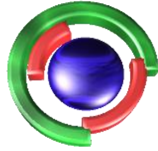
FUTURE VISION BIE

One Stop for All Study Materials & Lab Programs



By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Detailed Contents

Preface to the Second Edition iii

Preface to the First Edition vi

Brief Contents ix

1. Introduction to C++ 1
1.1 A Review of Structures *1*
1.1.1 The Need for Structures *1*
1.1.2 Creating a New Data Type Using Structures *4*
1.1.3 Using Structures in Application Programs *5*
1.2 Procedure-Oriented Programming Systems *5*
1.3 Object-Oriented Programming Systems *7*
1.4 Comparison of C++ with C *8*
1.5 Console Input/Output in C++ *9*
1.5.1 Console Output *9*
1.5.2 Console Input *12*
1.6 Variables in C++ *13*
1.7 Reference Variables in C++ *14*
1.8 Function Prototyping *19*
1.9 Function Overloading *21*
1.10 Default Values for Formal Arguments of Functions *23*
1.11 Inline Functions *25*

2. Classes and Objects 31
2.1 Introduction to Classes and Objects *31*
2.1.1 Private and Public Members *33*
2.1.2 Objects *36*
2.1.3 Scope Resolution Operator *37*
2.1.4 Creating Libraries Using the Scope Resolution Operator *38*
2.1.5 Using Classes in Application Programs *39*
2.1.6 `this` Pointer *40*
2.1.7 Data Abstraction *45*
2.1.8 Explicit Address Manipulation *47*
2.1.9 Arrow Operator *47*
2.1.10 Calling One Member Function from Another *48*

https://hemanthrajhemu.github.io

2.2 Member Functions and Member Data *49*
2.2.1 Overloaded Member Functions *49*
2.2.2 Default Values for Formal Arguments of Member Functions *51*
2.2.3 Inline Member Functions *52*
2.2.4 Constant Member Functions *52*
2.2.5 Mutable Data Members *54*
2.2.6 Friends *54*
2.2.7 Static Members *59*
2.3 Objects and Functions *65*
2.4 Objects and Arrays *66*
2.4.1 Arrays of Objects *67*
2.4.2 Arrays Inside Objects *67*
2.5 Namespaces *68*
2.6 Nested Inner Classes *71*

3. Dynamic Memory Management **78**
3.1 Introduction *78*
3.2 Dynamic Memory Allocation *79*
3.3 Dynamic Memory Deallocation *84*
3.4 `set_new_handler()` function *88*

4. Constructors and Destructors **92**
4.1 Constructors *92*
4.1.1 Zero-argument Constructor *94*
4.1.2 Parameterized Constructors *97*
4.1.3 Explicit Constructors *103*
4.1.4 Copy Constructor *105*
4.2 Destructors *109*
4.3 Philosophy of OOPS *112*

5. Inheritance **117**
5.1 Introduction *117*
5.1.1 Effects of Inheritance *118*
5.1.2 Benefits of Inheritance *120*
5.1.3 Inheritance in Actual Practice *120*
5.1.4 Base Class and Derived Class Objects *121*
5.1.5 Accessing Members of the Base Class in the Derived Class *121*
5.2 Base Class and Derived Class Pointers *122*
5.3 Function Overriding *127*
5.4 Base Class Initialization *129*
5.5 Protected Access Specifier *132*
5.6 Deriving by Different Access Specifiers *133*
5.6.1 Deriving by the Public Access Specifier *133*
5.6.2 Deriving by the Protected Access Specifier *135*
5.6.3 Deriving by the Private Access Specifier *136*
5.7 Different Kinds of Inheritance *139*
5.7.1 Multiple Inheritance *139*
5.7.2 Ambiguities in Multiple Inheritance *141*

https://hemanthrajhemu.github.io

Copyrighted Materials
Copyright © 2012 Oxford University Press Retrieved from www.knovel.com

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.

1.1.1 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
```

https://hemanthrajhemu.github.io

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)!

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

```
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!
```

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
```

https://hemanthrajhemu.github.io

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.

Listing 1.2 The need for structures

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

'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

https://hemanthrajhemu.github.io

1.1.2 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, as shown in Listing 1.3.

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

```
/*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*/
```

Step 2: As shown in Listing 1.4, put the definition of the associated functions in a source code and create a library.

Listing 1.4 Defining the associated functions of a structure

```
/*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*/
```

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.

https://hemanthrajhemu.github.io

1.1.3 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: As shown in Listing 1.5, embed calls to the associated functions by passing these variables in the source code.

Listing 1.5 Using a structure in an application program

```
/*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*/
```

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 Systems

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.

https://hemanthrajhemu.github.io

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.

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 focused on the associated library functions.

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

https://hemanthrajhemu.github.io

1.3 Object-Oriented Programming Systems

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 behaviour of the LCD projector and the desired behaviour of data structure's variables. In OOPS, with the help of a new programming construct and new keywords, associated functions of the data 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.

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

https://hemanthrajhemu.github.io

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 behaviour. 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 header file given in Listing 1.6 shows how the structure Date, which has been our running example so far, can be rewritten in C++.

Listing 1.6 Redefining the Date structure 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*/
```

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.

https://hemanthrajhemu.github.io

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.

1.5 Console Input/Output in C++

This section discusses console input and output in C++.

1.5.1 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 (see Listing 1.7).

Listing 1.7 Outputting in C++

```
/*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

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 **c**onsole **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.

https://hemanthrajhemu.github.io

Another object endl allows us to insert a new line into the output stream. Listing 1.8 illustrates this.

Listing 1.8 Inserting new line by 'endl'

```
/*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

One striking feature of the insertion operator is that it works equally well with values of all fundamental types as its right-hand operand. It does not need the format specifiers that are needed in the printf() family of functions. Listing 1.9 exemplifies this.

Listing 1.9 Outputting data with the insertion operator

```
/*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*/
```

https://hemanthrajhemu.github.io

Output
10
x
2.3
3.14159
Hello World

Just like the arithmetic addition operator, it is possible to cascade the `insertion` operator. Listing 1.10 is a case in point.

Listing 1.10 Cascading the `insertion` operator

```
/*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

It is needless to say that we can pass constants instead of variables as operands to the `insertion` operator, as shown in Listing 1.11.

Listing 1.11 Outputting constants using the `insertion` operator

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

Ouput
10
Hello World
3.4

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

https://hemanthrajhemu.github.io

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.

1.5.2 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.

Listing 1.12 Inputting in C++

```
/*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

The third statement in the `main()` function of 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 **c**onsole **in**put 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. Listing 1.13 exemplifies this.

Listing 1.13 Inputting data with the extraction operator

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

https://hemanthrajhemu.github.io

```
   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

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

Listing 1.14 Cascading the `extraction` operator

```
/*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

It was mentioned in the beginning of this section that `cin` is an object that is associated with the console. Hence, if it is the left-hand side operand of the `extraction` operator, the variable on the right gets its value from the keyboard. 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 `extraction` operator. In 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 its very beginning. For example, see Listing 1.15.

https://hemanthrajhemu.github.io

Listing 1.15 Declaring variables in C++

```
#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

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.

https://hemanthrajhemu.github.io

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. Listing 1.16 illustrates this.

Listing 1.16 Reference variables

```
/*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

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. Listing 1.17 illustrates this.

Listing 1.17 Reading the value of a reference variable

```
/*Beginning of reference02.cpp*/
#include<iostream.h>
void main()
{
```

https://hemanthrajhemu.github.io

```
   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

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. Listing 1.18 is an illustrative example.

Listing 1.18 Passing by reference

```
/*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

Functions can return by reference also. See Listing 1.19.

Listing 1.19 Returning by reference

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

https://hemanthrajhemu.github.io

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

Output
10
–1

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'. Listing 1.19 can be shortened as illustrated in Listing 1.20.

Listing 1.20 Returning by reference

```
/*Beginning of reference05.cpp*/
#include<iostream.h>
int & larger(int &, int &);
int main()
{
   int x,y;
   x=10;
   y=20;
   larger(x,y)=-1;
   cout<<x<<endl<<y<<endl;
}
int & larger(int & a, int & b)
{
   if(a>b) //return a reference to the larger parameter
      return a;
   else
      return b;
}
/*End of reference05.cpp*/
```

Output
10
–1

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

https://hemanthrajhemu.github.io

- 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, see Listing 1.21.

Listing 1.21 Returning the reference of a local variable

```
/*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*/
```

https://hemanthrajhemu.github.io

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, see Listing 1.22.

Listing 1.22 Function prototyping

```
/*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

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

https://hemanthrajhemu.github.io

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

Let us discuss 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 Listing 1.23 shows.

