

```

passfn.cpp: passing pointer to function type parameters
#include <iostream.h>
int small( int a, int b )
{
    return a < b ? a : b;
}
int large( int a, int b )
{
    return a > b ? a : b;
}
int select( int (*fn)(int, int), int x, int y )
{
    int value = fn( x, y );
    return value;
}
void main( void )
{
    int m, n;
    int (*ptrf)(int, int);    // definition of pointer to function
    cout << "Enter two integers: ";
    cin >> m >> n;
    int high = select( large, m, n );    // function as parameter
    ptrf = small;
    int low = select( ptrf, m, n );    // pointer to function as parameter
    cout << "Large = " << high << endl;
    cout << "Small = " << low;
}

```

Run

```

Enter two integers: 10 20
Large = 20
Small = 10

```

In the above program, the function declarator

```
int select( int (*fn)(int, int), int x, int y )
```

indicates that it takes the pointer to a function as the first parameter and the remaining two integer parameters. In main(), the statement

```
int high = select( large, m, n );    // function as parameter
```

passes the address of the function large() and two integer variables as actual parameters. The pointer to the function parameter large operates on the last two parameters m and n and returns an integer result. Similarly, the statement

```
int low = select( ptrf, m, n );    // pointer to function as parameter
```

passes a pointer to a function variable ptrf (note that, ptrf is initialized to the address of small()). Such a mechanism is useful in selecting the type of operation to be performed at runtime.

9.16 Pointers to Constant Objects

Consider the statement

```
const int* pi;    // it is the same as: int const * pi;
```

302 Mastering C++

It defines `pi` as a pointer to a constant integer. Let `pi` be initialized by the statement

```
int i[ 20 ];
pi = i;
```

i.e., `*pi` would refer to the integer `i[0]`. Due to the definition of `pi` (which, as mentioned above, is `const int* pi;`), statements such as

```
*pi = 10; or even pi[ 10 ] = 20;
```

are invalid. It results in compile time errors. But `pi` itself can be changed, i.e., a statement such as

```
pi++;
```

is perfectly valid. Such pointers can be used as character pointers, when the pointer has to be passed to a function for printing. It is a good practice to code such a function for instance, `print()` as follows:

```
void print( const char* str )
{
    cout << str;
}
```

It accepts a `const char *` (pointer to constant character). The string being pointed to cannot be modified. This is a safety measure, since it avoids accidental modification of the string passed to the function. In the function, the pointer `str` can be changed and a statement such as

```
str++;
```

is valid. But this does not affect the calling procedure, since the pointer is passed by value.

9.17 Constant Pointers

The statement

```
int* const pi = i;
```

defines a constant pointer to an integer (assume that `i` is an integer array). In this case, the use of a statement such as

```
*pi = 10;
```

is perfectly valid, but others that modify the pointer, such as

```
pi++;
```

are invalid and result in compile time errors.

A pointer definition such as

```
const int* const pi = i;
```

will disallow any modifications to `pi` or the integer to which `pi` is referencing. (Assume as before that `i` is an integer array).

9.18 Pointer to Structures

A pointer can also hold the address of user defined data types such as structures. Similar to pointers to standard data types, pointers to user defined data types can be initialized with address of statically or dynamically created data items. Note that in C++, structures can combine both the data and functions operating on it into a single unit. Both the data and function members of structure are accessed in the same way. The syntax for defining pointer to structures is shown in Figure 9.13.

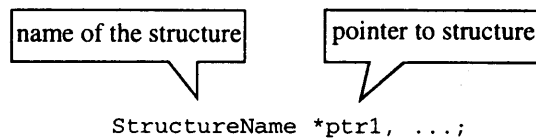


Figure 9.13: Syntax of defining pointer to structure

The syntax for accessing members of a structure using a structure pointer is as follows:

StructPtrVar->MemberName;

The symbol `->` is called the *arrow operator*. (The dot operator connects a structure with a member of the structure; the arrow operator connects a pointer with a member of the structure). The program `bdate.cpp` illustrates the mechanism of creating user defined data type variables dynamically.

```
// bdate.cpp: displaying birth date of the authors
#include <iostream.h>
struct date
{
    //specifies a structure
    int day;
    int month;
    int year;
    void show()
    {
        cout << day << "-" << month << "-" << year << endl;
    }
};
void read( date *dp )
{
    cout << "Enter day: ";
    cin >> dp->day;
    cout << "Enter month: ";
    cin >> dp->month;
    cout << "Enter year: ";
    cin >> dp->year;
}
void main()
{
    date d1, *dp1, *dp2;
    cout << "Enter birth date of boy..." << endl;
    read( &d1 );
    // read date2
    dp2 = new date; // allocate memory dynamically
    cout << "Enter birth date of girl..." << endl;
    read( dp2 );
    cout << "Birth date of boy: ";
    dp1 = &d1; // dp1 points to statically allocated structure
    dp1->show();
    cout << "Birth date of girl: ";
    dp2->show();
}
```

304 Mastering C++

```
    delete dp2;        // release memory
}
```

Run

```
Enter birth date of boy...
Enter day: 14
Enter month: 4
Enter year: 71
Enter birth date of girl...
Enter day: 1
Enter month: 4
Enter year: 72
Birth date of boy: 14-4-71
Birth date of girl: 1-4-72
```

In `main()`, the statement

```
    date d1, *dp1, *dp2;
```

creates variable `d1` and two pointers of type structure `date`. The statement,

```
    dp2 = new date;    // allocate memory dynamically
```

creates the structure `date` type item dynamically and stores its address in a pointer variable `dp2`. The statement

```
    dp1 = &d1;         // dp1 points to statically allocated structure
```

assigns the address of statically created variable `d1` to the pointer variable `dp1`. The statement,

```
    dp1->show();
```

accesses the member function `show()` of `date` using the pointer variable `dp1`. The statement

```
    delete dp2;
```

releases the memory allocated to the pointer variable `dp2`.

Arithmetic Operations on Pointer to structures

Consider the statement

```
    data *d1;
```

It defines the pointer variable `d1` to the structure `date`. The statement

```
    ++d1->day;
```

increments the contents of the member variable `day` and not `d1`. However, the statement

```
    (++)d1->day;
```

increments `d1` first, and then accesses `day`. The statement

```
    d1++->day;
```

increments `d1` after accessing the member variable `day`. The statement

```
    d1++; or ++d1;
```

increments `d1` by `sizeof(date)`.

Self Referential Structure

A structure having references to itself is called a self-referential structure. It is useful for implementing data structures such as linked list, trees, etc. A linked list consists of structures related to each other through pointers. The self referential pointer in the structure points to the next node of a list. The organization of a linked list is shown in Figure 9.14.

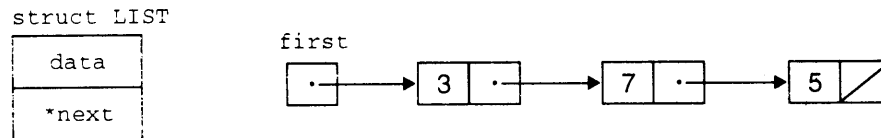


Figure 9.14: Linked list with self-referential structures

The program `list.cpp` illustrates the manipulation of a linked list. It supports `create`, `delete`, and `display` operations on the linked list. The structure `LIST` is a *self referential* structure, since it has a pointer to the next node as one of the data items.

```

// list.cpp: self referential structure-linked list
#include <iostream.h>
#include <new.h>
#include <process.h>
#define SUCC( node ) node->next
struct LIST
{
    int data; // node data
    LIST *next; // pointer to next node
};
// creates node using data and returns pointer to first node of the list
LIST * InsertNode( int data, LIST *first )
{
    LIST *newnode;
    newnode = new LIST; // allocate memory for node
    if( newnode == NULL )
    {
        cout << "Error: Out-of-memory" << endl;
        exit( 1 );
    }
    newnode->data = data; // Initialize list data member
    SUCC( newnode ) = first; // new node becomes first node
    return newnode;
}
// deletes node whose data matches input data and returns updated list
LIST * DeleteNode( int data, LIST *first )
{
    LIST *current, *pred; // work space for insertion
    if( !first )
    {
        cout << "Empty list" << endl;
        return first;
    }
    for(pred=current=first;current; pred=current,current = SUCC( current ))
        if( current->data == data )
        {
            // node found, release this node
            if( current == first ) // if node to be deleted is first node
                first = SUCC( current ); // then update list pointer
        }
}

