatically represented in C by three different integer variables taken together. Say,

```
int d,m,y;      //three integers for representing dates
```

Here 'd', 'm', and 'y' represent the day of the month, the month, and the year, respectively. Observe carefully. Although these three variables are not grouped together in the code, they actually belong to the same group. *The value of one variable may influence the value of the other two.* In order to understand this clearly, consider a function 'next_day()' that accepts the addresses of the three integers that represent a date and changes their values to represent the next day. The prototype of this function will be

```
void next_day(int *,int *,int *);  //function to calculate
                                   //the next day
```

Suppose,

```
d=1;
m=1;
y=2002;    //1st January, 2002
```

Now, if we write

```
next_day(&d,&m,&y);
```

'd' will become 2, 'm' will remain 1, and 'y' will remain 2002.

But if

```
d=28;
m=2;
y=1999;    //28th February, 1999
```

and we call the function as

```
next_day(&d,&m,&y);
```

'd' will become 1, 'm' will become 3, and 'y' will remain 1999.

Again, if

```
d=31;
m=12;
y=1999;    //31st December, 1999
```

and we call the function as

```
next_day(&d,&m,&y);
```

'd' will become 1, 'm' will become 1, and 'y' will become 2000.

As you can see, 'd', 'm', and 'y' actually belong to the same group. A change in the value of one may change the value of the other two. *But there is no language construct that actually places them in the same group*. Thus, members of the wrong group may be accidentally sent to the function (Listing 1.1)!

```
d1=28;     m1=2;      y1=1999;   //28th February, 1999
d2=19;     m2=3;      y2=1999;   //19th March, 1999
next_day(&d1,&m1,&y1);           //OK
next_day(&d1,&m2,&y2);           //What? Incorrect set passed!
```

Listing 1.1 Problem in passing groups of programmatically independent but logically dependent variables

As can be observed in Listing 1.1, there is nothing in the language itself that prevents the wrong set of variables from being sent to the function. Moreover, integer type variables that are not meant to represent dates might also be sent to the function!

Let us try arrays to solve the problem. Suppose the 'next_day()' function accepts an array as a parameter. Its prototype will be

```
void next_day(int *);
```

Let us declare date as an array of three integers.

```
int date[3];
date[0]=28;
date[1]=2;
date[2]=1999;  //28th February, 1999
```

Now, let us call the function as follows

```
next_day(date);
```

The values of 'date[0]', 'date[1]', and 'date[2]' will be correctly set to 1, 3, and 1999, respectively. Although this method seems to work, it certainly appears unconvincing. After all *any* integer array can be passed to the function, even if it does not necessarily represent a date. There is no data type of date itself. Moreover, this solution of arrays will not work if the *variables are not of the same type*. The solution to this problem is to create a data type called date itself using structures.

```
struct date     //a structure to represent dates
{
  int d, m, y;
};
```

Now the 'next_day()' function will accept the address of a variable of the structure date as a parameter. Accordingly, its prototype will be as follows

```
void next_day(struct date *);
```

Let us now call it as shown in Listing 1.2

```
struct date d1;
d1.d=28;
d1.m=2;
d1.y=1999;
next_day(&d1);
```

Listing 1.2 The need for structures

'd1.d', 'd1.m', and 'd1.y' will be correctly set to 1, 3, and 1999, respectively. Since the function takes the address of an entire structure variable as a parameter at a time, there is no chance of variables of the different groups being sent to the function.

Structure is a programming construct in C that allows us to put together variables that should be together.

Library programmers use structures to create new data types. Application programs and other library programs use these new data types by declaring variables of this data type.

```
struct date d1;
```

They call the associated functions by passing these variables or their addresses to them.

```
d1.d=31;
d1.m=12;
d1.y=2003;
next_day(&d1);
```

Finally, they use the resultant value of the passed variable further as per requirements.

```
printf("The next day is: %d/%d/%d\n", d1.d, d1.m, d1.y);
```

Output

The next day is: 01/01/2004

Creating a New Data Type Using Structures

Creation of a new data type using structures is loosely a three-step process that is executed by the library programmer.

Step 1: Put the structure definition and the prototypes of the associated functions in a header file.

```
/*Beginning of date.h*/
/*This file contains the structure definition and
prototypes of its associated functions*/
struct date
{
 int d,m,y;
};

void next_day(struct date *);          //get the next date
void get_sys_date(struct date *);      //get the current
                                       //system date
/*
  Prototypes of other useful and relevant functions to
  work upon variables of the date structure
*/
/*End of date.h*/
```

Listing 1.3 Header file containing definition of a structure variable and prototypes of its associated functions

Step 2: Put the definition of the associated functions in a source code and create a library.

```
/*Beginning of date.c*/
/*This file contains the definitions of the associated
functions*/
#include"date.h"

void next_day(struct date * p)
{
 //calculate the date that immediately follows the one
 //represented by *p and set it to *p.
}
void get_sys_date(struct date * p)
{
  //determine the current system date and set it to *p
}
/*
   Definitions of other useful and relevant functions to
   work upon variables of the date structure
*/
/*End of date.c*/
```

Listing 1.4 Defining the associated functions of a structure

Step 3: Provide the header file and the library, in whatever media, to other programmers who want to use this new data type.

Creation of a structure and creation of its associated functions are two separate steps that together constitute one complete process.

Using Structures in Application Programs

The steps to use this new data type are as follows:

Step 1: Include the header file provided by the library programmer in the source code.

```
/*Beginning of dateUser.c*/
#include"date.h"
void main()
{
   . . . .
   . . . . .
}
/*End of dateUser.c*/
```

Step 2: Declare variables of the new data type in the source code.

```
/*Beginning of dateUser.c*/
#include"date.h"
void main()
{
  struct date d;
  . . . .
  . . . .
}
/*End of dateUser.c*/
```

Step 3: Embed calls to the associated functions by passing these variables in the source code.

```
/*Beginning of dateUser.c*/
#include"date.h"
void main()
{
  struct date d;
  d.d=28;
  d.m=2;
  d.y=1999;
  next_day(&d);
  . . . .
  . . . .
}
/*End of dateUser.c*/
```

Listing 1.5 Using a structure in an application program

Step 4: Compile the source code to get the object file.

Step 5: Link the object file with the library provided by the library programmer to get the executable or another library.

1.2 Procedure-Oriented Programming System

In light of the previous discussion, let us understand the procedure-oriented programming system. The foregoing pattern of programming divides the code into functions. Data (contained in structure variables) is passed from one function to another to be read from or written into. The focus is on procedures. This programming pattern is, therefore, a feature of the procedure-oriented programming system.

In the procedure-oriented programming system, procedures are dissociated from data and are not a part of it. Instead, they receive structure variables or their addresses and work upon them. The code design is centered around procedures. While this may sound obvious, this programming pattern has its drawbacks.