Listing 1.23 Absence of function prototype produces weird results

```
/*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

A compiler that does not enforce prototyping will definitely compile the above program. But then it will have no way of knowing what type of value the `test()` function returns.

https://hemanthrajhemu.github.io

Therefore, erroneous results will be obtained during run time as the output of Listing 1.23 clearly shows.

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

What happens if the function prototype and the function call do not match? Such a situation cannot arise. Both the function prototype and the function definition are created by the same person, that is, the library programmer. The library programmer puts the function's prototype in a header file. He/she provides the function's definition in a library. The application programmer includes the header file in his/her application program file in which the function is called. He/she creates an object file from this application program 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 two parameters. If the program fails to provide such parameters, the prototype enables the compiler to detect the error. A compiler that does not enforce function prototyping will compile a function call in which an incorrect number and/or type of parameters have been passed. Run-time errors will arise as in the foregoing case.

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

Nevertheless, can the same benefits not be realized without prototyping? Is it not possible for the compiler to simply scan the rest of the source code and find out how the function 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 line 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

https://hemanthrajhemu.github.io

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.

Listing 1.24 Function overloading

```
/*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

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

The two function prototypes at the beginning of the program tell the compiler the two different ways in which the `add()` function can be called. When the compiler encounters the two distinct calls to the `add()` function, it already has the prototypes to satisfy them both. Thus, the compilation phase is completed successfully. During linking, the linker finds the two necessary definitions of the `add()` function and, hence, links successfully to create the executable file.

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

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

it decides that the function that takes two integers as formal arguments is to be executed. Accordingly, the linker then searches for the definition of the `add()` function where 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.

https://hemanthrajhemu.github.io

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

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

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

Function polymorphism is static in nature because the function definition to be executed is selected by the compiler during compile time itself. Thus, an overloaded function is 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. Listing 1.25 is an illustrative example.

Listing 1.25 Default values for function arguments

```
/*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

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

https://hemanthrajhemu.github.io

```
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. Listing 1.26 illustrates this.

Listing 1.26 Default values for more than one argument

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

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).

https://hemanthrajhemu.github.io

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 and 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

https://hemanthrajhemu.github.io

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

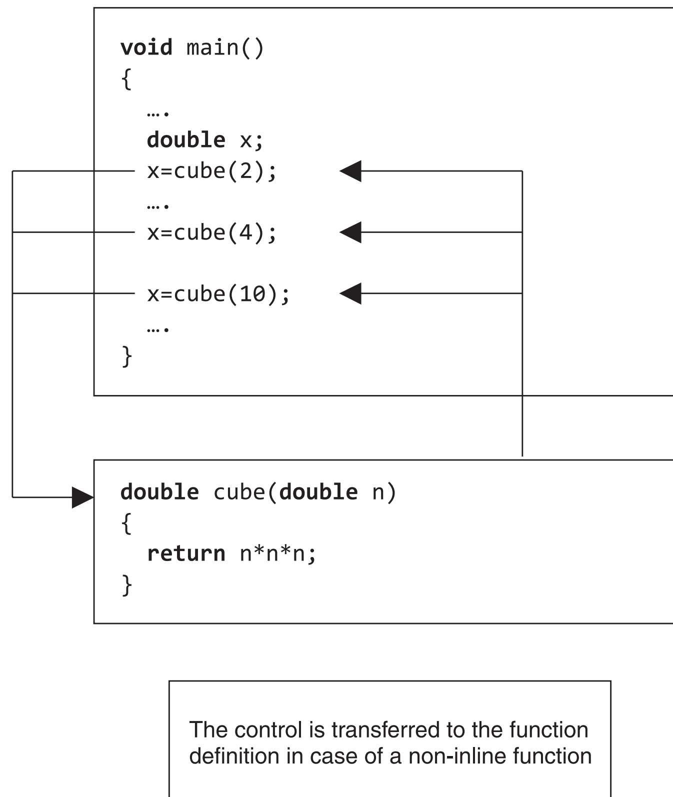


Figure 1.1 Transfer of control in a non-inline function

https://hemanthrajhemu.github.io

```
void main()
{
  ….
  double x;
  {
    double n;
    n=2;
    x=n*n*n;
  }
  ….
  {
    double n;
    n=4;
    x=n*n*n;
  }
  ….
  {
    double n;
    n=10;
    x=n*n*n;
  }
  ….
}
```

The control is not transferred to the function definition in case of an inline function since the call is replaced by the definition itself

Figure 1.2 Control does not get transferred in an inline function

Listing 1.27 Inline functions

```
/*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*/
```

https://hemanthrajhemu.github.io

Output
125
1728
2197
14

However, under some circumstances, the compiler, despite your indications, may not expand the function inline. Instead, it will issue a warning that the function could not be expanded inline and then compile all calls to such functions in the ordinary fashion. 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 limitations. They do not enable the library programmer to make variables of the structure that he/she has designed to be safe from unintentional modification by functions other than those defined by him/her. Moreover, 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 real-world objects. A real-world object has not only a perfect interface to manipulate its internal parts but also exclusive rights to do so. Consequently, a real-world object never reaches an invalid state during its lifetime. When we start operating a real-world object, it automatically assumes a valid state. In object-oriented programming systems (OOPS), we can incorporate these features of real-world objects into structure variables.

https://hemanthrajhemu.github.io

Inheritance allows a structure to inherit both data and functions of an existing structure. Polymorphism allows different functions to have the same name. It is of two types: static and dynamic.

Console output is achieved in C++ with the help of `insertion` operator and the `cout` object. Console input is achieved in C++ with the help of `extraction` operator and the `cin` object.

In C++, variables can be defined anywhere in a function. A reference variable shares the same memory location as the one of which it is a reference. Therefore, any change in its value automatically changes the value of the variable with which it is sharing memory. Calls to functions that return by reference can be placed on the left of the `assignment` operator.

Function prototyping is necessary in C++. Functions can be overloaded. Functions with different signatures can have the same name. A function argument can be given a default value so that if no value is passed for it in the function call, the default value is assigned to it. If a function is declared inline, its definition replaces its call, thus, speeding up the execution of the resultant executable.

Key Terms

creating new data types using structures
lack of data security in structures
no guaranteed initialization of data in structures
procedure-oriented programming system
object-oriented programming system
data security in classes
guaranteed initialization of data in classes
inheritance
polymorphism
console input/output in C++
- `cout`
- `ostream_withassign` class
- `insertion` operator
- `cin`
- `istream_withassign` class
- `extraction` operator
- `iostream.h` header file
- `endl`

reference variable
- passing by reference
- returning by reference

importance of function prototyping
function overloading
default values for function arguments
inline functions

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 `assignment` 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 need for structures.

https://hemanthrajhemu.github.io

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 data. 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 `assignment` operator. Experiment and find out whether such calls can be chained. Consider the following:

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

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

https://hemanthrajhemu.github.io

Copyrighted Materials
Copyright © 2012 Oxford University Press Retrieved from www.knovel.com

2 Classes and Objects

OVERVIEW

The previous chapter refreshed the reader's knowledge of the structure construct provided by C language—its use and usage. It also dealt with a critical analysis of structures along with their pitfalls and limitations. The reader was made aware of a strong need for data security and for a guaranteed initialization of data that structures do not provide.

This chapter is a logical continuation to the previous one. It begins with a thorough explanation of the class construct of C++ and the ways by which it fulfils the above-mentioned needs. Superiority of the class construct of C++ over the structure construct of C language is emphasized in this chapter.

This chapter also deals with how classes enable the library programmer to provide exclusive rights to the associated functions.

A description of various types and features of member functions and member data finds a prominent place in this chapter. This description covers:

- Overloaded member functions
- Default values for the arguments of member functions
- Inline member functions
- Constant member functions
- Mutable data members
- Friend functions and friend classes
- Static members

A section in this chapter is devoted to namespaces. They enable the C++ programmer to prevent pollution of the global namespace that leads to name clashes.

Example code to tackle arrays of objects and arrays inside objects form the penultimate portion of this chapter.

The chapter ends with an essay on nested classes—their need and use.

2.1 Introduction to Classes and Objects

Classes are to C++ what structures are to C. Both provide the library programmer a means to create new data types.

Let us briefly recapitulate the issues faced while programming in C described in the previous chapter. In C, the library programmer creates structures. He/she also provides a set of tested bug-free functions that correctly manipulate the data members of structure variables.