```

306 Mastering C++

```
        else
            SUCC( pred ) = SUCC( current ); // bypass the node
        delete current; // release allocated memory
        return first;
    }
    return( first );
}
// Display list
void DisplayList( LIST *first )
{
    LIST *list;
    for( list = first; list; list = SUCC( list ) )
        cout << "->" << list->data;
    cout << endl;
}
void main()
{
    LIST *list = NULL; // list is empty
    int choice, data;
    set_new_handler( 0 ); // makes new to return to NULL if it fails
    cout << "Linked-list manipulations program...\n";
    while(1)
    {
        cout << "List operation, 1- Insert, 2-Display, 3-Delete, 4-Quit: ";
        cin >> choice;
        switch( choice )
        {
            case 1:
                cout << "Enter data for node to be created: ";
                cin >> data;
                list = InsertNode( data, list );
                break;
            case 2:
                cout << "List contents: ";
                DisplayList( list );
                break;
            case 3:
                cout << "Enter data for node to be delete: ";
                cin >> data;
                list = DeleteNode( data, list );
                break;
            case 4:
                cout << "End of Linked List Computation !!.\n";
                return;
            default:
                cout << "Bad Option Selected\n";
                break;
        }
    }
}
```

Run

```

Linked-list manipulations program...
List operation, 1- Insert, 2-Display, 3-Delete, 4-Quit: 1
Enter data for node to be created: 5
List operation, 1- Insert, 2-Display, 3-Delete, 4-Quit: 1
Enter data for node to be created: 7
List operation, 1- Insert, 2-Display, 3-Delete, 4-Quit: 1
Enter data for node to be created: 3
List operation, 1- Insert, 2-Display, 3-Delete, 4-Quit: 2
List contents: ->3->7->5
List operation, 1- Insert, 2-Display, 3-Delete, 4-Quit: 3
Enter data for node to be delete: 7
List operation, 1- Insert, 2-Display, 3-Delete, 4-Quit: 2
List contents: ->3->5
List operation, 1- Insert, 2-Display, 3-Delete, 4-Quit: 4
End of Linked List Computation !!.

```

In `main()`, the statement

```
list = InsertNode( data, list );
```

takes an integer type data and a pointer to the first node as input parameters. It returns a pointer to the updated linked list. Initially, the second parameter has to be set to `NULL` indicating a empty linked list.

The statement

```
list = DeleteNode( data, list );
```

deletes a node which matches with the parameter data and returns the address of the first node in the linked list to the pointer list. The statement

```
DisplayList( list );
```

prints the data information contents of a linked list on the console.

9.19 Wild Pointers

Pointers have to be handled very carefully since, issues associated with them are confusing. Especially, the scope and extent of a data object, to which a pointer is pointing to is a crucial aspect. Pointers exhibit wild behavior if these crucial issues are not taken into consideration while accessing data. A pointer becomes a *wild pointer* when it is pointing to an unallocated memory or when it is pointing to a data item whose memory is already released. Side effects of such pointers are creation of *garbage memory* and *dangling reference*. The memory becomes *garbage memory* when a pointer pointing to a memory object (data item) is lost; i.e., it indicates that the memory item continues to exist, but the pointer to it is lost; it happens when memory is not released explicitly. A memory access using a pointer is known as *dangling reference* when a pointer to the memory item continues to exist, but memory allocated to that item is released; i.e., accessing memory object, for which no memory is allocated. Pointers become wild pointers under the following situations:

- ◆ When a pointer is uninitialized
- ◆ Pointer modification
- ◆ Pointer referencing to a data which is destroyed

(1) *When pointer is uninitialized*: It contains an illegal address and it is difficult to predict the outcome

of a program. For instance, in the definition

```
int *p;
```

it is impossible to predict which integer value the pointer `p` is pointing to. The pointer `wild1.cpp` illustrates accessing data through the uninitialized variables.

```
// wild1.cpp: accessing uninitialized pointer
#include <iostream.h>
void main()
{
    int *p; // pointer is uninitialized
    for( int i = 0; i < 10; i++ )
        cout << p[i] << " "; // accessing uninitialized pointer
}
```

Run (under MS-DOS)

```
0 21838 19532 17184 17736 19267 0 14 0 -1
```

Run (under UNIX)

```
-2130509557 73728 8192 0 105384 8224 0 0 -1139793920 -80506873
```

It can be observed that, the output generated by the program is different from system to system. The use of a statement such as

```
p[1] = 10;
```

might modify some sensitive data pertaining to a system leading to corruption of the whole system or the program may behave erratically. Under UNIX system, such errors will lead to segment violation error as illustrated in the program `wild2.cpp`.

```
// wild2.cpp: assigning data using uninitialized pointers
#include <iostream.h>
#include <string.h>
void main()
{
    char *name;
    strcpy( name, "Savithri " ); // assigning without memory allocation
    cout << name;
}
```

Run (under MS-DOS)

```
Savithri Null pointer assignment
```

Run (under UNIX)

```
Segmentation fault (core dumped)
```

In `main()`, the statement

```
strcpy( name, "Savithri " );
```

assigns the string "Savithri " to a pointer to string, for which memory is not allocated. From the output, it can be noted that, in the UNIX environment the program immediately terminates by core dumping when such a situation is detected. Hence, use a statement such as

```
name = new char[ 10 ];
```


to avoid such runtime errors before trying to store anything in the memory.

(2) *Pointer modification*: The inadvertent storage of a new address in a pointer variable is referred to as pointer modification. This situation will occur when some other wild pointer modifies the address of a valid pointer. It transforms a valid pointer to a wild pointer.

(3) *Pointer referencing to a data which is destroyed*. In this case, the pointer tries to access memory object or item which no longer exists. It is illustrated in the program wild3.cpp.

```
// wild3.cpp: assigning destroyed object
#include <iostream.h>
#include <string.h>
char * nameplease();
char * charplease();
void main()
{
    char *p1, *p2;
    p1 = nameplease();
    p2 = charplease();
    cout << "Name = " << p1 << endl;
    cout << "Char = " << p2 << endl;
}
char * nameplease()
{
    char name[] = "Savithri ";
    return name;
}
char * charplease()
{
    char ch;
    ch = 'X';
    return &ch;
}
```

Run

```
Name = SavivN'
Char = i
```

In the function nameplease(), invoked by the statement

```
p1 = nameplease();
```

when the address of the variable name is returned, the control comes out of the function nameplease() and hence, the variable name dies (since it is an auto variable). Thus p1 would contain the address of the variable which does not exist. In effect, this is a situation of dangling reference. In such a situation the compiler issues a warning such as

```
Suspicious pointer reference
```

```
Or
```

```
Returning a reference to a local object
```

It implies that a pointer or reference to a local (auto) variable/object should never be returned. As soon as the function is terminated, the memory assigned to the local variable is released or gets destroyed, and any reference or pointer points to some invalid data. However, returning a copy (return by value) of a local variable/object is valid.

310 Mastering C++

Another important point to be noted is that, avoid storing the address of a variable or an object into a pointer in the inner block, and using the same in the outer block. The program `wild4.cpp` illustrates the wild pointer accessing garbage location.

```
// wild4.cpp: out of scope of a block variable access
#include <iostream.h>
#include <string.h>
void main()
{
    char *p1;
    {
        char name[] = "Savithri ";
        p1 = name;
    }
    // do some processing here
    cout << "Name = " << p1 << endl;
}
```

Run

Name = Savith@ \$!

In `main()`, the statement

```
cout << "Name = " << p1 << endl;
```

accesses the data pointed to by the pointer variable `p1`. The variable `p1` is assigned to point to the variable `name` defined within an inner block. When the execution of this block is completed, all the variables are destroyed and hence, accessing of data stored in the variable `name` becomes invalid data. In some situation, the programs might execute properly, but they may corrupt other program's data and lead to system crash.

The above discussion also holds good for pointer to objects. Like variables, whenever objects goes out of scope, they are destroyed. Referencing such objects is like accessing invalid-data variable and hence, such reference should be avoided.

Review Questions

- 9.1 What are pointers ? What are the advantages of using pointers in programming ? Explain *addressing mode* required to access memory locations using pointers.
- 9.2 Under what situations, the use of pointers is indispensable ?
- 9.3 Write a program to print address of the variables defined by the following statement:

```
int a, b = 10;
float c = 2, d;
```
- 9.4 Explain the syntax for defining pointer variables. How different are these from normal variables?
- 9.5 What is dereferencing of pointers ? Write a program to dereference the pointer variables in the following statements (print value pointed to by pointer variables):

```
int *a; double *b;
a = &i; b = &f;
```
- 9.6 What are the differences between passing parameters by value and by pointers ? Give examples.