The drawback with this programming pattern is that the data is not secure. It can be manipulated by *any* procedure. Associated functions that were designed by the library programmer do not have the exclusive rights to work upon the data. They are not a part of the structure definition itself. Let us see why this is a problem.

Suppose the library programmer has defined a structure and its associated functions as described above. Further, in order to perfect his/her creation, he/she has rigorously tested the associated functions by calling them from small test applications. Despite his/her best efforts, he/she cannot be sure that an application that uses the structure will be bug free. The application program might modify the structure variables, not by the associated function he/she has created, but by some code inadvertently written in the application program itself. Compilers that implement the procedure-oriented programming system do not prevent unauthorized functions from accessing/manipulating structure variables.

1.3

Now let us look at the situation from the application programmer's point of view. Consider an application of around 25,000 lines (quite common in the real programming world), in which variables of this structure have been used quite extensively. During testing, it is found that the date being represented by one of these variables has become 29th February 1999! The faulty piece of code that is causing this bug can be anywhere in the program. Therefore, debugging will involve a visual inspection of the entire code (of 25000 lines!) and will not be limited to the associated functions only.

The situation becomes especially grave if the execution of the code that is likely to corrupt the data is conditional. For example,

```
if(<some condition>)
  d.m++;  //d is a variable of date structure… d.m may
          //become 13!
```

The condition under which the bug-infested code executes may not arise during testing. While distributing his/her application, the application programmer cannot be sure that it would run successfully. Moreover, every new piece of code that accesses structure variables will have to be visually inspected and tested again to ensure that it does not corrupt the members of the structure. After all, compilers that implement procedure-oriented programming systems do not prevent unauthorized functions from accessing/manipulating structure variables.

Let us think of a compiler that enables the library programmer to assign exclusive rights to the associated functions for accessing the data members of the corresponding structure. If this happens, then our problem is solved. If a function which is not one of the intended associated functions accesses the data members of a structure variable, a compile-time error will result. To ensure a successful compile of his/her application code, the application programmer will be forced to remove those statements that access data members of structure variables. Thus, the application that arises out of a successful compile will be the outcome of a piece of code that is free of any unauthorized access to the data members of the structure variables used therein. Consequently, if a run-time error arises, attention can be focussed on the associated library functions.

It is the lack of data security of procedure-oriented programming systems that led to the object-oriented programming system (OOPS). This new system of programming is the subject of our next discussion.

1.3 Object-Oriented Programming System

In OOPS, we try to model real-world objects. But, what are real-world objects? Most real-world objects have internal parts and interfaces that enable us to operate them. These interfaces *perfectly* manipulate the internal parts of the objects. They also have the *exclusive rights* to do so.

Let us understand this concept with the help of an example. Take the case of a simple LCD projector (a real-world object). It has a fan and a lamp. There are two switches—one to operate the fan and the other to operate the lamp. However, the operation of these switches is necessarily governed by rules. If the lamp is switched on, the fan should automatically switch itself on. Otherwise, the LCD projector will get damaged. For the same reason, the lamp should automatically get switched off if the fan is switched off. In order to cater to these conditions, the switches are suitably linked with each other. The interface to the LCD projector is perfect. Further, this interface has the exclusive rights to operate the lamp and fan.

This, in fact, is a common characteristic of all real-world objects. *If a perfect interface is required to work on an object, it will also have exclusive rights to do so.*

Coming back to C++ programming, we notice a resemblance between the observed behavior of the LCD projector and the desired behavior of date structure's variables. In OOPS, with the help of a new programming construct and new keywords, associated functions of the date structure can be given exclusive rights to work upon its variables. In other words, all other pieces of code can be prevented from accessing the data members of the variables of this structure.

Compilers that implement OOPS enable data security by diligently enforcing this prohibition. They do this by throwing compile-time errors against pieces of code that

violate the prohibition. This prohibition, if enforced, will make structure variables behave like real-world objects. Associated functions that are defined to perfectly manipulate structure variables can be given exclusive rights to do so.

There is still another characteristic of real-world objects—a guaranteed initialization of data. After all, when you connect the LCD projector to the mains, it does not start up in an invalid state (fan off and lamp on). By default, either both the lamp and the fan are off or both are on. Users of the LCD projector need not do this explicitly. The same characteristic is found in all real-world objects.

1.4

Programming languages that implement OOPS enable library programmers to incorporate this characteristic of real-world objects into structure variables. Library programmers can ensure a guaranteed initialization of data members of structure variables to the desired values. For this, application programmers do not need to write code explicitly.

Two more features are incidental to OOPS. They are:

- Inheritance
- Polymorphism

Inheritance allows one structure to inherit the characteristics of an existing structure.

As we know from our knowledge of structures, a variable of the new structure will contain data members mentioned in the new structure's definition. However, because of inheritance, it will also contain data members mentioned in the existing structure's definition from which the new structure has inherited.

Further, associated functions of the new structure can work upon a variable of the new structure. For this, the address/name of a variable of the new structure is passed to the associated functions of the new structure. Again, as a result of inheritance, associated functions of the existing structure from which the new structure has inherited will also be able to work upon a variable of the new structure. For this, the address/name of a variable of the new structure is passed to the associated functions of the existing structure.

In inheritance, data and interface may both be inherited. This is expected as data and interface complement each other. The parent structure can be given the general common characteristics while its child structures can be given the more specific characteristics. This allows code reusability by keeping the common code in a common place—the base structure. Otherwise, the code would have to be replicated in all of the child structures, which will lead to maintenance nightmares. Inheritance also enables code extensibility by allowing the creation of new structures that are better suited to our requirements as compared to the existing structures.

Polymorphism, as the name suggests, is the phenomena by virtue of which the same entity can exist in two or more forms. In OOPS, functions can be made to exhibit

polymorphic behavior. Functions with different set of formal arguments can have the same name. Polymorphism is of two types: static and dynamic. We will understand how this feature enables C++ programmers to reuse and extend existing code in the subsequent chapters.

1.4 Comparison of C++ with C

C++ is an extension of C language. It is a proper superset of C language. This means that a C++ compiler can compile programs written in C language. However, the reverse is not true. A C++ compiler can understand all the keywords that a C compiler can understand. Again, the reverse is not true. Decision-making constructs, looping constructs, structures, functions, etc. are written in exactly the same way in C++ as they are in C language. Apart from the keywords that implement these common programming constructs, C++ provides a number of additional keywords and language constructs that enable it to implement the object-oriented paradigm.

The following header file shows how the structure Date, which has been our running example so far, can be rewritten in C++.

```
/*Beginning of Date.h*/
class Date      //class instead of structure
{
  private:
    int d,m,y;
  public:
    Date();
    void get_sys_date(); //associated functions appear
                         //within the class definition
    void next_day();
};
/*End of Date.h*/
```

Listing 1.6 Redefining the Date structure in C++

The following differences can be noticed between Date structure in C (Listing 1.3) and C++ (Listing 1.6):

- The keyword `class` has been used instead of `struct`.
- Two new keywords—`private` and `public`—appear in the code.
- Apart from data members, the class constructor also has member functions.