The `Date` structure and its accompanying functions may be perfect. However, there is absolutely no guarantee that the client programs will use only these functions to manipulate the members of variables of the structure. See Listing 2.1.

https://hemanthrajhemu.github.io

Listing 2.1 Undesirable manipulation of structures not prevented in C

```
struct Date d1;
setDate(&d1);    //assign system date to d1.
printf("%d",d1.month);
d1.month = 13;   //undesirable but unpreventable!!
```

The bug arising out of the last line of the `main()` function above is easily detected even by a visual inspection. Nevertheless, the same will certainly not be the case if the code is around 25,000 lines long. Lines similar to the last line of the `main()` function above may be scattered all over the code. Thus, they will be difficult to hunt down.

Notice that the absence of a facility to bind the data and the code that can have the exclusive rights to manipulate the data can lead to difficult-to-detect run-time bugs. C does not provide the library programmer with the facilities to encapsulate data, to hide data, and to abstract data.

The C++ compiler provides a solution to this problem. Structures (the `struct` keyword) have been redefined to allow member functions also. Listing 2.2 illustrates this.

Listing 2.2 C++ allows member functions in structures

```
/*Beginning of structDistance01.cpp*/
#include<iostream.h>

struct Distance
{
   int iFeet;
   float fInches;
   void setFeet(int x)
   {
      iFeet=x;
   }
   int getFeet()
   {
      return iFeet;
   }
   void setInches(float y)
   {
      fInches=y;
   }
   float getInches()
   {
      return fInches;
   }
};

void main()
{
   Distance d1,d2;
   d1.setFeet(2);
   d1.setInches(2.2);
   d2.setFeet(3);
   d2.setInches(3.3);
   cout<<d1.getFeet()<<" "<<d1.getInches()<<endl;
   cout<<d2.getFeet()<<" "<<d2.getInches()<<endl;
```

https://hemanthrajhemu.github.io

```
}
/*End of structDistance01.cpp*/
```

Output
2 2.2
3 3.3

First, we must notice that functions have also been defined within the scope of the structure definition. This means that not only the member data of the structure can be accessed through the variables of the structures but also the member functions can be invoked. The `struct` keyword has actually been redefined in C++. This latter point is illustrated by the `main()` function in Listing 2.2 above. We must make careful note of the way variables of the structure have been declared and how the member functions have been invoked.

Member functions are invoked in much the same way as member data are accessed, that is, by using the variable-to-member access operator. In a member function, one can refer directly to members of the object for which the member function is invoked. For example, as a result of the second line of the `main()` function in Listing 2.2, it is `d1.iFeet` that gets the value of 2. On the other hand, it is `d2.iFeet` that gets the value of 3 when the fourth line is invoked. This is explained in the section on the `this` pointer that follows shortly.

Each structure variable contains a separate copy of the member data within itself. However, only one copy of the member function exists. Again, the section on the `this` pointer explains this.

However, in the above example, note that the member data of structure variables can still be accessed directly. The following line of code illustrates this.

```
d1.iFeet=2; //legal!!
```

2.1.1 Private and Public Members

What is the advantage of having member functions also in structures? We have put together the data and functions that work upon the data but we have not been able to give exclusive rights to these functions to work upon the data. Problems in code debugging can still arise as before. Specifying member functions as public but member data as private obtains the advantage. The syntax for this is illustrated by Listing 2.3.

Listing 2.3 Making members of structures private

```
/*Beginning of structDistance02.cpp*/
#include<iostream.h>
struct Distance
{
   private:
      int iFeet;
      float fInches;
   public:
      void setFeet(int x)
{
   iFeet=x;     //LEGAL: private member accessed by
                //member function
}
int getFeet()
```

https://hemanthrajhemu.github.io

```
{
  return iFeet;
}
void setInches(float y)
{
  fInches=y;
}
float getInches()
  {
  return fInches;
  }
};

void main()
{
  Distance d1,d2;
  d1.setFeet(2);
  d1.setInches(2.2);
  d2.setFeet(3);
  d2.setInches(3.3);
  d1.iFeet++;                    //ERROR!!: private member accessed by
                                 //non-member function
  cout<<d1.getFeet()<<" "<<d1.getInches()<<endl;
  cout<<d2.getFeet()<<" "<<d2.getInches()<<endl;
}
/*End of structDistance02.cpp*/
```

First, let us have a close look at the modified definition of the structure `Distance`. Two new keywords, private and public have been introduced in the definition of the structure. Their presence in the foregoing example tells the compiler that `iFeet` and `fInches` are private data members of variables of the structure `Distance` and the member functions are public. Thus, values of `iFeet` and `fInches` of each variable of the structure `Distance` can be accessed/modified only through member functions of the structure and not by any non-member function in the program (again note that it is the `iFeet` and `fInches` of the *invoking object* that are accessed/modified by the member functions). Any attempt to violate this restriction is prevented by the compiler because that is how the C++ compiler recognizes the private keyword. Since the member functions are public, they can be invoked from any part of the program.

As we can observe from Listing 2.3, the compiler refuses to compile the line in which a private member of a structure variable is accessed from a non-member function (the `main()` function in Listing 2.3).

The keywords `private` and `public` are also known as access modifiers or access specifiers because they control the access to the members of structures.

C++ introduces a new keyword `class` as a substitute for the keyword `struct`. *In a structure, members are public by default*. See the definition in Listing 2.4.

Listing 2.4 Structure members are public by default

```
struct Distance
{
  private:
    int iFeet;
    float fInches;
```

https://hemanthrajhemu.github.io

```
    public:
       void setFeet(int x)
       {
          iFeet=x;
       }
       int getFeet()
       {
          return iFeet;
       }
       void setInches(float y)
       {
          fInches=y;
       }
       float getInches()
       {
          return fInches;
       }
};
```

can also be written as

```
struct Distance
{
       void setFeet(int x)          //public by default
       {
          iFeet=x;
       }
       int getFeet()                //public by default
       {
          return iFeet;
       }
       void setInches(float y)      //public by default
       {
          fInches=y;
       }
       float getInches()            //public by default
       {
          return fInches;
       }
    private:
       int iFeet;
       float fInches;
};
```

In Listing 2.4, the member functions have not been placed under any access modifier. Therefore, they are public members by default.

On the other hand, *class members are private by default*. This is the only difference between the `class` keyword and the `struct` keyword.

Thus, the structure `Distance` can be redefined by using the `class` keyword as shown in Listing 2.5.

Listing 2.5 Class members are private by default

```
class Distance
{
       int iFeet;                   //private by default
       float fInches;               //private by default
```

https://hemanthrajhemu.github.io

```
    public:
      void setFeet(int x)
      {
        iFeet=x;
      }
      int getFeet()
      {
        return iFeet;
      }
      void setInches(float y)
      {
        fInches=y;
      }
      float getInches()
      {
        return fInches;
      }
};
```

The `struct` keyword has been retained to maintain backward compatibility with C language. A header file created in C might contain the definition of a structure, and structures in C will have member data only. A C++ compiler will easily compile a source code that has included the above header file since the new definition of the `struct` keyword allows, not mandates, the inclusion of member functions in structures.

Functions in a C language source code access member data of structures. A C++ compiler will easily compile such a source code since the C++ compiler treats members of structures as public members by default.

2.1.2 Objects

Variables of classes are known as objects.

An object of a class occupies the same amount of memory as a variable of a structure that has the same data members. This is illustrated by Listing 2.6.

Listing 2.6 Size of a class object is equal to that of a structure variable with identical data members

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

struct A
{
  char a;
  int b;
  float c;
};

class B                                //a class with the same data members
{
  char a;
  int b;
  float c;
};

void main()
{
```

https://hemanthrajhemu.github.io

```
   cout<<sizeof(A)<<endl<<sizeof(B)<<endl;
}
/*End of objectSize.cpp*/
```

Output
9
9

Introducing member functions does not influence the size of objects. The reason for this will become apparent when we study the `this` pointer. Moreover, making data members private or public does not influence the size of objects. The access modifiers merely control the accessibility of the members.

2.1.3 Scope Resolution Operator

It is possible and usually necessary for the library programmer to define the member functions *outside* their respective classes. The scope resolution operator makes this possible. Listing 2.7 illustrates the use of the scope resolution operator (::).

Listing 2.7 The scope resolution operator

```
/*Beginning of scopeResolution.cpp*/
class Distance
{
      int iFeet;
      float fInches;
   public:
      void setFeet(int);              //prototype only
      int getFeet();                  //prototype only
      void setInches(float);          //prototype only
      float getInches();              //prototype only
};

void Distance::setFeet(int x)         //definition
{
   iFeet=x;
}

int Distance::getFeet()               //definition
{
   return iFeet;
}

void Distance::setInches(float y)     //definition
{
   fInches=y;
}

float Distance::getInches()           //definition
{
   return fInches;
}
/*End of scopeResolution.cpp*/
```

We can observe that the member functions have been only prototyped within the class; they have been *defined* outside. The scope resolution operator signifies the class to which they

https://hemanthrajhemu.github.io

belong. The class name is specified on the left-hand side of the scope resolution operator. The name of the function being defined is on the right-hand side.

2.1.4 Creating Libraries Using the Scope Resolution Operator

As in C language, creating a new data type in C++ using classes is also a three-step process that is executed by the library programmer.

Step 1: Place the class definition in a header file.