- 9.7 What are the different arithmetic operations that can be performed on pointer variables ? Consider the following definitions:

```
int *a, *b, c; float *e; char *p;
```

The pointer variables a, b, and c are initially pointing to memory locations 100, 150, and 50 (assume) respectively. What is the address stored in the pointer variable (a, b, and c) on execution of the following statements ?

```
a++;
b = --a;
cout << *b++;
cout << *++p;
e++;
a = &c;
```

- 9.8 Consider the following definitions:

```
int *a, *b, c; float *e; char *p; int i1, *ip;
char ch; long l; double *d; long double lb;
```

What is the return value of `sizeof()` operator when applied to the variables created by the above statements individually? For instance, the return value of `sizeof(int)` or `sizeof(i1)` is 2 (in DOS) and 4 (in UNIX). Comment on such differences.

- 9.9 What is runtime memory management ? What support is provided by C++ for this and how does it differ from C's memory management ?
- 9.10 Write a program for finding the smallest and largest in a list of N numbers. Accept the value of N at runtime and allocate the necessary amount of storage for storing numbers.
- 9.11 Write an interactive program for manipulation of matrices. Support addition, subtraction, and multiplication operations on them. Create matrices dynamically.
- 9.12 Write a program for sorting names of persons by swapping pointers instead of data. Use Comb sort algorithm for sorting. (Comb sort is explained in the chapter *Arrays and Strings*).
- 9.13 Explain syntax for defining pointers to functions. Write a program which supports the following:

```
a = compute( sin, 1.345 );
b = compute( log, 150 );
c = computer( sqrt, 4.0 );
```

- 9.14 Consider the function `show()`, which is defined as follows:

```
void show( int a, int b, int c)
{
    cout << a << " " << b << " " << c;
}
int *i, j;
i = &j;
j = 2;
int k[] = { 1, 2, 3 };
```

What is the output of the following statements: (Note that actual parameters are evaluated from right to left while assigning them to formal parameters)

```
show( *i, j, *k );
show( *i, *i++, *i);
show( *k, *k++, *k++ );
```

- 9.15 What are the differences between pointers to constants and constant pointers ? Give examples.
- 9.16 Write a program for creating a linked list and support insertion and deletion operations on it.

312 Mastering C++

Nodes of linked list have to be modeled using nested structures.

- 9.19 Define the following: (a) Wild pointers (b) Garbage (c) Dangling reference. Consider the following program:

```
#include <iostream.h>
void main()
{
    int * a;
    const int *b;
    int *const p;
    int c = 2, d = 3;
    cout << a; b = &c; p = &d;
    *b = 10;
    b = new int;
    *b = 10;
    delete b;
    cout << *b;
    a = new int[10];
    a[9] = 20;
    a[10] = 30;
    a = new int[5];
    a++;
    ++b;
    cout << *a;
}
```

Observe the above program carefully and find out where all garbage, dangling reference, and wild pointers exist. Identify statements which are treated as erroneous by the compiler.

- 9.20 Write the function `locate(s, pattern)`, which returns -1 if the string `pattern` does not exist in `s`, otherwise returns location at which it is found.
- 9.21 Consider the following statements:

```
char *name;
char str[20];
name = new char[ strlen(str)+1 ];
strcpy( name, str );
```

Why one more extra byte is allocated to the string `name`? What will happen if one extra byte is not allocated? What is the effect of the following statements during runtime:

```
char *s;
cin >> s;
```

Does the second statement leads to any runtime error? Give reasons.

10

Classes and Objects

10.1 Introduction

Object-oriented programming paradigm is playing an increasingly significant role in the design and implementation of software systems. It simplifies the development of large and complex software systems and helps in the production of software, which is modular, easily understandable, reusable, and adaptable to changes. The object-oriented approach centers around modeling the real world problems in terms of objects (*data decomposition*), which is in contrast to older, more traditional approaches that emphasize a function oriented view, separating data and procedures (*algorithm decomposition*). Object oriented modeling is a new way of visualizing problems using models organized around the real-world concepts. Objects are the result of programming methodology rather than a language.

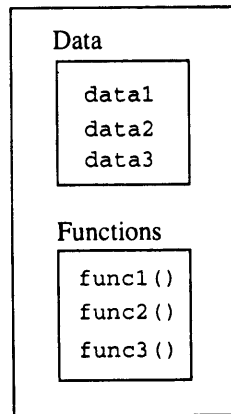


Figure 10.1: Class grouping data and functions

Object-oriented programming constructs modeled out of data types called *classes*. Defining variables of a class data type is known as a *class instantiation* and such variables are called *objects*. (Object is an instance of a class.) A class encloses both the *data* and *functions* that operate on the data, into a *single unit* as shown in Figure 10.1. The variables and functions enclosed in a class are called *data members* and *member functions* respectively. Member functions define the permissible operations on the data members of a class.

Placing data and functions together in a single unit is the central theme of object-oriented programming. The programmers are entirely responsible for creating their own classes and can also have access to classes developed by the software vendors.

Classes are the basic language construct of C++ for creating the user defined data types. They are syntactically an extension of structures. The difference is that, all the members of structures are *public by default* whereas, the members of classes are *private by default*. Class follows the principle that *the information about a module should be private to the module unless it is specifically declared public*.

10.2 Class Specification

C++ provides support for defining classes, which is a significant feature that makes C++ an object oriented language. In C terms, a class is the natural evolution of a structure. Classes contain not only data but also functions. The functions are called member functions and define the set of operations that can be performed on the data members of a class. Thus, a class can be described as a collection of data members along with member functions. This property of C++, which allows association of data and functions into a single unit is called *encapsulation*. Sometimes, classes may not contain any data members or member functions (and such classes are called as *empty classes*). The syntax of a class specification is shown in Figure 10.2.

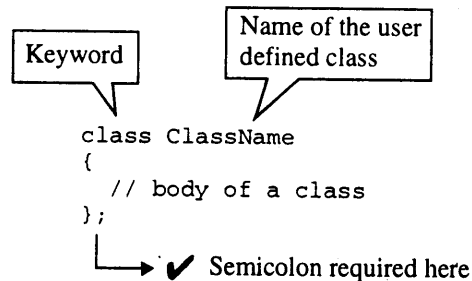


Figure 10.2: Syntax of class specification

The class specifies the type and scope of its members. The keyword `class` indicates that the name which follows (`ClassName`) is an abstract data type. The body of a class is enclosed within the curly braces followed by a semicolon—the end of a class specification. The body of a class contains declaration of variables and functions, collectively known as *members*. The variables declared inside a class are known as *data members*, and functions are known as *member functions*. These members are usually grouped under two sections, *private* and *public*, which define the visibility of members.

The private members are accessible only to their own class's members. On the other hand, public members are not only accessible to their own members, but also from outside the class. The members in the beginning of class without any access specifier are *private by default*. Hence, the first use of the keyword `private` in a class is optional. A class which is totally *private* is hidden from the external world and will not serve any useful purpose.

The following declaration illustrates the specification of a class called `student` having `roll_no` and `name` as its data members:

```

class student
{
    int roll_no;           // roll number
    char name[ 20 ];      // name of a student
public:

```

```

void setdata( int roll_no_in, char *name_in )
{
    roll_no = roll_no_in;
    strcpy( name, name_in );
}
void outdata()
{
    cout << "Roll No = " << roll_no << endl;
    cout << "Name = " << name << endl;
}
};

```

A class should be given some meaningful name, (for instance, `student`) reflecting the information it holds. The class name `student` becomes a new data type identifier, which satisfies the properties of *abstraction*; it can be used to define instances of class data type. The class `student` contains two data members and two member functions. The data members are private by default while both the member functions are public as specified. The member function `setdata()` can be used to assign values to the data members `roll_no` and `name`. The member function `outdata()` can be used for displaying the value of data members. The data members of the class `student` cannot be accessed by any other function except member functions of the `student` class. It is a general practice to declare data members as *private* and member functions as *public*. Three different notations for representation of the `student` class is shown in Figure 10.3.