- A function that has the same name as the class itself is also present in the class. Incidentally, it has no return type specified. This is the class constructor and is discussed in Chapter 4 of this book.

The next chapter contains an in-depth study of the above class construct. It explains the meaning and implications of this new feature. It also explains how this and many more new features implement the features of OOPS, such as data hiding, data encapsulation, data abstraction, and a guaranteed initialization of data. However, before proceeding to Chapter 2, let us digress slightly and study the following:

- Console input/output in C++
- Some non-object-oriented features provided exclusively in C++ (reference variables, function overloading, default arguments, inline functions)

Remember that C++ program files have the extension '.cpp' or '.C'. The former extension is normally used for Windows or DOS-based compilers while the latter is normally used for UNIX-based compilers. The compiler's manual can be consulted to find out the exact extension.

5 Console Input/Output in C++

Console Output

The output functions in C language, such as 'printf()', can be included in C++ programs because they are anyway defined in the standard library. However, there are some more ways of outputting to the console in C++. Let us consider an example.

```
/*Beginning of cout.cpp*/
#include<iostream.h>
void main()
{
  int x;
  x=10;
  cout<<x;      //outputting to the console
}
/*End of cout.cpp*/
```

Output

10

Listing 1.7 Outputting in C++

The third statement in the 'main()' function (Listing 1.7) needs to be understood.

'cout' (pronounce see-out) is actually an object of the class 'ostream_withassign' (you can think of it as a variable of the structure 'ostream_withassign'). It stands as an alias for the console **out**put device, that is, the monitor (hence the name).

The << symbol, originally the left shift operator, has had its definition extended in C++. In the given context, it operates as the 'insertion' operator. It is a binary operator. It takes two operands. The operand on its left must be some object of the 'ostream' class. The operand on its right must be a value of some fundamental data type. The value on the right side of the 'insertion operator' is 'inserted' (hence the name) into the stream headed towards the device associated with the object on the left. Consequently, the value of 'x' is displayed on the monitor.

The file 'iostream.h' needs to be included in the source code to ensure successful compilation because the object 'cout' and the 'insertion' operator have been declared in that file.

Another object 'endl' allows us to insert a new line into the output stream. The following example illustrates this.

```
/*Beginning of endl.cpp*/
#include<iostream.h>
void main()
{
  int x,y;

  x=10;
  y=20;

  cout<<x;
  cout<<endl;  //inserting a new line by endl
  cout<<y;
}
/*End of endl.cpp*/
```

Output

```
10
20
```

Listing 1.8 Inserting a new line by 'endl'

One striking feature of the 'insertion' operator is that it works equally well with value of all fundamental types as its right-hand operand. It does not need the format specifier that are needed in the 'printf()' family of functions. The following listing exemplifies this

```
/*Beginning of cout.cpp*/
#include<iostream.h>
void main()
{
  int iVar;
  char cVar;
  float fVar;
  double dVar;
  char * cPtr;

  iVar=10;
  cVar='x';
  fVar=2.3;
  dVar=3.14159;
  cPtr="Hello World";

  cout<<iVar;
  cout<<endl;
  cout<<cVar;
  cout<<endl;
  cout<<fVar;
  cout<<endl;
  cout<<dVar;
  cout<<endl;
  cout<<cPtr;
  cout<<endl;
}
/*End of cout.cpp*/
```

Output

```
10
x
2.3
3.14159
Hello World
```

Listing 1.9 Outputting data with the 'insertion' operator

Just like the arithmetic addition operator, it is possible to cascade the 'insertion' operato The following example (Listing 1.10) is a case in point.

```
/*Beginning of coutCascade.cpp*/
#include<iostream.h>
void main()
{
  int x;
  float y;

  x=10;
  y=2.2;

  cout<<x<<endl<<y; //cascading the insertion operator
}
/*End of coutCascade.cpp*/
```

Output

10
2.2

Listing 1.10 Cascading the 'insertion' operator

It is needless to say that we can pass constants instead of variables as operands to the 'insertion' operator.

```
/*Beginning of coutMixed.cpp*/
#include<iostream.h>
void main()
{
  cout<<10<<endl<<"Hello World\n"<<3.4;
}
/*End of coutMixed.cpp*/
```

Output

10
Hello World
3.4

Listing 1.11 Outputting constants using the 'insertion' operator

In Listing 1.11, note the use of the new line character in the string that is passed as one of the operands to the 'insertion' operator.

It was mentioned in the beginning of this section that 'cout' is an object that is associated with the console. Hence, if it is the left-hand side operand of the 'insertion' operator, the value on the right is displayed on the monitor. You will learn in the chapter on stream handling that it is possible to pass objects of some other classes that are similarly associated with disk files as the left-hand side operand to the 'insertion operator'. In such cases, the values on the right get stored in the associated files.

Console Input

The input functions in C language, such as 'scanf()', can be included in C++ programs because they are anyway defined in the standard library. However, we do have some more ways of inputting from the console in C++. Let us consider an example.

```
/*Beginning of cin.cpp*/
#include<iostream.h>
void main()

{
  int x;
  cout<<"Enter a number: ";
  cin>>x; //console input in C++
  cout<<"You entered: "<<x;
}
/*End of cin.cpp*/
```

Output

Enter a number: **10**<*enter*>
You entered: 10

Listing 1.12 Inputting in C++

The third statement in the 'main()' function (Listing 1.12) needs to be understood.

'cin' (pronounce see-in) is actually an object of the class 'istream_withassign' (you can think of it as a variable of the structure 'istream_withassign'). It stands as an alias for the console input device, that is, the keyboard (hence the name).

The >> symbol, originally the right-shift operator, has had its definition extended in C++. In the given context, it operates as the 'extraction' operator. It is a binary operator and takes two operands. The operand on its left must be some object of the 'istream_withassign' class. The operand on its right must be a variable of some fundamental data type. The value for the variable on the right side of the 'extraction' operator is *extracted* (hence the name) from the stream originating from the device associated with the object on the left. Consequently, the value of 'x' is obtained from the keyboard.

The file 'iostream.h' needs to be included in the source code to ensure successful compilation because the object 'cin' and the 'extraction' operator have been declared in that file.