```
/*Beginning of Distance.h*/
/*Header file containing the definition of the Distance class*/

class Distance
{
     int iFeet;
     float fInches;
   public:
     void setFeet(int);               //prototype only
     int getFeet();                   //prototype only
     void setInches(float);           //prototype only
     float getInches();               //prototype only
};
/*End of Distance.h*/
```

Step 2: Place the definitions of the member functions in a C++ source file (the library source code). A file that contains definitions of the member functions of a class is known as the implementation file of that class. Compile this implementation file and put in a library.

```
/*Beginning of Distlib.cpp*/
/*Implementation file for the class Distance*/
#include"Distance.h"

void Distance::setFeet(int x)         //definition
{
   iFeet=x;
}

int Distance::getFeet()               //definition
{
   return iFeet;
}

void Distance::setInches(float y)     //definition
{
   fInches=y;
}

float Distance::getInches()           //definition
{
   return fInches;
}
/*End of Distlib.cpp*/
```

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

https://hemanthrajhemu.github.io

2.1.5 Using Classes in Application Programs

The steps followed by programmers for using this new data type are:
Step 1: Include the header file provided by the library programmer in their source code.

```
/*Beginning of Distmain.cpp*/
#include"Distance.h"

void main()
{
   . . . .
   . . . .
}
/*End of Distmain.cpp*/
```

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

```
/*Beginning of Distmain.cpp*/
#include"Distance.h"

void main()
{
   Distance d1,d2;
   . . . .
   . . . .
}
/*End of Distmain.cpp*/
```

Step 3: Embed calls to the associated functions by passing these variables in their source code. See Listing 2.8.

Listing 2.8 Using classes in application programs

```
/*Beginning of Distmain.cpp*/
/*A sample driver program for creating and using objects of the class Dis-
tance*/
#include<iostream.h>
#include"Distance.h"

void main()
{
   Distance d1,d2;
   d1.setFeet(2);
   d1.setInches(2.2);
   d2.setFeet(3);
   d2.setInches(3.3);
   cout<<d1.getFeet()<<" "<<d1.getInches()<<endl;
   cout<<d2.getFeet()<<" "<<d2.getInches()<<endl;
}
/*End of Distmain.cpp*/
```

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.

Output of Listing 2.8
2 2.2
3 3.3

https://hemanthrajhemu.github.io

Implementation files are compiled and converted into static and dynamic libraries in the usual manner.

Again, we notice that there is no obvious connection between the member data being accessed within the member function and the object that is invoking the function.

2.1.6 `this` Pointer

The facility to create and call member functions of class objects is provided by the C++ compiler. You have already seen how this facility is to be used. However, how does the compiler support this facility? The compiler does this by using a unique pointer known as the `this` pointer. A thorough understanding of the `this` pointer is vital for understanding many concepts in C++.

The `this` pointer is always a constant pointer. The `this` pointer always points at the object with respect to which the function was called. An explanation that follows shortly explains why and how it functions.

After the compiler has ascertained that no attempt has been made to access the private members of an object by non-member functions, it converts the C++ code into an ordinary C language code as follows:

1. It converts the class into a structure with only data members as follows.

   Before

```
class Distance
{
      int iFeet;
      float fInches;
   public:
      void setFeet(int);            //prototype only
      int getFeet();                //prototype only
      void setInches(float);        //prototype only
      float getInches();            //prototype only
};
```

   After

```
struct Distance
{
   int iFeet;
   float fInches;
};
```

2. It puts a declaration of the `this` pointer as a leading formal argument in the prototypes of all member functions as follows.

   Before

```
void setFeet(int);
```

   After

```
void setFeet(Distance * const, int);
```

   Before

```
int getFeet();
```

https://hemanthrajhemu.github.io

After

```
int getFeet(Distance * const);
```

Before

```
void setInches(float);
```

After

```
void setInches(Distance * const, float);
```

Before

```
float getInches();
```

After

```
float getInches(Distance * const);
```

3. It puts the definition of the `this` pointer as a leading formal argument in the definitions of all member functions as follows. It also modifies all the statements to access object members by accessing them through the `this` pointer using the pointer-to-member access operator (->).

Before

```
void Distance::setFeet(int x)
{
   iFeet=x;
}
```

After

```
void setFeet(Distance * const this, int x)
{
   this->iFeet=x;
}
```

Before

```
int Distance::getFeet()
{
   return iFeet;
}
```

After

```
int getFeet(Distance * const this)
{
   return this->iFeet;
}
```

Before

```
void Distance::setInches(float y)
{
   fInches=y;
}
```

After

```
void setInches(Distance * const this, float y)
```

https://hemanthrajhemu.github.io

```
{
   this->fInches=y;
}
```

Before

```
float Distance::getInches()
{
   return fInches;
}
```

After

```
float getInches(Distance * const this)
{
   return this->fInches;
}
```

We must understand how the scope resolution operator works. The scope resolution operator is also an operator. Just like any other operator, it operates upon its operands. The scope resolution operator is a *binary* operator, that is, it takes two operands. The operand on its left is the name of a pre-defined class. On its right is a member function of that class. Based upon this information, the scope resolution operator inserts a constant operator of the correct type as a leading formal argument to the function on its right. For example, if the class name is `Distance`, as in the above case, the compiler inserts a pointer of type `Distance * const` as a leading formal argument to the function on its right.

4. It passes the address of invoking object as a leading parameter to each call to the member functions as follows.

Before

```
d1.setFeet(1);
```

After

```
setFeet(&d1,1);
```

Before

```
d1.setInches(1.1);
```

After

```
setInches(&d1,1.1);
```

Before

```
cout<<d1.getFeet()<<endl;
```

After

```
cout<<getFeet(&d1)<<endl;
```

Before

```
cout<<d1.getInches()<<endl;
```

https://hemanthrajhemu.github.io

After

```
cout<<getInches(&d1)<<endl;
```

In the case of C++, the dot operator's definition has been extended. It not only takes data members as in C but also member functions as its right-hand side operand. If the operand on its right is a data member, then the dot operator behaves just like it does in C language. However, if the operand on its right is a member function, then the dot operator causes the address of the object on its left to be passed as an implicit leading parameter to the function call.

Clearly, members of the *invoking object* are referred to when they are accessed without any qualifiers in member functions. It should also be obvious that multiple copies of member data exist (one inside each object) but only one copy exists for each member function.

It is evident that the `this` pointer should continue to point at the same object—the object with respect to which the member function has been called—throughout its lifetime. For this reason, the compiler creates it as a constant pointer.

The accessibility of the implicit object is the same as that of the other objects passed as parameters in the function call and the local objects inside that function. Listing 2.9 illustrates this. A new function—`add()`—has been added to the existing definition of the `Distance` class.

Listing 2.9 Accessing data members of local objects inside member functions and of objects that are passed as parameters

```
/*Beginning of Distance.h*/
class Distance
{
  /*
    rest of the class Distance
  */
    Distance add(Distance);
};
/*End of Distance.h*/

/*Beginning of Distlib.cpp*/
#include"Distance.h"

Distance Distance::add(Distance dd)
{
  Distance temp;
  temp.iFeet=iFeet+dd.iFeet;          //legal to access both
                                      //temp.iFeet and
                                      //dd.iFeet
  temp.fInches=fInches+dd.fInches;    //ditto
  return temp;
}

/*
  definitions of the rest of the functions of class
  Distance
*/
/*End of Distlib.cpp*/

/*Beginning of Distmain.cpp*/
#include<iostream.h>
#include"Distance.h"
```

https://hemanthrajhemu.github.io

```
void main()
{
   Distance d1,d2,d3;
   d1.setFeet(1);
   d1.setInches(1.1);
   d2.setFeet(2);
   d2.setInches(2.2);
   d3=d1.add(d2);
   cout<<d3.getFeet()<<"'-"<<d3.getInches()<<"''\n";
}
/*End of Distmain.cpp*/
```

Output

3'-3.3'

The definition of `Distance :: add()` function, after the previously described conversion by the compiler is carried out, will appear as follows.