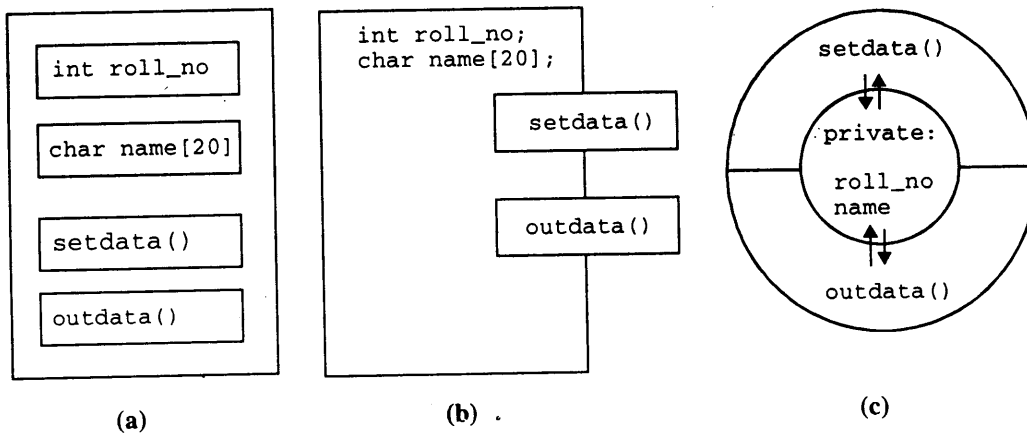


Figure 10.3: Different representations of the class student

The name of data and member functions of a class can be the same as those in other classes; the members of different classes do not conflict with each other. Essentially, a class identifies all the data members associated with its declaration. The following example illustrates this concept:

```

class Person
{
    private:
        char name[20];
        int age;
        .....
};

```

The data member name appears in the student class and in the Person class declarations, but their scope is limited to their respective classes. However, *more than one class with the same class-name in a program is an error, whether the declarations are identical or not*. A class can have multiple member functions (but not data members) with the same name as long as they differ in terms of *signature*; this feature is known as *method overloading*.

Like structures, the data members of the class cannot be initialized during their declaration, but they can be initialized by its member functions as follows:

```
class GeoObject
{
    ....
    float x, y = 5;    // Error: data members cannot be initialized here
    void SetOrigin() // set point to origin
    {
        x = y = 0.0;
    }
};
```

The data members `x` or `y` of the class `GeoObject` cannot be initialized at the point of their declaration, but, they can be initialized in member functions as indicated in the `SetOrigin()` member function.

10.3 Class Objects

A class specification only declares the structure of objects and it must be instantiated in order to make use of the services provided by it. This process of creating objects (variables) of the class is called *class instantiation*. It is the definition of an object that actually creates objects in the program by setting aside memory space for its storage. Hence, a class is like a *blueprint of a house* and it indicates *how the data and functions are used* when the class is instantiated. The necessary resources are allocated only when a class is instantiated. The syntax for defining objects of a class is shown in Figure 10.4. Note that the keyword `class` is optional.

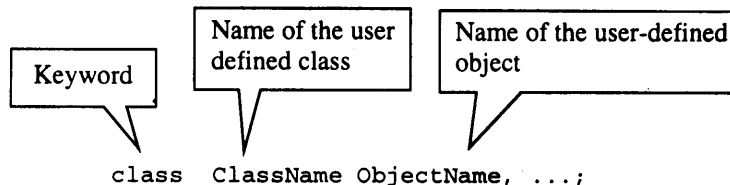


Figure 10.4: Syntax for creating objects

An example of class instantiation for creating objects is shown below:

```
class student s1;
or
student s1;
```

It creates the object `s1` of the class `student`. More than one object can be created with a single statement as follows:

```
class student s1, s2, s3, s4;
or
student s1, s2, s3, s4;
```

It creates multiple objects of the class `student`.

The definition of an object is similar to that of a variable of any primitive data type. Objects can also be created by placing their names immediately after the closing brace like in the creation of the structure variables. Thus, the definition

```
class student
{
    ....
    ....
} s1, s2, s3, s4;
```

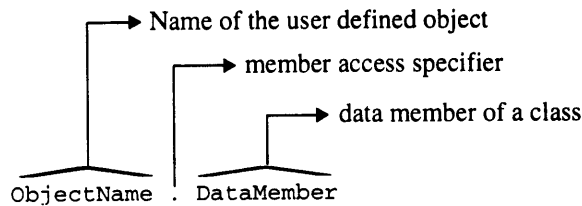
creates objects *s1*, *s2*, *s3*, and *s4* of the class *student*. In C++, the convention of defining objects at the point of class specification is rarely followed; the user would like to define the objects as and when required, or at the point of their usage.

An object is a *conceptual entity* possessing the following properties:

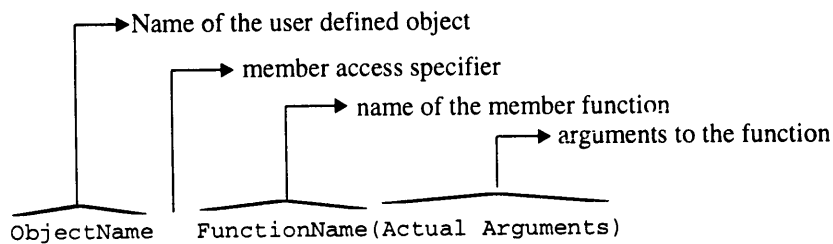
- ◆ it is identifiable.
- ◆ it has features that span a local state space.
- ◆ it has operations that can change the status of the system locally, while also inducing operations in peer objects.
- ◆ it refers to a thing, either a tangible or a mental construct, which is identifiable by the users of the target system.

10.4 Accessing Class Members

Once an object of a class has been created, there must be a provision to access its members. This is achieved by using the member access operator, dot (.). The syntax for accessing members (data and functions) of a class is shown in Figure 10.5.



(a) Syntax for accessing data member of a class



(b) Syntax for accessing member function of a class

Figure 10.5: Syntax for accessing class members

318 Mastering C++

If a member to be accessed is a function, then a pair of parentheses is to be added following the function name. The following statements access member functions of the object `s1`, which is an instance of the `student` class:

```
s1.setdata( 10, "Rajkumar" );
s1.outdata();
```

The program `student.cpp` illustrates the declaration of the class `student` with the operations on its objects.

```
// student.cpp: member functions defined inside the body of the student class
#include <iostream.h>
#include <string.h>
class student
{
private:
    int roll_no;          // roll number
    char name[ 20 ];     // name of a student
public:
    // initializing data members
    void setdata( int roll_no_in, char *name_in )
    {
        roll_no = roll_no_in;
        strcpy( name, name_in );
    }
    // display data members on the console screen
    void outdata()
    {
        cout << "Roll No = " << roll_no << endl;
        cout << "Name = " << name << endl;
    }
};
void main()
{
    student s1;          // first object/variable of class student
    student s2;          // second object/variable of class student
    s1.setdata( 1, "Tejaswi" ); // object s1 calls member setdata()
    s2.setdata( 10, "Rajkumar" ); // object s2 calls member setdata()
    cout << "Student details..." << endl;
    s1.outdata();        // object s1 calls member function outdata()
    s2.outdata();        // object s2 calls member function outdata()
}
```

Run

```
Student details...
Roll No = 1
Name = Tejaswi
Roll No = 10
Name = Rajkumar
```

The various actions performed on objects of the class `student` are portrayed in Figure 10.6 with the client object accessing the services provided by the class `student`.

In `main()`, the statements

```

student s1; // first object/variable of class student
student s2; // second object/variable of class student

```

create two objects called `s1` and `s2` of the `student` class. The statements

```

s1.setdata( 1, "Tejaswi" ); //object s1 calls member function setdata
s2.setdata( 10,"Rajkumar" ); //object s2 calls member function setdata

```

initialize the data members of the objects `s1` and `s2`. The object `s1`'s data member `roll_no` is assigned 1 and name is assigned `Tejaswi`. Similarly, the object `s2`'s data member `roll_no` is assigned 10 and name is assigned `Rajkumar`.