Again, just like the 'insertion' operator, the 'extraction' operator works equally well with variables of all fundamental types as its right-hand operand. It does not need the format specifiers that are needed in the 'scanf()' family of functions. The following listing exemplifies this.

```
/*Beginning of cin.cpp*/
#include<iostream.h>
void main()
{
  int iVar;
  char cVar;
  float fVar;

  cout<<"Enter a whole number: ";
  cin>>iVar;
  cout<<"Enter a character: ";
  cin>>cVar;
  cout<<"Enter a real number: ";
  cin>>fVar;

  cout<<"You entered: "<<iVar<<" "<<cVar<<" "<<fVar;
}
/*End of cin.cpp*/
```

Output

Enter a whole number: **10**<*enter*>
Enter a character: **x**<*enter*>
Enter a real number: **2.3**<*enter*>
You entered: 10 x 2.3

Listing 1.13 Inputting data with the extraction operator

Just like the 'insertion' operator, it is possible to cascade the 'extraction' operator. Listing 1.14 is a case in point.

```
/*Beginning of cinCascade.cpp*/
#include<iostream.h>
void main()
{
```

```
    int x,y;
    cout<<"Enter two numbers\n";
    cin>>x>>y;    //cascading the extraction operator
    cout<<"You entered "<<x<<" and "<<y;
}
/*End of cinCascade.cpp*/
```

Output

Enter two numbers
10<*enter*>
20<*enter*>
You entered 10 and 20

Listing 1.14 Cascading the 'extraction' operator

It was mentioned in the beginning of this section that **cin** is an object that is associate with the console. Hence, if it is the left-hand side operand of the 'extraction' operato the variable on the right gets its value from the keyboard. You will learn in the chapter o stream handling that it is possible to pass objects of some other classes that are similarl associated with disk files as the left-hand side operand to the 'extraction' operator. I such cases, the variable on the right gets its value from the associated files.

1.6 Variables in C++

Variables in C++ can be declared anywhere inside a function and not necessarily at it very beginning. For example,

```
#include<iostream.h>
void main()
{
  int x;
  x=10;
  cout<<"Value of x= "<<x<<endl;

  int * iPtr;  //declaring a variable in the middle of a
               //function
  iPtr=&x;
  cout<<"Address of x= "<<iPtr<<endl;
}
```

Output

Value of x= 10
Address of x= 0x21878163

Listing 1.15 Declaring variables in C++

1.7 Reference Variables in C++

First, let us understand the basics. How does the operating system (OS) display the value of variables? How are assignment operations such as 'x=y' executed during run time? A detailed answer to these questions is beyond the scope of this book. A brief study is, nevertheless, possible and necessary for a good understanding of reference variables. What follows is a simplified and tailored explanation.

The OS maintains the addresses of each variable as it allocates memory for them during run time. In order to access the value of a variable, the OS first finds the address of the variable and then transfers control to the byte whose address matches that of the variable.

Suppose the following statement is executed ('x' and 'y' are integer type variables).

```
x=y;
```

The steps followed are:

1. The OS first finds the address of 'y'.
2. The OS transfers control to the byte whose address matches this address.
3. The OS reads the value from the block of four bytes that starts with this byte (most C++ compilers cause integer type variables to occupy four bytes during run time and we will accept this value for our purpose).
4. The OS pushes the read value into a temporary stack.
5. The OS finds the address of 'x'.
6. The OS transfers control to the byte whose address matches this address.
7. The OS copies the value from the stack, where it had put it earlier, into the block of four bytes that starts with the byte whose address it has found above (address of 'x').

Notice that addresses of the variables on the left as well as on the right of the 'assignment' operator are determined. However, the value of the right-hand operand is also determined. The expression on the right must be capable of being evaluated to a value. This is an important point and must be borne in mind. It will enable us to understand a number of concepts later. Especially, you must remember that the expression on the left of the 'assignment' operator must be capable of being evaluated to a valid address at which data can be written.

Now, let us study reference variables. *A reference variable is nothing but a reference for an existing variable*. It shares the memory location with an existing variable. The syntax for declaring a reference variable is as follows:

```
<data-type> & <ref-var-name>=<existing-var-name>;
```

For example, if 'x' is an existing integer type variable and we want to declare 'iRef' as a reference to it, the statement is as follows:

```
int & iRef=x;
```

'iRef' is a reference to 'x'. This means that although 'iRef' and 'x' have separate entries in the OS, their addresses are actually the same!

Thus, a change in the value of 'x' will naturally reflect in 'iRef' and vice versa. The following program (Listing 1.16) illustrates this.

```
/*Beginning of reference01.cpp*/
#include<iostream.h>
void main()
{
  int x;
  x=10;
  cout<<x<<endl;
  int & iRef=x;        //iRef is a reference to x
  iRef=20;             //same as x=10;
  cout<<x<<endl;
  x++;                 //same as iRef++;
  cout<<iRef<<endl;
}
/*End of reference01.cpp*/
```

Output

```
10
20
21
```

Listing 1.16 Reference variables

Reference variables must be initialized at the time of declaration (otherwise the compiler will not know what address it has to record for the reference variable).

Reference variables are variables in their own right. They just happen to have the address of another variable. After their creation, they function just like any other variable.

We have just seen what happens when a value is written into a reference variable. The value of a reference variable can be read in the same way as the value of an ordinary variable is read. The following program (Listing 1.17) illustrates this:

```
/*Beginning of reference02.cpp*/
#include<iostream.h>
void main()
{
  int x,y;
  x=10;
  int & iRef=x;
  y=iRef;              //same as y=x;
  cout<<y<<endl;
  y++;                 //x and iRef unchanged
  cout<<x<<endl<<iRef<<endl<<y<<endl;
}
/*End of reference02.cpp*/
```

Output

```
10
10
10
11
```

Listing 1.17 Reading the value of a reference variable

A reference variable can be a function argument and thus change the value of the parameter that is passed to it in the function call. An illustrative example follows.

```
/*Beginning of reference03.cpp*/
#include<iostream.h>
void increment(int &);    //formal argument is a reference
                          //to the passed parameter
void main()
{
  int x;
  x=10;
  increment(x);
  cout<<x<<endl;
}
```

```
void increment(int & r)
{
  r++;              //same as x++;
}
/*End of reference03.cpp*/
```

Output

11

Listing 1.18 Passing by reference

Functions can return by reference also.

```
/*Beginning of reference04.cpp*/
#include<iostream.h>
int & larger(const int &, const int &);
void main()
{
  int x,y;
  x=10;
  y=20;
  int & r=larger(x,y);
  r=-1;
  cout<<x<<endl<<y<<endl;
}

int & larger(const int & a, const int & b)
{
  if(a>b)       //return a reference to the larger parameter
    return a;
  else
    return b;
}
/*End of reference04.cpp*/
```

Output

10

–1

Listing 1.19 Returning by reference

In the foregoing listing, 'a' and 'x' refer to the same memory location while 'b' and 'y' refer to the same memory location. From the 'larger()' function, a reference to 'b', that is, reference to 'y' is returned and stored in a reference variable 'r'. The 'larger()' function

does not return the value 'b' because the return type is `int&` and not `int`. Thus, the address of 'r' becomes equal to the address of 'y'. Consequently, any change in the value of 'r' also changes the value of 'y'. The foregoing program in Listing 1.19 can be shortened as follows.

```
/*Beginning of reference05.cpp*/
#include<iostream.h>
int & larger(const int &, const int &);
void main()
{
  int x,y;
  x=10;
  y=20;

  larger(x,y)=-1;
  cout<<x<<endl<<y<<endl;
}
int & larger(const int & a, const int & b)
{
  if(a>b)
    return a;
  else
    return b;
}
/*End of reference05.cpp*/
```

Output

```
10
-1
```

Listing 1.20 Returning by reference

The name of a non-constant variable can be placed on the left of the 'assignment' operator because a valid address—the address of the variable—can be determined from it. A call to a function that returns by reference can be placed on the left of the 'assignment' operator for the same reason.