```
Distance add(Distance * const this, Distance dd)
{
   Distance temp;
   temp.iFeet=this->iFeet+dd.iFeet;
   temp.fInches=this->fInches+dd.fInches;
   return temp;
}
```

When this function is called from the `main()` function with respect to 'd1', the `this` pointer points at 'd1'. Thus, it is the private data member of 'd1' that is being accessed in the second and third lines of the `add()` function.

So, now we can

- Declare a class
- Define member data and member functions
- Make members private and public
- Declare objects and call member functions with respect to objects

What advantages does all this lead to? The advantage that library programmers can now derive from this arrangement is epitomized in the following observation:

An executable file will not be created from a source code in which private data members of an object have been accessed by non-member functions.

Once again, the importance of compile-time errors over run-time errors is emphasized. Suppose, an if block exists in a function that is not intended by the library programmer to access the data members of a structure. This if block contains a bug (say 'd1.month' has been assigned the value of 13, where 'd1' is a variable of the structure 'date').

A pure C compiler would not recognize this statement as an invalid access. During testing, the if condition of this if block might never become true. The bug would remain undetected; the executable will get created with bugs. Thus, *creating bug-free executables is difficult and unreliable in C*. This is due to the absence of language constructs that enforce data security.

On the other hand, a C++ compiler that also detects invalid access of private data members would immediately throw an error during compile time itself and prevent the creation of the executable. Thus, *creating bug-free executables is easier and more reliable in C++ than in C*. This is due to the presence of language constructs that enforce data security.

https://hemanthrajhemu.github.io

2.1.7 Data Abstraction

The class construct provides facilities to implement data abstraction. Data abstraction is an important concept and should be understood properly. Let us take up the example of the LCD projector from the previous chapter. It has member data (light and fan) as well as member functions (switches that operate the light and the fan). This real-world object hides its internal operations from the outside world. It, thus, obviates the need for the user to know the possible pitfalls that might be encountered during its operation. During its operation, the LCD projector never reaches an invalid state. Moreover, the LCD projector does not start in an invalid state.

Data abstraction is a virtue by which an object hides its internal operations from the rest of the program. It makes it unnecessary for the client programs to know how the data is internally arranged in the object. Thus, it obviates the need for the client programs to write precautionary code upon creating and while using objects.

Now, in order to understand this concept, let us take an example in C++. The library programmer, who has designed the `Distance` class, wants to ensure that the `fInches` portion of an object of the class should never exceed 12. If a value larger than 12 is specified by an application programmer while calling the `Distance::setInches()` function, the logic incorporated within the definition of the function should automatically increment the value of `iFeet` and decrement the value of `fInches` by suitable amounts. A modified definition of the `Distance::setInches()` function is as follows.

```
void Distance::setInches(float y)
{
   fInches=y;
   if(fInches>=12)
   {
      iFeet+=fInches/12;
      fInches-=((int)fInches/12)*12;
   }
}
```

Here, we notice that an application programmer need not send values less than 12 while calling the `Distance::setInches()` function. The default logic within the `Distance::setInches()` function does the necessary adjustments. This is an example of data abstraction.

The above restriction may not appear mandatory. However, very soon we will create classes where similar restrictions will be absolutely necessary (and also complicated).

Similarly, the definition of the `Distance::add()` function should also be modified as follows by the library programmer. Here, it can be assumed that the value of `fInches` portion of neither the invoking object nor the object appearing as formal argument ('dd') can be greater than 12.

```
Distance Distance::add(Distance dd)
{
   Distance temp;
   temp.iFeet=iFeet+dd.iFeet;
   temp.setInches(fInches+dd.fInches);
   return temp;
}
```

Now, if we write the statements shown in Listing 2.10

https://hemanthrajhemu.github.io

Listing 2.10 Enforcing restrictions on the data members of a class

```
d1.setFeet(1);
d1.setInches(9.5);
d2.setFeet(2);
d2.setInches(5.5);
d3=d1.add(d2);
```

then the value of `d3.fInches` will become 3 (not 15) and the value of `d3.iFeet` will become 4 (not 3).

It has already been mentioned that real-world objects never attain an invalid state. They also do not start in an invalid state. Does C++ enable the library programmer to implement this feature in class objects?

Let us continue with our earlier example—the `Distance` class. Recollect that it is the library programmer's intention to ensure that the value of `fInches` portion of none of the objects of the class `Distance` should exceed 12. Now, let us consider Listing 2.11.

Listing 2.11 Object gets created with improper values

```
/*Beginning of DistJunk.cpp*/
#include<iostream.h>
#include"Distance.h"

void main()
{
  Distance d1;
  cout<<d1.getFeet()<<" "<<d1.getInches()<<endl;
}

/*End of DistJunk.cpp*/
```

Output

297 34.56

As you can see, the value of `fInches` of 'd1' is larger than 12! This happened because the value of both `iFeet` and `fInches` automatically got set to junk values when 'd1' was allocated memory and the junk value is larger than 12 for `d1.fInches`. Thus, the objective of the library programmer to keep the value of `fInches` less than 12 has not yet been achieved.

It would be unrealistic to expect that an application programmer will explicitly initialize each object that is declared.

```
Distance d1;
d1.setFeet(0);           //initialization
d1.setInches(0.0);       //initialization
```

Obviously, the library programmer would like to add a function to the `Distance` class that gets called automatically whenever an object is created and sets the values of the data members of the object properly. Such a function is the *constructor*. The concept of constructor and a related function, the *destructor*, is discussed in one of the later chapters.

But we may say that even if `Distance` was an ordinary structure and `setInches()` function was a non-member function just as in C, data abstraction would still be in place. Nevertheless, in the case of C, the library programmer cannot force calls to only those functions that have been defined. He/she cannot prevent calls to those functions that

https://hemanthrajhemu.github.io

he/she has not defined. *Data abstraction is effective due to data hiding only* (recall the case of the overhead projector systems discussed earlier).

On the other side of the coin, in C language, life becomes difficult for an application programmer also. If a certain member of a structure variable acquires an invalid or a wrong value, he/she has to hunt through the entire source code to detect the bug. This problem rapidly gains significance as the code length increases. In actual practice, it is common to have code of more than 25,000 lines.

Let us now sum up as follows:

Perfect definitions of the member functions are guaranteed to achieve their objective because of data hiding.

This is the essence of the object-oriented programming system. Real-world objects have not only working parts but also an exclusive interface to these inner-working parts. A perfect interface is guaranteed to work because of its exclusive rights.

2.1.8 Explicit Address Manipulation

An application programmer can manipulate the member data of any object by explicit address manipulation. Listing 2.12 illustrates the point.

Listing 2.12 Explicit address manipulation

```
/*Beginning of DistAddrManip.cpp*/
#include"Distance.h"
#include<iostream.h>

void main()
{
  Distance d1;
  d1.setFeet(256);
  d1.setInches(2.2);
  char * p=(char *)&d1;              //explicit address manipulation
  *p=1;                              //undesirable but unpreventable
  cout<<d1.getFeet()<<" "<<d1.getInches()<<endl;
}
/*End of DistAddrManip.cpp*/
```

Output

257 2.2

However, such explicit address manipulation by an application programmer cannot be prevented. It is left as an exercise for the readers to explain the output of the above program (Listing 2.12).

2.1.9 Arrow Operator

Member functions can be called with respect to an object through a pointer pointing at the object. The arrow operator (->) does this. An illustrative example is shown in Listing 2.13.

Listing 2.13 Accessing members through pointers

```
/*Beginning of PointerToMember.cpp*/
#include<iostream.h>
#include"Distance.h"
```

https://hemanthrajhemu.github.io

```
void main()
{
    Distance d1;                        //object
    Distance * dPtr;                    //pointer
    dPtr=&d1;                           //pointer initialized
   /*Same as d1.setFeet(1) and d1.setInches(1.1)*/
    dPtr->setFeet(1);                   //calling member functions
    dPtr->setInches(1.1);               //through pointers
   /*Same as d1.getFeet() and d1.getInches()*/
   cout<<dPtr->getFeet()<<" "<<dPtr->getInches()<<endl;
}
/*End of PointerToMember.cpp*/
```

Output

1 1.1

It is interesting to note that just like the dot (.) operator, *the definition of the arrow (->) operator has also been extended in C++*. It takes not only data members on its right as in C, but also member functions as its right-hand side operand. If the operand on its right is a data member, then the arrow operator behaves just as it does in C language. However, if it is a member function of a class where a pointer of the same class type is its left-hand side operand, then the compiler simply passes the value of the pointer as an implicit leading parameter to the function call. Thus, the statement