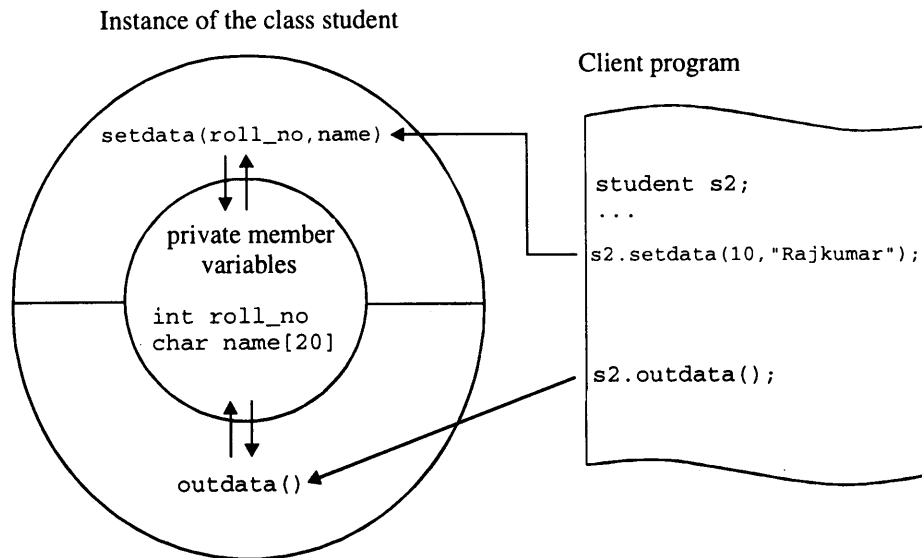


Figure 10.6: Student object and member access

The statements

```

s1.outdata(); // object s1 calls member function outdata
s2.outdata(); // object s2 calls member function outdata

```

call their member `outdata()` to display the contents of data members namely, `roll_no` and name of student objects `s1` and `s2` in succession. Thus, the two objects `s1` and `s2` of the class `student` have different data values as shown in Figure 10.7.

Client-Server Model

In conventional programming languages, a *function is invoked on a piece of data* (function-driven communication), whereas in an OOP (object-oriented programming language), a *message is sent to an object* (message-driven communication) i.e., conventional programming is based on *function abstraction* whereas, object oriented programming is based on *data abstraction*.

The object accessing its class members resembles a client-server model. A client seeks a service whereas, a server provides services requested by a client. In the above example, the class `student`

resembles a *server* whereas, the objects of the class `student` resemble *clients*. They make calls to the server by sending messages. In the statement

```
s2.setdata( 10, "Rajkumar" ); // object s2 calls member function setdata
```

the object `s2` sends the message `setdata` to the server with the parameters `10` and `Rajkumar`. As a server, the member function `setdata()` of the class `student` performs the operation of setting the data members according to the messages sent to it. Similarly, the statement

```
s2.outdata();
```

can be visualized as sending message (`outdata`) to object `s2`'s class to display object contents. The term message is commonly used in OOPs terminology to provide an illusion of objects as discrete entities, and a user communicates with them by calling their member functions as shown in Figure 10.8. Thus, by its very nature, *OO computation resembles a client-server computing model*.

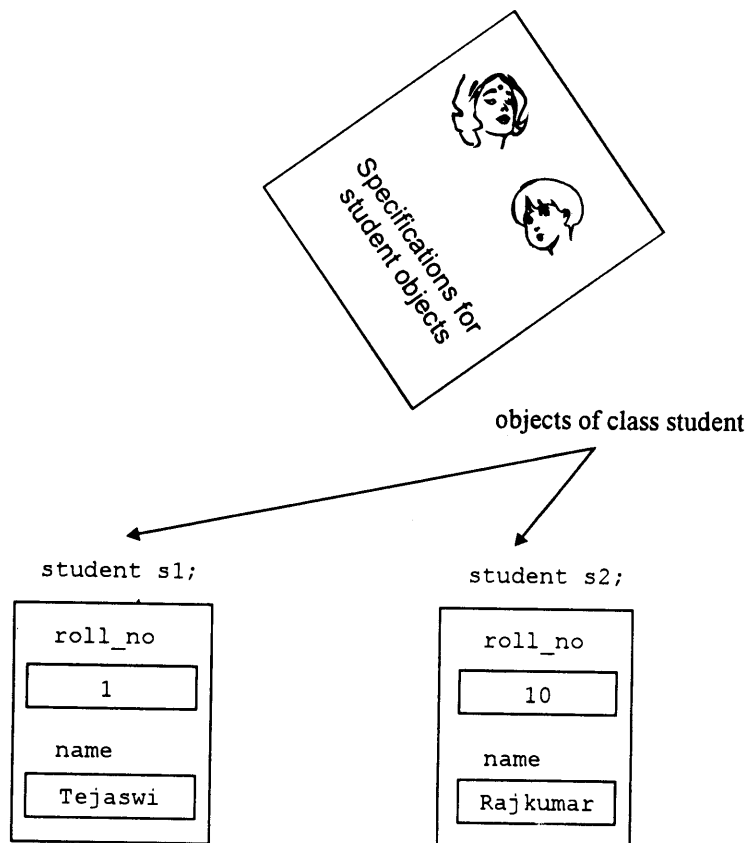


Figure 10.7: Two objects of the class student

In OOPs, the process of programming involves the following steps:

- ◆ Creation of classes for defining objects and their behaviors.
- ◆ Creation of class objects; class declaration acts like a blueprint for which physical resources are not allocated.
- ◆ Establishment of communication among objects through message passing

Similar to the real world objects, OO objects also have a life cycle. They can be created and destroyed *automatically* whenever necessary. Communication between the objects can take place as long as they are alive (active). Communication among the objects takes place in the same way as people pass messages to one another. The concept of programming with *message passing model* is an efficient way of modeling real-world problems on computers.

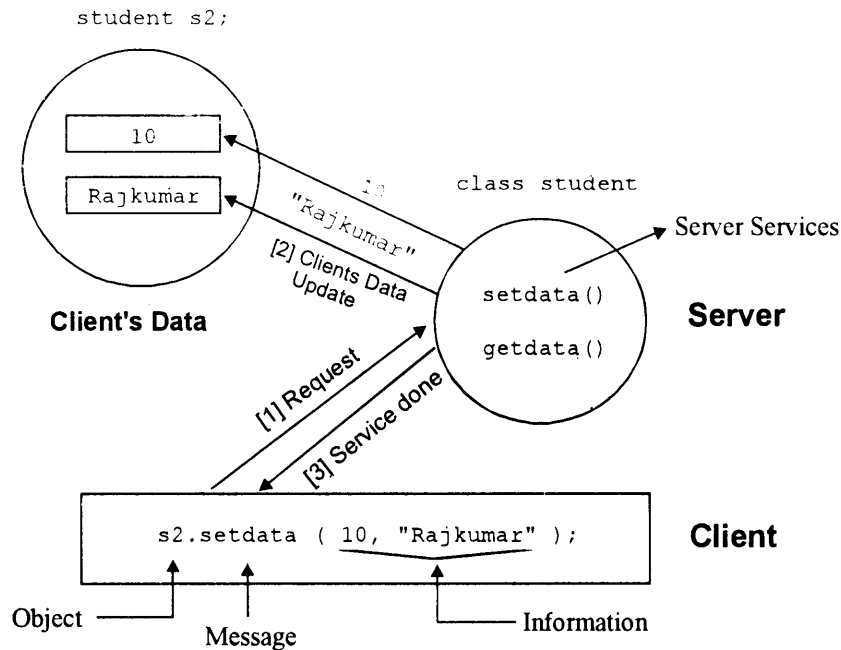


Figure 10.8: Client-Server model for message communication

A message for an object is interpreted as a request for execution of a procedure. The subroutine or function is invoked soon after receiving the message and the desired results are generated within an object. It comprises the name of an object, the name of a function, and the information to be sent to an object.

10.5 Defining Member Functions

The data members of a class must be declared within the body of the class, whereas the member functions of the class can be defined in any one of the following ways:

- ◆ Inside the class specification
- ◆ Outside the class specification

The syntax of a member function definition changes depending on whether it is defined inside or outside the class specification. However, irrespective of the location of their definition (inside or outside a class), the member function must perform the same operation. Therefore, the code inside the function body would be identical in both the cases. The compiler treats these two types of function definitions in a different manner.