If the compiler finds the name of a non-constant variable on the left of the 'assignment' operator in the source code, it writes instructions in the executable to

- determine the address of the variable,
- transfer control to the byte that has that address, and
- write the value on the right of the 'assignment' operator into the block that begins with the byte found above.

A function that returns by reference primarily returns the address of the returned variable
If the call is found on the left of the assignment operator, the compiler writes necessary
instructions in the executable to

- transfer control to the byte whose address is returned by the function and
- write the value on the right of the assignment operator into the block that begins with the byte found above.

The name of a variable can be placed on the right of the 'assignment' operator. A call to a function that returns by reference can be placed on the right of the 'assignment' operator for the same reason.

If the compiler finds the name of a variable on the right of the 'assignment' operator in the source code, it writes instructions in the executable to

- determine the address of the variable,
- transfer control to the byte that has that address,
- read the value from the block that begins with the byte found above, and
- push the read value into the stack.

A function that returns by reference primarily returns the address of the returned variable
If the call is found on the right of the 'assignment' operator, the compiler writes necessary
instructions in the executable to

- transfer control to the byte whose address is returned by the function,
- read the value from the block that begins with the byte found above, and
- push the read value into the stack.

A constant cannot be placed on the left of the 'assignment' operator. This is because constants do not have a valid address. Moreover, how can a constant be changed? Functions that return by value, return the value of the returned variable, which is a constant. Therefore, a call to a function that returns by value cannot be placed on the left of the 'assignment' operator.

You may notice that the formal arguments of the 'larger()' function in the foregoing listing have been declared as constant references because they are not supposed to change the values of the passed parameters even accidentally.

We must avoid returning a reference to a local variable. For example,

```
/*Beginning of reference06.cpp*/
#include<iostream.h>
int & abc();
void main()
{
  abc()=-1;
}

int & abc()
{
  int x;
  return x;     //returning reference of a local variable
}
/*End of reference06.cpp*/
```

Listing 1.21 Returning the reference of a local variable

The problem with the above program is that when the 'abc()' function terminates, 'x' will go out of scope. Consequently, the statement

```
abc()=-1;
```

in the 'main()' function will write '–1' in an unallocated block of memory. This can lead to run-time errors.

1.8 Function Prototyping

Function prototyping is necessary in C++. A prototype describes the function's interface to the compiler. It tells the compiler the return type of the function as well as the number, type, and sequence of its formal arguments.

The general syntax of function prototype is as follows:

return_type function_name(argument_list);

For example,

```
int add(int, int);
```

This prototype indicates that the 'add()' function returns a value of integer type and takes two parameters both of integer type.

Since a function prototype is also a statement, a semicolon must follow it.

Providing names to the formal arguments in function prototypes is optional. Even if such names are provided, they need not match those provided in the function definition. For example,

```
/*Beginning of funcProto.cpp*/
#include<iostream.h>
int add(int,int);                    //function prototype

void main()
{
  int x,y,z;
  cout<<"Enter a number: ";
  cin>>x;
  cout<<"Enter another number: ";
  cin>>y;
  z=add(x,y);                        //function call
  cout<<z<<endl;
}
int add(int a,int b)          //function definition
{
  return (a+b);
}
/*End of funcProto.cpp*/
```

Output

Enter a number: **10**<*enter*>
Enter another number: **20**<*enter*>
30

Listing 1.22 Function prototyping

Why is prototyping important? By making prototyping necessary, the compiler ensure the following:

- The return value of a function is handled correctly.
- Correct number and type of arguments are passed to a function.

Let us discus these points.

Consider the following statement in Listing 1.22:

```
int add(int, int);
```

The prototype tells the compiler that the 'add()' function returns an integer type value. Thus, the compiler knows how many bytes have to be retrieved from the place where the

'add()' function is expected to write its return value and how these bytes are to be interpreted.

In the absence of prototypes, the compiler will have to assume the type of the returned value. Suppose, it assumes that the type of the returned value is an integer. However, the called function may return a value of an incompatible type (say a structure type). Now suppose an integer type variable is equated to the call to a function where the function call precedes the function definition. In this situation, the compiler will report an error against the function definition and not the function call. This is because the function call abided by its assumption, but the definition did not. However, if the function definition is in a different file to be compiled separately, then no compile-time errors will arise. Instead, wrong results will arise during run time as the following program shows.

```
/*Beginning of def.c*/
/*function definition*/
struct abc
{
  char a;
  int b;
  float c;
};

struct abc test()
{
  struct abc a1;
  a1.a='x';
  a1.b=10;
  a1.c=1.1;
  return a1;
}
/*End of def.c*/

/*Beginning of driver.c*/
void main()
{
  int x;
  x=test();          //no compile time error!!
  printf("%d",x);
}
/*End of driver.c*/
```

Output

1688

Listing 1.23 Absence of function prototype produces weird results

A compiler that does not enforce prototyping will definitely compile the above progra
But then it will have no way of knowing what type of value the 'test()' function retur
Therefore, erroneous results will be obtained during run time as the output of Listi
1.23 clearly shows.

Since the C++ compiler necessitates function prototyping, it will report an error agai
the function call because no prototype has been provided to resolve the function ca
Again, if the correct prototype *is* provided, the compiler will still report an error sin
this time the function call does not match the prototype. The compiler will not be able
convert a 'struct abc' to an integer. *Thus, function prototyping guarantees protecti*
from errors arising out of incorrect function calls.

What happens if the function prototype and the function call do not match? Such a situati
cannot arise. The function prototype and the function definition are both created by t
same person, that is, the library programmer. The library programmer puts the functio
prototype in a header file. He/she provides the function's definition in a library. Th
application programmer includes the header file in his/her application program file
which the function is called. He/she creates an object file from this application progra
file and links this object file to the library to get an executable file.

The function's prototype also tells the compiler that the 'add()' function accepts tw
parameters. If the program fails to provide such parameters, the prototype enables th
compiler to detect the error. A compiler that does not enforce function prototyping wi
compile a function call in which an incorrect number and/or type of parameters ha
been passed. Run-time errors will arise as in the foregoing case.