```
dPtr->setFeet(1);
```

after conversion becomes

```
setFeet(dPtr,1);
```

Now, the value of `dPtr` is copied into the `this` pointer. Therefore, the `this` pointer also points at the same object at which `dPtr` points.

2.1.10 Calling One Member Function from Another

One member function can be called from another. An illustrative example is shown in Listing 2.14.

Listing 2.14 Calling one member function from another

```
/*Beginning of NestedCall.cpp*/
class A
{
     int x;
   public:
     void setx(int);
     void setxindirect(int);
};

void A::setx(int p)
{
   x=p;
}

void A::setxindirect(int q)
{
   setx(q);
```

https://hemanthrajhemu.github.io

```
}
void main()
{
   A A1;
   A1.setxindirect(1);
}
/*End of NestedCall.cpp*/
```

It is relatively simple to explain the above program. The call to the `A::setxindirect()` function changes from

```
A1.setxindirect(1);
```

to

```
setxindirect(&A1,1);
```

The definition of the `A::setxindirect()` function changes from

```
void A::setxindirect(int q)
{
   setx(q);
}
```

to

```
void setxindirect(A * const this, int q)
{
    this->setx(q);                    //calling function through a pointer
}
```

which, in turn, changes to

```
void setxindirect(A * const this, int q)
{
   setx(this,q);                    //action of arrow operator
}
```

2.2 Member Functions and Member Data

Let us study the various kinds of member functions and member data that classes in C++ have.

2.2.1 Overloaded Member Functions

Member functions can be overloaded just like non-member functions. Listing 2.15 illustrates this point.

Listing 2.15 Overloaded member functions

```
/*Beginning of memFuncOverload.cpp*/
#include<iostream.h>
class A
{
   public:
      void show();
      void show(int);             //function show() overloaded!!
};
```

https://hemanthrajhemu.github.io

```
void A::show()
{
   cout<<"Hello\n";
}

void A::show(int x)
{
   for(int i=0;i<x;i++)
      cout<<"Hello\n";
}

void main()
{
   A A1;
   A1.show();                        //first definition called
   A1.show(3);                       //second definition called
}
/*End of memFuncOverload.cpp*/
```

Output

Hello
Hello
Hello
Hello

Function overloading enables us to have two functions of the same name and same signature in two different classes. The class definitions given in Listing 2.16 illustrate the point.

Listing 2.16 Facility of overloading functions permits member functions of two different classes to have the same name

```
class A
{
   public:
      void show();
};
class B
{
   public:
      void show();
};
```

A function of the same name `show()` is defined in both the classes—'A' and 'B'. The signature also appears to be the same. But with our knowledge of the `this` pointer, we know that the signatures are *actually* different. The function prototypes in the respective classes are actually as follows.

```
void show(A * const);
void show(B * const);
```

Without the facility of function overloading, it would not be possible for us to have two functions of the same name in different classes. Without the facility of function overloading, choice of names for member functions would become more and more restricted. Later, we will find that function overloading enables function overriding that, in turn, enables dynamic polymorphism.

https://hemanthrajhemu.github.io

2.2.2 Default Values for Formal Arguments of Member Functions

We already know that default values can be assigned to arguments of non-member functions. Default values can be specified for formal arguments of member functions also. An illustrative example follows in Listing 2.17.

Listing 2.17 Giving default values to arguments of member functions

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

class A
{
  public:
    void show(int=1);
};

void A::show(int p)
{
  for(int i=0;i<p;i++)
    cout<<"Hello\n";
}

void main()
{
  A A1;
   A1.show();                  //default value taken
   A1.show(3);                 //default value overridden
}
/*End of memFuncDefault.cpp*/
```

Output

Hello
Hello
Hello
Hello

Again, it has to be kept in mind that a member function should be overloaded with care if default values are specified for some or all of its formal arguments. For example, the compiler will report an *ambiguity error* when it finds the second prototype for the `show()` function of class A in Listing 2.18.

Listing 2.18 Giving default values to arguments of overloaded member functions can lead to ambiguity errors

```
class A
{
  public:
    void show();
    void show(int=0);            //ambiguity error
};
```

Reasons for such ambiguity errors have already been explained in the section on function overloading in Chapter 1. As in the case of non-member functions, if default values are specified for more than one formal argument, they must be specified from the right to the

https://hemanthrajhemu.github.io

left. Similarly, default values must be specified in the function prototypes and not in function definitions. Further, default values can be specified for a formal argument of any type.

2.2.3 Inline Member Functions

Member functions are made inline by either of the following two methods.

- By defining the function within the class itself (as in Listing 2.5)
- By only prototyping and not defining the function within the class. The function is defined outside the class by using the scope resolution operator. The definition is prefixed by the `inline` keyword. As in non-member functions, the definition of the inline function must appear before it is called. Hence, the function should be defined in the same header file in which its class is defined. Listing 2.19 illustrates this.

Listing 2.19 Inline member functions

```
/*Beginning of memInline.cpp*/
class A
{
   public:
      void show();
};

inline void A::show()          //definition in header file itself
{
                               //definition of A::show() function
}
/*End of memInline.cpp*/
```

2.2.4 Constant Member Functions

Let us consider this situation. The library programmer desires that one of the member functions of his/her class should not be able to change the value of member data. This function should be able to merely read the values contained in the data members, but not change them. However, he/she fears that while defining the function he/she might accidentally write the code to do so. In order to prevent this, he/she seeks the compiler's help. If he/she declares the function as a *constant function*, and thereafter attempts to change the value of a data member through the function, the compiler throws an error.

Let us consider the class `Distance`. The `Distance::getFeet()`, `Distance::getInches()`, and the `Distance::add()` functions should obviously be constant functions. They should not change the values of `iFeet` or `fInches` members of the invoking object even by accident.

Member functions are specified as constants by *suffixing the prototype and the function definition header with the* `const` *keyword*. The modified prototypes and definitions of the member functions of the class `Distance` are illustrated in Listing 2.20.

Listing 2.20 Constant member functions

```
/*Beginning of Distance.h*/
/*Header file containing the definition of the Distance
class*/
class Distance
{
```

https://hemanthrajhemu.github.io

```
        int iFeet;
        float fInches;
    public:
        void setFeet(int);
        int getFeet() const;                    //constant function
        void setInches(float);
        float getInches() const;                //constant function
        Distance add(Distance) const;           //constant function
};
/*End of Distance.h*/

/*Beginning of Distlib.cpp*/
/*Implementation file for the class Distance*/
#include"Distance.h"

void Distance::setFeet(int x)
{
    iFeet=x;
}
int Distance::getFeet() const                   //constant function
{
    iFeet++;            //ERROR!!
    return iFeet;
}

void Distance::setInches(float y)
{
    fInches=y;
}

float Distance::getInches() const               //constant function
{
    fInches=0.0;        //ERROR!!
    return fInches;
}

Distance Distance::add(Distance dd) const       //constant
                                                //function
{
    Distance temp;
    temp.iFeet=iFeet+dd.iFeet;
    temp.setInches(fInches+dd.fInches);
    iFeet++;            //ERROR!!
    return temp;
}
/*End of Distlib.cpp*/
```

For constant member functions, the memory occupied by the invoking object is a read-only memory. How does the compiler manage this? For constant member functions, the `this` pointer becomes 'a constant pointer to a constant' instead of only 'a constant pointer'. For example, the `this` pointer is of type `const Distance * const` for the `Distance::getFeet()`, `Distance::getInches()`, and `Distance::add()` functions. For the other member functions of the class `Distance`, the `this` pointer is of type `Distance * const`.

Clearly, only constant member functions can be called with respect to constant objects. Non-constant member functions cannot be called with respect to constant objects. However, constant as well as non-constant functions can be called with respect to non-constant objects.

https://hemanthrajhemu.github.io

2.2.5 Mutable Data Members

A mutable data member is *never* constant. It can be modified inside constant functions also. Prefixing the declaration of a data member with the keyword `mutable` makes it `mutable`. Listing 2.21 illustrates this.

Listing 2.21 Mutable data members

```
/*Beginning of mutable.h*/
class A
{
    int x;                  //non-mutable data member
    mutable int y;          //mutable data member
  public:
    void abc() const            //a constant member function
    {
    x++;      //ERROR: cannot modify a non-constant data
              //member in a constant member function
    y++;      //OK: can modify a mutable data member in a
              //constant member function
  }
  void def()                    //a non-constant member function
  {
    x++;      //OK: can modify a non-constant data member
              //in a non-constant member function
    y++;      //OK: can modify a mutable data member in a
              //non-constant member function
  }
};
/*End of mutable.h*/
```

We frequently need a data member that can be modified even for constant objects. Suppose, there is a member function that saves the data of the invoking object in a disk file. Obviously, this function should be declared as a constant to prevent even an inadvertent change to data members of the invoking object. If we need to maintain a flag inside each object that tells us whether the object has already been saved or not, such a flag should be modified within the above constant member function. Therefore, this data member should be declared a mutable data member.