Member Functions Inside the Class Body

The syntax for specifying a member function declaration is similar to a normal function definition except that it is enclosed within the body of a class and is shown in Figure 10.9. All the member functions defined within the body of a class are treated as inline by default except those members having looping statements such as `for`, `while`, etc., and it also depends on compilers.

```

class ClassName
{
    private:
        int age;
        int SetAge(int agein) — Member function
        {
            age = agein; // body of the function
        }
        . . .
    public:
        int b;
        void myfunc() — Member function
        {
            // body of a function
        }
};

```

Figure 10.9: Member function defined within a class

The program `date1.cpp` demonstrating the definition of member functions with the class specification of the `date` class. It has private data members `day`, `month`, `year` and inline member functions, `set()` which initializes data members and `show()`, which displays the value stored in the data members.

```

// date1.cpp: date class with member functions defined inside a class
#include <iostream.h>
class date
{
    private:
        int day;
        int month;
        int year;
    public:
        void set( int DayIn, int MonthIn, int YearIn )
        {
            day = DayIn;
            month = MonthIn;
            year = YearIn;
        }
        void show()
        {
            cout << day << "-" << month << "-" << year << endl;
        }
};

```

```

void main()
{
    date d1, d2, d3;    // date objects d1, d2, and d3 creation
    // set date of births
    d1.set( 26, 3, 1958 );
    d2.set( 14, 4, 1971 );
    d3.set( 1, 9, 1973 );
    cout << "Birth Date of the First Author: ";
    d1.show();
    cout << "Birth Date of the Second Author: ";
    d2.show();
    cout << "Birth Date of the Third Author: ";
    d3.show();
}

```

Run

```

Birth Date of the First Author: 26-3-1958
Birth Date of the Second Author: 14-4-1971
Birth Date of the Third Author: 1-4-1972

```

Member functions defined inside a class are considered as *inline* functions by default thus, offering both advantages and limitations of inline functions. However, in some implementations, member functions having loop instructions such as `for`, `while`, `do..while`, etc., are not treated as inline functions. The compiler produces a warning message if an attempt is made to define inline member functions with loop instructions. Normally, functions with a small body are defined inside the class specification. In the above `student` class specification, the functions `set()` and `show()` are treated as *inline* functions by the compiler.

Member Functions Outside the Class Body

Another method of defining a member function is to declare *function prototype* within the body of a class and then define it outside the body of a class. Since the functions defined outside the class specification have the same syntax as normal functions, there should be a mechanism of binding the functions to the class to which they belong. This is done by using the *scope resolution operator* (`::`). It acts as an *identity-label* to inform the compiler, the class to which the function belongs. The general format of a member function definition is shown in Figure 10.10. This form of syntax can be used with members defined either inside or outside the body of a class, but member functions defined outside the body of a class must follow this syntax.

```

class ClassName
{
    ....
    ReturnType MemberFunction(arguments);  function prototype
    ....
};
ReturnType ClassName :: MemberFunction ( arguments )
{
    // body of the function
}

```

Figure 10.10: Member function definition outside a class declaration

The label `ClassName::` informs the compiler that the function `MemberFunction` is the member of the class `ClassName`. The scope of the function is restricted to only the objects and other members of the class. The program `date1.cpp` having member functions inside the body of the `date` class is modified to `date2.cpp` which defines member functions outside the body of a class.

// **date2.cpp**: `date` class with member functions defined outside the class body

```
#include <iostream.h>
class date
{
    private:
        int day;
        int month;
        int year;
    public:
        void set( int DayIn, int MonthIn, int YearIn ); //declaration
        void show(); // declaration
};
void date::set( int DayIn, int MonthIn, int YearIn ) //definition
{
    day = DayIn;
    month = MonthIn;
    year = YearIn;
}
void date::show() // definition
{
    cout << day << "-" << month << "-" << year << endl;
}
void main()
{
    date d1, d2, d3; // date objects d1, d2, and d3 creation
    // set date of births
    d1.set( 26, 3, 1958 );
    d2.set( 14, 4, 1971 );
    d3.set( 1, 9, 1973 );
    cout << "Birth Date of the First Author: ";
    d1.show();
    cout << "Birth Date of the Second Author: ";
    d2.show();
    cout << "Birth Date of the Third Author: ";
    d3.show();
}
```

Run

```
Birth Date of the First Author: 26-3-1958
Birth Date of the Second Author: 14-4-1971
Birth Date of the Third Author: 1-4-1972
```

Consider the member functions `set` and `show` defined in the above program:

```
void date::set( int DayIn, int MonthIn, int YearIn )
{
    day = DayIn;
    ....
}
```



```

void date::show()
{
    cout << day << "-" << month << "-" << year << endl;
}

```

In the above definitions, the label `date::` informs the compiler that the functions `set` and `show` are the members of the `date` class. It can access all the members (data and functions) of the `date` class and also global data items and functions if necessary. Some of the special characteristics of the member functions are the following.

- ◆ A program can have several classes and they can have member functions with the same name. The ambiguity of the compiler in deciding *which function belongs to which class* can be resolved by the use of membership label (`ClassName::`), the scope resolution operator.
- ◆ Private members of a class, can be accessed by all the members of the class, whereas non-member functions are not allowed to access. However, friend functions (discussed later) can access them.
- ◆ Member functions of the same class can access all other members of their own class without the use of dot operator.
- ◆ Member functions defined as `public` act as an interface between the service provider (server) and the service seeker (client).
- ◆ A class can *have* multiple member functions with the same name as long as they differ in terms of argument specification (data type or number of arguments).

10.6 Outside Member Functions as `inline`

OOP provides feature of separating policy from the mechanism. Policy provides guidelines for defining specification whereas mechanism provides guidelines for design and implementation. It is a good practice to declare the class specification first and then implement class member functions outside the class specification. The inline member functions are a group of member functions that decrease the overhead involved in accessing member functions and make the usage of member functions more efficient. An *inline member function* is treated like a *macro*; any call to this function in a program is replaced by the function itself. This is called *inline expansion*. By this, the overhead incurred in the transfer of control by the function call and the function return statements are cut down. Note that inline functions are also called *open subroutines* since they get expanded at the point of a call whereas, normal functions are called *closed subroutines* since only call to a function exists at the point of their call. A member function prototype defined within a class is declared without any special keyword.

C++ treats all the member functions that are defined within a class as *inline* functions and those defined outside as *non-inline* (outline). Member function declared outside the class declaration can be made inline by prefixing the `inline` to its definition as shown in Figure 10.11.

Keyword : indicates function defined outside a class body is inline

```

inline Returntype ClassName :: FunctionName (arguments)
{
    // body of Inline function
}

```

Figure 10.11: Inline function definition outside the class declaration

The keyword `inline` acts as a function qualifier. The modified program of `date2.cpp` is listed in `date3.cpp`, making all the member functions of the class `date` as inline member functions.

```
// date3.cpp: date class with member functions defined outside as inline
#include <iostream.h>
class date
{
    // specifies a structure
private:
    int day;
    int month;
    int year;
public:
    void set( int DayIn, int MonthIn, int YearIn ); //declaration
    void show(); // declaration
};
inline void date::set( int DayIn, int MonthIn, int YearIn )
{
    day = DayIn;
    month = MonthIn;
    year = YearIn;
}
inline void date::show() // definition
{
    cout << day << "-" << month << "-" << year << endl;
}
void main()
{
    date d1, d2, d3; // date objects d1, d2, and d3 creation
    // set date of births
    d1.set( 26, 3, 1958 );
    d2.set( 14, 4, 1971 );
    d3.set( 1, 4, 1972 );
    cout << "Birth Date of the First Author: ";
    d1.show();
    cout << "Birth Date of the Second Author: ";
    d2.show();
    cout << "Birth Date of the Third Author: ";
    d3.show();
}
```

Run

```
Birth Date of the First Author: 26-3-1958
Birth Date of the Second Author: 14-4-1971
Birth Date of the Third Author: 1-4-1972
```

In the above program, the member functions `set()` and `show()` of the class `date` are considered as inline member functions defined outside the body of the class `date`. They are explicitly defined as inline functions with the use of the *inline* qualifier. The use of the *inline* qualifier in the statements

```
inline void date::set( int DayIn, int MonthIn, int YearIn )
inline void date::show()
```

inform the compiler to treat the member functions `set` and `show` as inline functions. The method of invoking inline member functions is the same as those of the normal functions. In `main()`, the statements

```
d1.set( 26, 3, 1958 );
d2.show();
```

will be replaced by the function itself since the function is an inline function. Note that, the inline qualifier is tagged to the inline member function at the point of its definition.