Finally, *function prototyping produces automatic type conversion wherever appropriat*
We take the case of compilers that do not enforce prototyping. Suppose, a function expec
an integer type value (assuming integers occupy four bytes) but a value of double ty
(assuming doubles occupy eight bytes) is wrongly passed. During run time, the value
only the first four bytes of the passed eight bytes will be extracted. This is obvious
undesirable. However, the C++ compiler automatically converts the double type val
into an integer type. This is because it inevitably encounters the function prototype befo
encountering the function call and therefore knows that the function expects an integ
type value. However, it must be remembered that such automatic type conversions due
function prototypes occur only when it makes sense. For example, the compiler wi
prevent an attempted conversion from a structure type to integer type.

Nevertheless, can the same benefits not be realized without prototyping? Is it not possibl
for the compiler to simply scan the rest of the source code and find out how the functio
has been defined? There are two reasons why this solution is inappropriate. They are:

- It is inefficient. The compiler will have to suspend the compilation of the lin
containing the function call and search the rest of the file.

- Most of the times the function definition is not contained in the file where it is called. It is usually contained in a library.

Such compile-time checking for prototypes is known as *static type checking.*

1.9 Function Overloading

C++ allows two or more functions to have the same name. For this, however, they must have different signatures. *Signature of a function means the number, type, and sequence of formal arguments of the function.* In order to distinguish amongst the functions with the same name, the compiler expects their signatures to be different. Depending upon the type of parameters that are passed to the function call, the compiler decides which of the available definitions will be invoked. For this, function prototypes should be provided to the compiler for matching the function calls. Accordingly, the linker, during link time, links the function call with the correct function definition.

Listing 1.24 clarifies this.

```
/*Beginning of funcOverload.cpp*/
#include<iostream.h>
int add(int,int);          //first prototype
int add(int,int,int);      //second prototype

void main()
{
  int x,y;
  x=add(10,20);            //matches first prototype
  y=add(30,40,50);         //matches second prototype
  cout<<x<<endl<<y<<endl;
}

int add(int a,int b)
{
  return(a+b);
}

int add(int a,int b,int c)
{
  return(a+b+c);
}
/*End of funcOverload.cpp*/
```

Output

```
30
120
```

Listing 1.24 Function overloading

Just like ordinary functions, the definitions of overloaded functions are also put in librari
Moreover, the function prototypes are placed in header files.

The two function prototypes at the beginning of the program tell the compiler the t
different ways in which the 'add()' function can be called. When the compiler encount
the two distinct calls to the 'add()' function, it already has the prototypes to satisfy th
both. Thus, the compilation phase is completed successfully. During linking, the link
finds the two necessary definitions of the 'add()' function and, hence, links successfu
to create the executable file.

*The compiler decides which function is to be called based upon the number, type, a
sequence of parameters that are passed to the function call.* When the compiler encount
the first function call,

```
x=add(10,20);
```

it decides that the function that takes two integers as formal arguments is to be execut
Accordingly, the linker then searches for the definition of the 'add()' function wh
there are two integers as formal arguments.

Similarly, the second call to the 'add()' function

```
y=add(30,40,50);
```

is also handled by the compiler and the linker.

Note the importance of function prototyping. Since function prototyping is mandatory
C++, it is possible for the compiler to support function overloading properly. The compi
is able to not only restrict the number of ways in which a function can be called but al
support more than one way in which a function can be called. *Function overloading
possible because of the necessity to prototype functions.*

By itself, function overloading is of little use. Instead of giving exactly the same nam
for functions that perform similar tasks, it is always possible for us to give them simil
names. However, function overloading enables the C++ compiler to support anoth
feature, that is, function overriding (which in turn is not really a very useful thing
itself but forms the basis for dynamic polymorphism—one of the most striking featu
of C++ that promotes code reuse).

Function overloading is also known as *function polymorphism* because, just li
polymorphism in the real world where an entity exists in more than one form, the sam
function name carries different meanings.

Function polymorphism is static in nature because the function definition to be execut
is selected by the compiler during compile time itself. Thus, an overloaded function
said to exhibit *static polymorphism.*

1.10 Default Values for Formal Arguments of Functions

It is possible to specify default values for some or all of the formal arguments of a function. If no value is passed for an argument when the function is called, the default value specified for it is passed. If parameters are passed in the normal fashion for such an argument, the default value is ignored. An illustrative example follows.

```
/*Beginning of defaultArg.cpp*/
#include<iostream.h>
int add(int,int,int c=0);//third argument has default value

void main()
{
  int x,y;
  x=add(10,20,30);          //default value ignored
  y=add(40,50);             //default value taken for the
                            //third parameter
  cout<<x<<endl<<y<<endl;
}

int add(int a,int b,int c)
{
  return (a+b+c);
}
/*End of defaultArg.cpp*/
```

Output

```
60
90
```

Listing 1.25 Default values for function arguments

In the above listing, a default value—zero—has been specified for the third argument of the 'add()' function. In the absence of a value being passed to it, the compiler assigns the default value. If a value is passed to it, the compiler assigns the passed value. In the first call

```
x=add(10,20,30);
```

the values of 'a', 'b', and 'c' are 10, 20, and 30, respectively. But, in the second function call

```
y=add(40,50);
```

the values of 'a', 'b', and 'c' are 10, 20, and 0, respectively. The default value—zero—for the third parameter 'c' is taken. This explains the output of the above listing.

Default values can be assigned to more than one argument. The following program illustrates this.

```
/*Beginning of multDefaultArg.cpp*/
#include<iostream.h>
int add(int,int b=0,int c=0); //second and third argument
                              //have default values

void main()
{
  int x,y,z;
  x=add(10,20,30);            //all default values ignored
  y=add(40,50);               //default value taken for the
                              //third argument
  z=add(60);                  //default value taken for
                              //the second and the third
                              //arguments
  cout<<x<<endl<<y<<endl<<z<<endl;
}

int add(int a,int b,int c)
{
  return (a+b+c);
}
/*End of multDefaultArg.cpp*/
```

Output

```
60
90
60
```

Listing 1.26 Default vlaues for more than one argument

There is no need to provide names to the arguments taking default values in the function prototypes.

```
int add(int,int=0,int=0);
```

can be written instead of

```
int add(int,int b=0,int c=0);
```

Default values must be supplied starting from the rightmost argument. Before supplying default value to an argument, all arguments to its right must be given default values. Suppose you write

```
int add(int,int=0,int);
```

you are attempting to give a default value to the second argument from the right without specifying a default value for the argument on its right. The compiler will report an error that the default value is missing (for the third argument).

Default values must be specified in function prototypes alone. They should not be specified in the function definitions.

While compiling a function call, the compiler will definitely have its prototype. Its definition will probably be located after the function call. It might be in the same file, or it will be in a different file or library. Thus, to ensure a successful compilation of the function calls where values for arguments having default values have not been passed, the compiler must be aware of those default values. Hence, default values must be specified in the function prototype.