2.2.6 Friends

A class can have global non-member functions and member functions of other classes as friends. Such functions can directly access the private data members of objects of the class.

Friend non-member functions

A friend function is a non-member function that has special rights to access private data members of any object of the class of whom it is a friend. In this section, we will study only those friend functions that are not member functions of some other class.

A friend function is prototyped within the definition of the class of which it is intended to be a friend. The prototype is prefixed with the keyword `friend`. Since it is a non-member

https://hemanthrajhemu.github.io

function, it is defined without using the scope resolution operator. Moreover, it is not called with respect to an object. An illustrative example is shown in Listing 2.22.

Listing 2.22 Friend functions

```
/*Beginning of friend.cpp*/
class A
{
     int x;
  public:
       friend void abc(A&);       //prototype of the friend function
};

void abc(A& AObj)     //definition of the friend function
{
   AObj.x++;          //accessing private members of the object
}

void main()
{
  A A1;
  abc(A1);
}
/*End of friend.cpp*/
```

A few points about the friend functions that we must keep in mind are as follows:

- `friend` keyword should appear in the prototype only and not in the definition.
- Since it is a non-member function of the class of which it is a friend, it can be prototyped in either the private or the public section of the class.
- A friend function takes one extra parameter as compared to a member function that performs the same task. This is because it cannot be called with respect to any object. Instead, the object itself appears as an explicit parameter in the function call.
- We need not and should not use the scope resolution operator while defining a friend function.

There are situations where a function that needs to access the private data members of the objects of a class cannot be called with respect to an object of the class. In such situations, the function must be declared as a friend. We will encounter one such situation in Chapter 8.

Friend functions do not contradict the principles of OOPS. Since it is necessary to prototype the friend function inside the class itself, the list of functions that can access the private members of a class's object remains well defined and restricted. The benefits provided by data hiding are not compromised by friend functions.

Friend classes

A class can be a friend of another class. *Member functions of a friend class can access private data members of objects of the class of which it is a friend.* If class B is to be made a friend of class A, then the statement

```
friend class B;
```

should be written within the definition of class A. Listing 2.23 illustrates this.

https://hemanthrajhemu.github.io

Listing 2.23 Declaring friend classes

```
class A
{
  friend class B;                //declaring B as a friend of A
  /*
    rest of the class A
  */
};
```

It does not matter whether the statement declaring class B as a friend is mentioned within the private or the public section of class A. Now, member functions of class B can access the private data members of objects of class A. Listing 2.24 exemplifies this.

Listing 2.24 Effect of declaring a friend class

```
/*Beginning of friendClass.cpp*/
class B;  //forward declaration… necessary because
          //definition of class B is after the statement
          //that declares class B a friend of class A.
class A
{
    int x;
  public:
    void setx(const int=0);
    int getx()const;
    friend class B;              //declaring B as a friend of A
};
class B
{
    A * APtr;
  public:
    void Map(A * const);
    void test_friend(const int);
};
void B::Map(A * const p)
{
  APtr = p;
}
void B::test_friend(const int i)
{
  APtr->x=i;                     //accessing the private data member
}
/*End of friendClass.cpp*/
```

As we can see, member functions of class B are able to access private data member of objects of the class A although they are not member functions of class A. This is because they are member functions of class B that is a friend of class A.

Friendship is not transitive. For example, consider Listing 2.25.

https://hemanthrajhemu.github.io

Listing 2.25 Friendship is not transitive

```
class B;
class C;

/*Beginning of friendTran.cpp*/
class A
{
   friend class B;
   int a;
};

class B
{
   friend class C;
};

class C
{
   void f(A * p)
   {
      p->a++;                      //error: C is not a friend of A
                                   //despite being a friend of a friend
   }
};
/*End of friendTran.cpp*/
```

Friend member functions

How can we make some specific member functions of one class friendly to another class? For making only `B::test_friend()` function a friend of class A, replace the line

```
friend class B;
```

in the declaration of the class A with the line

```
friend void B::test_friend();
```

The modified definition of the class A is

```
class A
{
   /*
      rest of the class A
   */
      friend void B::test_friend();
};
```

However, in order to compile this code successfully, the compiler should first see the definition of the class B. Otherwise, it does not know that `test_friend()` is a member function of the class B. This means that we should put the definition of class B before the definition of class A.

However, a pointer of type A * is a private data member of class B. So, the compiler should also know that there is a class A before it compiles the definition of class B. This problem of *circular dependence* is solved by forward declaration. This is done by inserting the line

```
class A;                           //Declaration only! Not definition!!
```

before the definition of class B. Now, the declarations and definitions of the two classes appear as shown in Listing 2.26.

https://hemanthrajhemu.github.io

Listing 2.26 Forward declaring a class that requires a friend

```
/*Beginning of friendMemFunc.h*/
class A;

class B
{
      A * APtr;
   public:
      void Map(const A * const);
      void test_friend(const int=0);
};

class A
{
      int x;
   public:
      friend void B::test_friend(const int=0);
};
/*End of friendMemFunc.h*/
```

Another problem arises if we try to define the `B::test_friend()` function as an inline function by defining it within class B itself. See Listing 2.27.

Listing 2.27 Problem in declaring a friend member function inline

```
class B
{
   /*
      rest of the class B
   */
   public:
      void test_friend(const int p)
      {
         APtr->x=p;                  //will not compile
      }
};
```

But how will the code inside `B::test_friend()` function compile? The compiler will not know that there is a data member 'x' inside the definition of class A. For overcoming this problem, merely prototype `B::test_friend()` function within class B; *define it as inline* after the definition of class A in the header file itself. The revised definitions appear in Listing 2.28.

Listing 2.28 Declaring a friend member function inline

```
/*Beginning of friendMemFuncInline.h*/
class A;

class B
{
      A * APtr;
   public:
      void Map(const A * const);
      void test_friend(const int=0);
};
```

https://hemanthrajhemu.github.io

```
class A
{
      int x;
   public:
      friend void B::test_friend(const int=0);
};
inline void B::test_friend(const int p)
{
   APtr->x=p;
}
/*End of friendMemFuncInline.h*/
```

Friends as bridges

Friend functions can be used as bridges between two classes.

Suppose there are two unrelated classes whose private data members need a simultaneous update through a common function. This function should be declared as a friend to both the classes. See Listing 2.29.

Listing 2.29 Friends as bridges

```
class B;                              //forward declaration
class A
{
   /*
      rest of the class A
   */
   friend void ab(const A&, const B&);
};
class B
{
   /*
      rest of the class B
   */
   friend void ab(const A&, const B&);
};
```

2.2.7 Static Members

Static member data

Static data members hold global data that is common to all objects of the class. Examples of such global data are

- count of objects currently present,
- common data accessed by all objects, etc.

Let us consider class `Account`. We want all objects of this class to calculate interest at the rate of say 4.5%. Therefore, this data should be globally available to all objects of this class (Listing 2.30).

This data cannot and should not be a member of the objects themselves. Otherwise, multiple copies of this data will be embedded within the objects taking up unnecessary space. Same

https://hemanthrajhemu.github.io

value would have to be maintained for this data in all objects. This is very difficult. Thus, this data cannot be stored in a member variable of class `Account`.

At the same time, this data should not be stored in a global variable. Then the data is liable to be changed by even non-member functions. It will also potentially lead to name conflicts. However, this means that it should be stored in a member variable of class `Account`!

How can this conflict be resolved? Storing the data in a *static variable* of the class resolves this conflict. Static data members are members of the *class* and not of any *object* of the class, that is, they are not contained inside any object.

We prefix the declaration of a variable within the class definition with the keyword `static` to make it a static data member of the class. See Listing 2.30.

Listing 2.30 Declaring a static data member

```
/*Beginning of Account.h*/
class Account
{
  static float interest_rate;                  //a static data member
  /*
    rest of the class Account
  */
};
/*End of Account.h*/
```

A statement declaring a static data member inside a class will obviously not cause any memory to get allocated for it. Moreover, memory for a static data member will not get allocated when objects of the class are declared. This is because a static data member is not a member of any object. Therefore, we must not forget to write the statement to define (allocate memory for) a static member variable. Explicitly defining a static data member outside the class is necessary. Otherwise, the linker produces an error. The following statement allocates memory for `interest_rate` member of class `Account`.

```
float Account::interest_rate;
```

The above statement initializes `interest_rate` to zero. If some other initial value (say 4.5) is desired instead, the statement should be rewritten as follows.

```
float Account::interest_rate=4.5;
```

Static data members should be defined in the implementation files only. The header file is included in both the implementation file and the driver program. If a static data member is defined in the header file, the static data member's definition would be in two files—the library file created from the implementation file and the object file created from the driver program. But in order to get the executable, the linker will have to link these files. Upon finding two definitions of the static data member, the linker would throw an error.