The feature of inline member functions is useful only when they are short. Declaring a function having many statements as `inline` is not advisable, since it will make the object code of a program very large. However, some C++ compilers judge (determine) whether a given function can be appropriately sized to *inline expanded*. If the function is too large to be expanded, it will not be treated as inline. In this case, declaring a function inline will not guarantee that the compiler will consider it as an inline function.

When to Use inline Functions

The following are simple thumb rules in deciding as to when inline functions should be used:

- ◆ In general, inline functions should not be used.
- ◆ Defining inline functions can be considered once a fully developed and tested program runs too slowly and shows *bottlenecks* in certain functions. A *profiler* (which runs the program and determines where most of the execution time is spent) can be used in deciding such an optimization.
- ◆ Inline functions can be used when member functions consist of one very simple statement such as the return statement in `date::getday()`, which can be implemented as follows:

```
inline int date::getday() // definition
{
    return day;
}
```

- ◆ It is only useful to implement an inline function if the time spent during a function call is more compared to the function body execution time. An example, where an inline function has no effect at all is the following:

```
inline void date::show() // definition
{
    cout << day << "-" << month << "-" << year << endl;
}
```

The above function, which is presumed to be a member of the class `date` for the sake of argument, contains only one statement; but takes relatively a long time to execute. In general, functions which perform input and output operation spend a considerable amount of time. The effect of conversion of the function `show()` to inline would lead to reduction in execution time.

Inline functions have one disadvantage: the actual code is inserted by the compiler and therefore it should be known at compile-time. Hence, an inline function cannot be located in a run-time library. Practically, an inline function is placed near the declaration of a class, usually in the same header file. It results in a header file having the declaration of a class with its implementation visible to the user.

10.7 Accessing Member Functions within the Class

A member function of a class is accessed by the objects of that class using the dot operator. A member function of a class can call any other member function of its own class irrespective of its privilege and this situation is called *nesting* of member functions. The method for calling member functions of one's own class is similar to calling any other standard (library) functions as illustrated in the program `nesting.cpp`.

```
// nesting.cpp: A member function accessing another member function
#include <iostream.h>
class NumberPairs
{
    int num1, num2;      // private by default
public:
    void read()
    {
        cout << "Enter First Number: ";
        cin >> num1;
        cout << "Enter Second Number: ";
        cin >> num2;
    }
    int max()           // member function
    {
        if( num1 > num2 )
            return num1;
        else
            return num2;
    }
    // Nesting of member function
    void ShowMax()
    {
        // calls member function max()
        cout << "Maximum = " << max();
    }
};
void main()
{
    NumberPairs n1;
    n1.read();
    n1.ShowMax();
}
```

Run

```
Enter First Number: 5
Enter Second Number: 10
Maximum = 10
```

The class `NumberPairs` has the member function `ShowMax()` having the statement

```
cout << "Maximum = " << max();
```

It calls the member function `max()` to compute the maximum of class data members `num1` and `num2`.

10.8 Data Hiding

Data is hidden inside a class, so that it cannot be accessed even by mistake by any function outside the class, which is a key feature of OOP. C++ imposes a restriction to access both the data and functions of a class. It is achieved by declaring the data part as *private*. All the data and functions defined in a class are private by default. But for the sake of clarity, the items are declared as private explicitly. Normally, data members are declared as *private* and member functions are declared as *public*. This is illustrated in the program `part.cpp`.

```
// part.cpp: class hiding vehicle details
#include <iostream.h>
class part
{
private:          // private members
    int ModelNum; // model number
    int PartNum;  // part number
    float cost;   // cost of a part
public:          // public members
    void SetPart( int mn, int pn, float c )
    {
        ModelNum = mn;
        PartNum = pn;
        cost = c;
    }
    void ShowPart()
    {
        cout << "Model: " << ModelNum << endl;
        cout << "Number: " << PartNum << endl;
        cout << "Cost: " << cost << endl;
    }
};
void main()
{
    part p1, p2; // objects p1 and p2 of class part are defined
    // Values are passed to their object
    p1.SetPart( 1996, 23, 1250.55 );
    p2.SetPart( 2000, 243, 2354.75 );
    // Each object display their values
    cout << "First Part Details ..." << endl;
    p1.ShowPart();
    cout << "Second Part Details ..." << endl;
    p2.ShowPart();
}
```

Run

```
First Part Details ...
Model: 1996
Number: 23
Cost: 1250.550049
Second Part Details ...
Model: 2000
```

Number: 243
 Cost: 2354.75

In the above program, the data fields `ModelNum`, `PartNum`, and `cost` of the class `part` cannot be accessed by direct references using `p1.ModelNum`, `p1.PartNum`, and `p1.cost` respectively. When a class is used, its declaration must be available. Thus, a user of the class is presented with a description of the class. The internal details of the class, which are not essential to the user are not presented to him. This is the concept of *information hiding* or *data encapsulation*. As far as the user is concerned, the knowledge of accessible data and member functions of a class is enough. These interfaces, usually called the *user interface methods*, specify their abstracted functionality. Thus, to the user, a class is like a black box with a characterized behavior.

The purpose of data encapsulation is to prevent accidental modification of information of a class. It is achieved by imposing a set of rules—the manner in which a class is to be manipulated and the data and functions of the class can be accessed. The following are the three kinds of users of a class:

- ◆ A class member, which can access all the data members and functions of its class.
- ◆ Generic users, which define the instance of a class.
- ◆ Derived classes, which can access members based on privileges.

Each user has different access privileges to the object. A class differentiates between access privileges by partitioning its contents and associating each one of them with any one of the following keywords:

- ◆ `private`
- ◆ `public`
- ◆ `protected`

These keywords are called *access-control specifiers*. All the members that follow a keyword (upto another keyword) belong to that type. If no keyword is specified, then the members are assumed to have private privilege. The following specification of a class illustrates these concepts:

```
class PiggyBank
{
    int Money;                // Private by default
    void Display()           // Private by default
    {
        ...
    }
private:                    // Private by declaration
    int AccNumber;
public:
    int code;                // Public
    void SetData(int a, int b) // Public
    {
        ...
    }
protected:
    int PolicyCode;         // Protected
    void GetPolicyCode()    // Protected
    {
        ...
    }
};
```





In the above declaration, the members `Money`, `AccNumber`, and `Display()` will be of type `private`; the members `code` and `SetData()` will be of type `public`; and the members `PolicyCode` and `GetData()` will be of type `protected`.



Data hiding is mainly designed to protect well-intentioned programmers from honest mistakes. It protects access to the data according to the design decision made while designing a class. Programmers who really want to figure out a way to access highly protected data such as `private`, will find it hard to do so even by accident. *There are mechanisms to access even private data using friends, pointer to members, etc. from outside the class.*

Private Members

The private members of a class have strict access control. Only the member functions of the same class can access these members. The private members of a class are inaccessible outside the class, thus, providing a mechanism for preventing accidental modifications of the data members. It is illustrated in Figure 10.12. Strictly speaking, information hiding is implemented only partially, the private members can still be accessed. Access control in C++ has the objective of reducing the likelihood of bugs and enhancing consistency. Since the basic intention of declaring a class is to use it in a program, the class should have at least one member that is not `private`.

```

class Person
{
    private :  Note: colon here
               access specifier
              // private members
    .....
    int age;  private data
    int getage();  private function
    .....
};

Person p1;
a=p1.age;  cannot access private data
p1.getage();  cannot access private function

```

Figure 10.12: Private members accessibility

The following example illustrates the situation when all the members of a class are declared as `private`:

```

class Inaccessible
{
    int x;
    void Display()
    {
        cout << "\nData = " << x;
    }
};

void main()
{
    Inaccessible obj1;           // Creating an object.
    obj1.x = 5;                 // Error: Invalid access.
    obj1.Display();             // Error: Invalid access.
}

```

The class having all the members with private access control is of no use; there is no means available to communicate with the external world. Therefore, classes of the above type will not contribute anything to the program.