You must also remember that the function prototypes are placed in header files. These are included in both the library files that contain the function's definition as well as the client program files that contain calls to the functions. While compiling the library file that contains the function definition, the compiler will obviously read the function prototype before it reads the function definition. Suppose the function definition also contains default values for the arguments. Even if the same default values are supplied for the same arguments, the compiler will think that you are trying to supply two different default values for the same argument. This is obviously unacceptable because the default value can be only one in number. Thus, *default values must be specified in the function prototypes and should not be specified again in the function definitions.*

If default values are specified for the arguments of a function, the function behaves like an overloaded function and, therefore, should be overloaded with care; otherwise ambiguity errors might be caused. For example, if you prototype a function as follows:

```
int add(int,int,int=0);
int add(int,int);
```

This can confuse the compiler. If only two integers are passed as parameters to the function call, both these prototypes will match. The compiler will not be able to decide with which definition the function call has to be resolved. This will lead to an ambiguity error.

Default values can be given to arguments of any data type as follows:

```
double hra(double,double=0.3);
void print(char='a');
```

1.11 Inline Functions

Inline functions are used to increase the speed of execution of the executable files. C++ inserts calls to the normal functions and the inline functions in different ways in an executable.

The executable program that is created after compiling the various source codes and linking them consists of a set of machine language instructions. When a program is started, the operating system loads these instructions into the computer's memory. Thus each instruction has a particular memory address. The computer then goes through these instructions one by one. If there are any instructions to branch out or loop, the control skips over instructions and jumps backward or forward as needed. When a program reaches the function call instruction, it stores the memory address of the instruction immediately following the function call. It then jumps to the beginning of the function whose address it finds in the function call instruction itself, executes the function code, and jumps back to the instruction whose address it had saved earlier.

Obviously, an overhead is involved in

- making the control jump back and forth and
- storing the address of the instruction to which the control should jump after the function terminates.

The C++ inline function provides a solution to this problem. *An inline function is a function whose compiled code is 'in line' with the rest of the program.* That is, the compiler replaces the function call with the corresponding function code. With inline code, the program does not have to jump to another location to execute the code and then jump back. Inline functions, thus, run a little faster than regular functions.

However, there is a trade-off between memory and speed. If an inline function is called repeatedly, then multiple copies of the function definition appear in the code (see Diagrams 1.1 and 1.2). Thus, the executable program itself becomes so large that it occupies a lot of space in the computer's memory during run time. Consequently, the program runs slow instead of running fast. Thus, inline functions must be chosen with care.

For specifying an inline function, you must:

- prefix the definition of the function with the `inline` keyword and
- define the function before all functions that call it, that is, define it in the header file itself.

The following listing illustrates the inline technique with the inline 'cube()' function that cubes its argument. Note that the entire definition is in one line. That is not a necessary

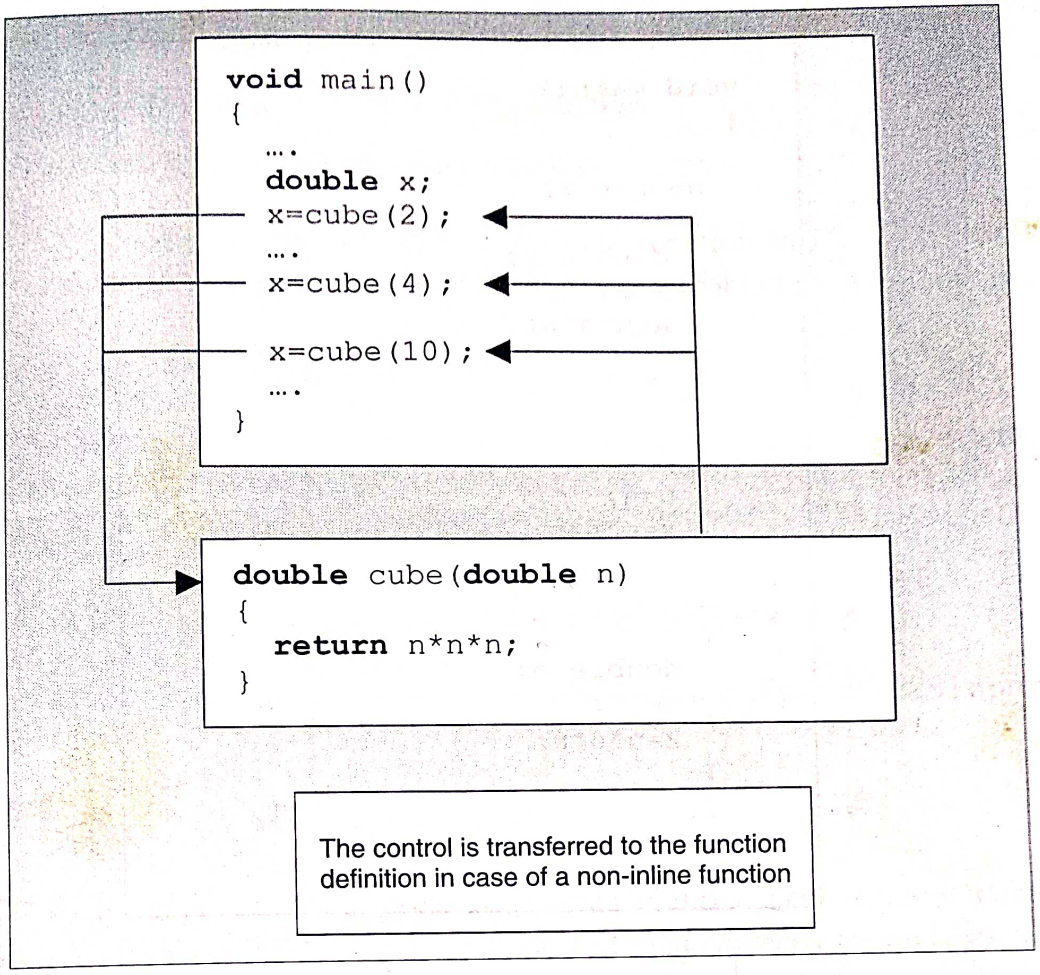

Diagram 1.1 Transfer of control in a non-inline function

condition. But if the definition of a function does not fit in one line, the function probably a poor candidate for an inline function!

```
/*Beginning of inline.cpp*/
#include<iostream.h>

inline double cube(double x) { return x*x*x; }

void main()
{
  double a,b;
  double c=13.0;
  a=cube(5.0);
  b=cube(4.5+7.5);
  cout<<a<<endl;
  cout<<b<<endl;
  cout<<cube(c++)<<endl;
  cout<<c<<endl;
}
/*End of inline.cpp*/
```