Making static data members private prevents any change from non-member functions as only member functions can change the values of static data members.

Introducing static data members does not increase the size of objects of the class. Static data members are not contained within objects. There is only one copy of the static data member in the memory. Let us try the following program (Listing 2.31) to find out.

https://hemanthrajhemu.github.io

Listing 2.31 Static data members are not a part of objects

```
/*Beginning of staticSize.cpp*/
#include<iostream.h>
class A
{
  int x;
  char y;
  float z;
  static float s;
};
float A::s=1.1;
void main()
{
  cout<<sizeof(A)<<endl;
}
/*End of staticSize.cpp*/
```

Output

9

Static data members can be of any type. For example, name of the bank that has the accounts can be stored as a character array in a static data member of the class as illustrated in Listing 2.32.

Listing 2.32 Static data member can be of any type

```
/*Beginning of Account.h*/

class Account
{
  static float interest_rate;
  static char name[30];
  /*
    rest of the class Account
  */
};

/*End of Account.h*/

/*Beginning of Account.cpp*/
#include"Account.h"

float A::interest_rate=4.5;
char A::name[30]="The Rich and Poor Bank";
/*
  definitions of the rest of the functions of class Account
*/
/*End of Account.cpp*/
```

Static data members of integral type can be initialized within the class itself if the need arises. For example, see Listing 2.33.

https://hemanthrajhemu.github.io

Listing 2.33 Initializing integral static data members within the class itself

```
/*Beginning of Account.h*/
class Account
{
  static int nameLength=30;
  static char name[nameLength];
  /*
    rest of the class Account
  */
};
/*End of Account.h*/
/*Beginning of Account.cpp*/
#include"Account.h"
int A::nameLength;
char A::name[nameLength]="The Rich and Poor Bank";
/*
  definitions of the rest of the functions of class Account
*/
/*End of Account.cpp*/
```

We must notice that the static data member that has been initialized inside the class must be still defined outside the class to allocate memory for it. Once the initial value has been supplied within the class, the static data member must not be re-initialized when it is defined.

Non-integral static data members cannot be initialized like this. For example, see Listing 2.34.

Listing 2.34 Non-integral static data members cannot be initialized within the class

```
/*Beginning of Account.h*/
class Account
{
  static char name[30]="The Rich and Poor Bank";     //error!!
  /*
    rest of the class Account
  */
};
/*End of Account.h*/
```

In Listing 2.33, the variable `nameLength` is referred to directly without the class name and the scope resolution operator while defining the variable `name`. One static data member can directly refer to another without using the scope resolution operator.

Member functions can refer to static data members directly. An example follows (Listing 2.35).

Listing 2.35 Accessing static data members from non-static member functions

```
/*Beginning of Account.h*/
class Account
{
    static float interest_rate;
  public:
```

https://hemanthrajhemu.github.io

```
        void updateBalance();
        /*
           rest of the class Account
        */
};

/*End of Account.h*/

/*Beginning of Account.cpp*/
#include"Account.h"

float Account::interest_rate=4.5;
void Account::updateBalance()
{
   if(end_of_year)
      balance+=balance*interest_rate/100;
}
/*
   definitions of the rest of the functions of class Account
*/
/*End of Account.cpp*/
```

The `object-to-member access operator` can be used to refer to the static data member of a class with respect to an object. The class name with the scope resolution operator can do this directly.

```
f=a1.interest_rate;                 //a1 is an object of the class Account
f=Account::interest_rate;
```

There are some things static data members can do but non-static data members cannot.

- A static data member can be of the *same type* as the class of which it is a member. See Listing 2.36.

Listing 2.36 Static data members can be of the same type as their class

```
class A
{
    static A A1;        //OK : static
    A * APtr;           //OK : pointer
    A A2;               //ERROR!! : non-static
};
```

- A static data member can appear as the *default value* for the formal arguments of member functions of its class. See Listing 2.37.

Listing 2.37 A static data member can appear as the default argument in the member functions

```
class A
{
      static int x;
      int y;
   public:
      void abc(int=x);           //OK
      void def(int=y);           //ERROR!! : object required
};
```

https://hemanthrajhemu.github.io

A static data member can be declared to be a constant. In that case, the member functions will be able to only read it but not modify its value.

Static member functions

How do we create a member function that need not be called with respect to an existing object? This function's sole purpose is to access and/or modify static data members of the class. Static member functions fulfill the above criteria. Prefixing the function prototype with the keyword `static` specifies it as a static member function. However, the keyword `static` should not reappear in the definition of the function.

Suppose there is a function `set_interest_rate()` that sets the value of the `interest_rate` static data member of class `Account`. The application programmer should be able to call this function even if no objects have been declared. As discussed previously, this function should be static. Its definition can be as shown in Listing 2.38.

Listing 2.38 Static member function

```
/*Beginning of Account.h*/
class Account
{
     static float interest_rate;
  public:
     static void set_interest_rate(float);
  /*
     rest of the class Account
  */
};
/*End of Account.h*/

/*Beginning of Account.cpp*/
#include"Account.h"

float Account::interest_rate = 4.5;

void Account::set_interest_rate(float p)
{
  interest_rate=p;
}
/*
  definitions of the rest of the functions of class Account
*/
/*End of Account.cpp*/
```

Now, the `Account::set_interest_rate()` function can be called directly without an object.

```
Account::set_interest_rate(5);
```

Static member functions do not take the `this` pointer as a formal argument. Therefore, accessing non-static data members through a static member function results in compile-time errors. *Static member functions can access only static data members of the class.*

Static member functions can still be called with respect to objects.

```
a1.set_interest_rate(5);          //a1 is an object of the class
                                  //Account
```

https://hemanthrajhemu.github.io

2.3 Objects and Functions

Objects can appear as local variables inside functions. They can also be passed by value or by reference to functions. Finally, they can be returned by value or by reference from functions. Listings 2.39 and 2.40 illustrate all this.

Listing 2.39 Returning class objects

```
/*Beginning of Distance.h*/
class Distance
{
  public:
    /*function to add the invoking object with another
    object passed as a parameter and return the resultant
    object*/
    Distance add(Distance);
    /*
      rest of the class Distance
    */
};
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include"Distance.h"

Distance Distance::add(Distance dd)
{
  Distance temp;
  temp.iFeet=iFeet+dd.iFeet;
  temp.setInches(fInches+dd.fInches);
  return temp;
}
/*
  definitions of the rest of the functions of class
  Distance
*/

/*End of Distance.cpp*/

/*Beginning of Distmain.cpp*/
#include<iostream.h>
#include"Distance.h"

void main()
{
  Distance d1,d2,d3;
  d1.setFeet(5);
  d1.setInches(7.5);
  d2.setFeet(3);
  d2.setInches(6.25);
  d3=d1.add(d2);
  cout<<d3.getFeet()<<" "<<d3.getInches()<<endl;
}

/*End of Distmain.cpp*/
```

Output

9 1.75

https://hemanthrajhemu.github.io

Listing 2.40 Returning class objects by reference

```
/*Beginning of Distance.h*/
/*Header file containing the definition of the Distance
class*/
class Distance
{
/*definition of the class Distance*/
};
Distance& larger(Distance&, Distance&);
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include"Distance.h"
Distance& larger(Distance& dd1, Distance& dd2)
{
   float i,j;
   i=dd1.getFeet()*12+dd1.getInches();
   j=dd2.getFeet()*12+dd2.getInches();
   if(i>j)
      return dd1;
   else
      return dd2;
}
/*
definitions of the rest of the functions of class Distance
*/
/*End of Distance.cpp*/

/*Beginning of Distmain.cpp*/
#include<iostream.h>
#include"Distance.h"
void main()
{
   Distance d1,d2;
   d1.setFeet(5);
   d1.setInches(7.5);
   d2.setFeet(5);
   d2.setInches(6.25);
   Distance& d3=larger(d1,d2);
   d3.setFeet(0);
   d3.setInches(0.0);
   cout<<d1.getFeet()<<» «<<d1.getInches()<<endl;
   cout<<d2.getFeet()<<» «<<d2.getInches()<<endl;
}
/*End of Distmain.cpp*/
```

Output

```
0 0.0
5 6.25
```

2.4 Objects and Arrays

Let us understand how arrays of objects and arrays inside objects are handled in C++.

https://hemanthrajhemu.github.io