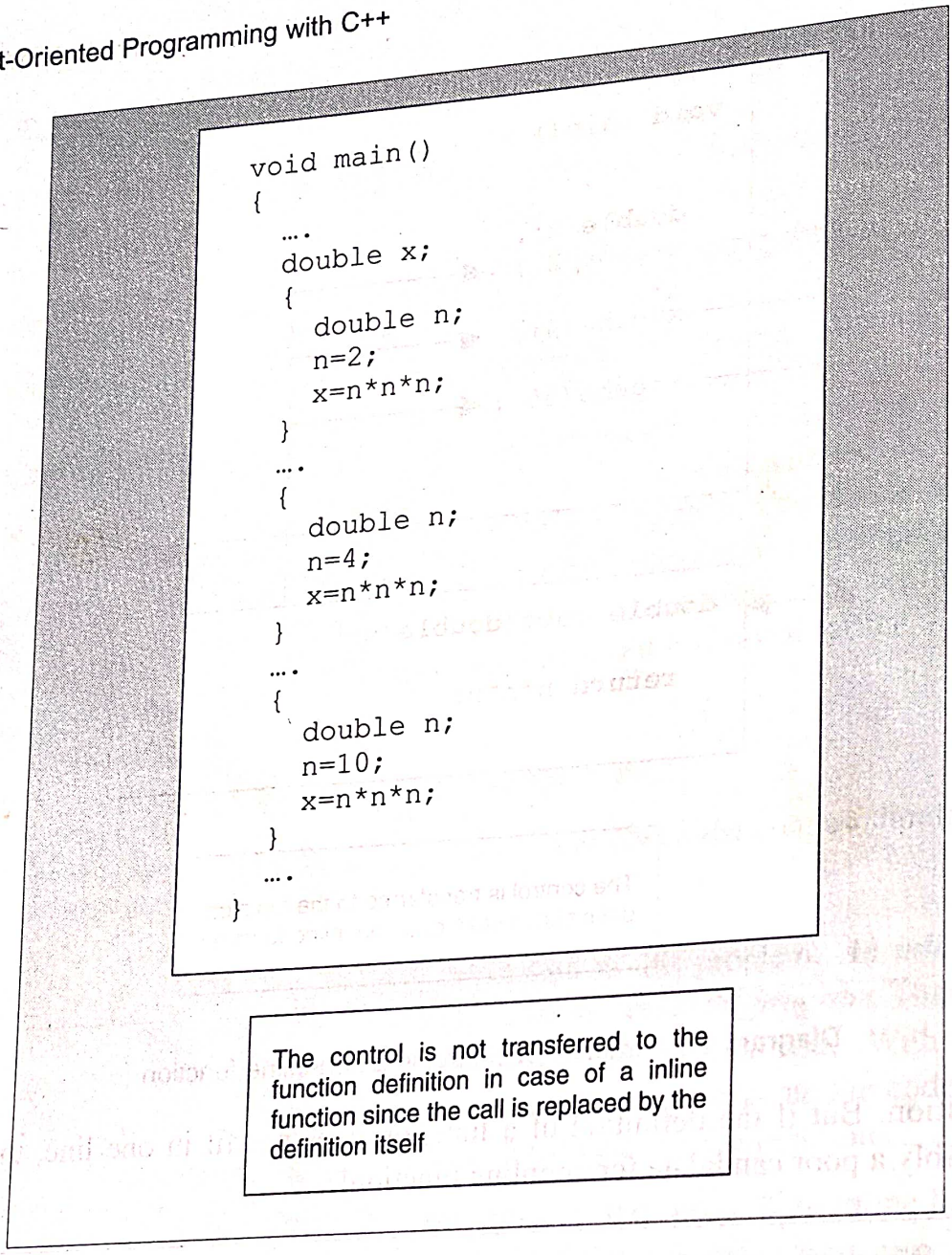

Diagram 1.2 Control does not get transferred in an inline function

Output

125
1728
2197
14

Listing 1.27 Inline functions

However, under some circumstances, the compiler, despite your indications, may no expand the function inline. Instead, it will issue a warning that the function could not b expanded inline and then compile all calls to such functions in the ordinary fashio Those conditions are:

- The function is recursive.
- There are looping constructs in the function.
- There are static variables in the function.

Let us briefly compare macros in C and inline function in C++. Macros are a poor predecessor to inline functions. For example, a macro for cubing a number is as follows:

```
#define CUBE(X) X*X*X
```

Here, a mere text substitution takes place with 'X' being replaced by the macro parameter.

```
a=CUBE(5.0);          //replaced by a=5.0*5.0*5.0;
b=CUBE(4.5+7.5);      //replaced by
                      //b=4.5+7.5*4.5+7.5*4.5+7.5;
c=CUBE(x++);          //replaced by c=x++*x++*x++;
```

Only the first statement works properly. An intelligent use of parentheses improves matters slightly.

```
#define CUBE(X) ((X)*(X)*(X))
```

Even now, 'CUBE(c++)' undesirably increments 'c' thrice. But the inline 'cube()' function evaluates 'c', passes the value to be cubed, and then correctly increments 'c' once.

It is advisable to use inline functions instead of macros.

Summary

Variables sometimes influence each other's values. A change in the value of one may necessitate a corresponding adjustment in the value of another. It is, therefore, necessary to pass these variables together in a single group to functions. Structures enable us to do this.

Structures are used to create new data types. This is a two-step process.

Step 1: Create the structure itself.

Step 2: Create associated functions that work upon variables of the structure.

While structures do fulfil the important need described above, they nevertheless have limitation They do not enable the library programmer to make variables of the structure that he/she has designe to be safe from unintentional modification by functions other than those defined by him/her. Moreove they do not guarantee a proper initialization of data members of structure variables.

Both of the above drawbacks are in direct contradiction with the characteristics possessed by rea world objects. A real-world object has not only a perfect interface to manipulate its internal pa

Exercises

1. Which programming needs do structures fulfill? Why does C language enable us to create structures?
2. What are the limitations of structures?
3. What is the procedure-oriented programming system?
4. What is the object-oriented programming system?
5. Which class is 'cout' an object of?
6. Which class is 'cin' an object of?
7. What benefits does a programmer get if the compiler forces him/her to prototype a function?
8. Why will an ambiguity error arise if a default value is given to an argument of an overloaded function?
9. Why should default values be given to function arguments in the function's prototype and not in the function's definition?
10. State true or false.
    (a) Structures enable a programmer to secure the data contained in structure variables from being changed by unauthorized functions.
    (b) The 'insertion operator' is used for outputting in C++.

(c) The 'extraction operator' is used for outputting in C++.
(d) A call to a function that returns by reference cannot be placed on the left of the assignme operator.
(e) An inline function cannot have a looping construct.

11. Think of some examples from your own experience in C programming where you felt the nee for structures. Do you see an opportunity for programming in OOPS in those examples?
12. Structures in C do not enable the library programmers to guarantee an initialization of dat Appreciate the implications of this limitation by taking the date structure as an example.
13. Calls to functions that return by reference can be put on the left-hand side of the assignme operator. Experiment and find out whether such calls can be chained. Consider the followin

```
f(a, b) = g(c, d) = x;
```

where 'f' and 'g' are functions that return by reference while 'a', 'b', 'c', 'd', and 'x' variables.