Protected Members

The access control of the protected members is similar to that of private members and has more significance in inheritance. Hence, detailed discussion on this is postponed to the chapter on *Inheritance*. Access control of protected members is shown in Figure 10.13.

```
class Person
{
    protected:           ← Note: colon here
    // protected members
    .....
    int age;             ← protected data
    int getage();        ← protected function
    .....
};
Person p1;
a=p1.age;
p1.getage(); } → cannot access protected member
                ( same as private )
```

Figure 10.13: Protected members accessibility

Public Members

The members of a class, which are to be visible (accessible) outside the class, should be declared in *public* section. All data members and functions declared in the public section of the class can be accessed without any restriction from anywhere in the program, either by functions that belong to the class or by those external to the class. Accessibility control of public members is shown in Figure 10.14.

```
class Person
{
    public:              ← Note: colon here
    // public members
    .....
    int age;             ← public data
    int getage();        ← public function
    .....
};
Person p1;
a=p1.age; ✓ can access public data
p1.getage(); ✓ can access public function
```

Figure 10.14: Public members accessibility

10.9 Access Boundary of Objects Revisited

Hierarchy of access, in which privilege code can see the whole structure of an object, but external code can see only the public features. The access-limit of members within a class, or from objects of a class is shown in Table 10.1 and Figure 10.15.

Access Specifier	Accessible to	
	Own class Members	Objects of a Class
private:	Yes	No
protected:	Yes	No
public:	Yes	Yes

Table 10.1: Visibility of class members

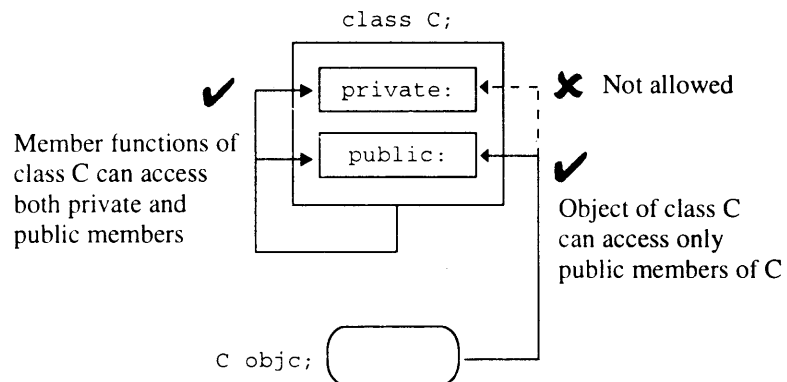


Figure 10.15: Class member accessibility

The following declaration of a class illustrates the visibility limit of the various class members:

```
class MyClass
{
    private:
        int a;
        void f1()
        {
            //can refer to data members a, b, c and functions f1, f2, and f3
        }
    protected:
        int b;
        void f2()
        {
            //can refer to data member a, b, c and functions f1, f2, and f3
        }
    public:
        int c;
        void f3()
        {
            //can refer to data member a, b, c and functions f1, f2, and f3
        }
}
```

Consider the statements

```
MyClass objx; // objx is an object of class MyClass
int d;       // temporary variable d
```

They define an object `objx` and an integer variable `d`. The accessibility of members of the class `MyClass` through the object `objx` is illustrated in the following section.

1. Accessing private members of the class `MyClass`:

```
d = objx.a; // Error: 'MyClass::a' is not accessible
objx.f1(); // Error: 'MyClass::f1()' is not accessible
```

Both the statements are invalid because the private members of the class are inaccessible.

2. Accessing protected members of the class `MyClass`:

```
d = objx.b; // Error: 'MyClass::b' is not accessible
objx.f2(); // Error: 'MyClass::f2()' is not accessible
```

Both the statements are invalid because the protected members of the class are inaccessible.

3. Accessing public members of the class `MyClass`:

```
d = objx.c; // OK
objx.f3(); // OK
```

Both the statements are valid because the public members of the class are accessible.

10.10 Empty Classes

Although the main reason for using a class is to encapsulate data and code, it is however, possible to have a class that has neither data nor code. In other words, it is possible to have empty classes. The declaration of empty classes is as follows:

```
class xyz { };
class Empty { };
class abc
{
};
```

During the initial stages of development of a project, some of the classes are either not fully identified, or not fully implemented. In such cases, they are implemented as empty classes during the first few implementations of the project. Such empty classes are also called as *stubs*. The significant usage of empty classes can be found with exception handling; it is illustrated in the chapter *Exception Handling*.

10.11 Pointers within a Class

The size of data members such as vectors when defined using arrays must be known at compile time itself. In this case, vector size cannot be increased or decreased irrespective of the requirement. This inflexibility of arrays can be overcome by having a data member for storing vector elements whose size can be dynamically changed during runtime. The program `vector.cpp` facilitates the creation of the vector of varying size during runtime. It has a pointer member instead of an array member. The size of the vector is varied by creating an object whose vector size is known only at runtime.

```
/* vector.cpp: vector class with array dynamically allocated
#include <iostream.h>
class vector
```

```

    int *v; // pointer to a vector
    int sz; // size of a vector
public:
    void VectorSize( int size ) // allocate memory dynamically
    {
        sz = size;
        v = new int[ size ]; // dynamically allocate vector
    }
    void read();
    void show_sum();
    void release() // release memory allocated
    {
        delete v;
    }
};
void vector::read()
{
    for( int i = 0; i < sz; i++ )
    {
        cout << "Enter vector[ " << i << " ] ? ";
        cin >> v[i];
    }
}
void vector::show_sum()
{
    int sum = 0;
    for( int i = 0; i < sz; i++ )
        sum += v[i];
    cout << "Vector Sum = " << sum;
}
void main()
{
    vector v1;
    int count;
    cout << "How many elements are there in vector: ";
    cin >> count;
    v1.VectorSize( count ); // set vector size
    v1.read();
    v1.show_sum();
    v1.release(); // free vector resources
}

```

Run

```

How many elements are there in vector: 5
Enter vector[ 0 ] ? 1
Enter vector[ 1 ] ? 2
Enter vector[ 2 ] ? 3
Enter vector[ 3 ] ? 4
Enter vector[ 4 ] ? 5
Vector Sum = 15

```

In `main()`, the statement

```
vector v1;
```

creates an object `v1` of the class `vector` and the statement

```
v1.VectorSize( count );    // set vector size
```

allocates the required amount (specified by the parameter `count`) of memory, dynamically for vector elements storage. The last statement

```
v1.release();
```

releases the memory allocated to the pointer data member `v` of the `vector` class. The operation of dynamic allocation of memory to data members can be at best realized by defining *constructor* and *destructor* functions. (More details can be found in the chapter *Object Initialization and Cleanup*).

10.12 Passing Objects as Arguments

It is possible to have functions which accept objects of a class as arguments, just as there are functions which accept other variables as arguments. Like any other data type, an object can be passed as an argument to a function by the following ways:

- ♦ pass-by-value, a copy of the entire object is passed to the function
- ♦ pass-by-reference, only the address of the object is passed implicitly to the function
- ♦ pass-by-pointer, the address of the object is passed explicitly to the function

In the case of *pass-by-value*, a copy of the object is passed to the function and any modifications made to the object inside the function is not reflected in the object used to call the function. Whereas, in *pass-by-reference* or *pointer*, an address of the object is passed to the function and any changes made to the object inside the function is reflected in the actual object. The parameter passing by reference or pointer is more efficient since, only the address of the object is passed and not a copy of the entire object.

Passing Objects by Value

The program `distance.cpp` illustrates the use of objects as function arguments in *pass-by-value* mechanism. It performs the addition of distance in feet and inches format.

```
// distance.cpp: distance manipulation in feet and inches
#include <iostream.h>
class distance
{
private:
    float feet;
    float inches;
public:
    void init( float ft, float in )
    {
        feet = ft;
        inches = in;
    }
    void read()
    {
        cout << "Enter feet: ";   cin >> feet;
        cout << "Enter inches: "; cin >> inches;
    }
}
```

```

void show()
{
    cout << feet << "-" << inches << '\n';
}
void add( distance d1, distance d2 )
{
    feet = d1.feet + d2.feet;
    inches = d1.inches + d2.inches;
    if( inches >= 12.0 )
    {
        // 1 foot = 12.0 inches
        feet = feet + 1.0;
        inches = inches - 12.0;
    }
}
};
void main()
{
    distance d1, d2, d3;
    d2.init( 11.0, 6.25 );
    d1.read();
    cout << "d1 = "; d1.show();
    cout << "\nd2 = "; d2.show();
    d3.add( d1, d2 ); // d3 = d1 + d2
    cout << "\nd3 = d1+d2 = "; d3.show();
}

```

Run

```

Enter feet: 12.0
Enter inches: 7.25
d1 = 12'-7.25"
d2 = 11'-6.25"
d3 = d1+ d2 = 24'-1.5"

```

In main(), the statement

```
d3.add( d1, d2 ); // d3 = d1 + d2
```

invokes the member function add() of the class distance by the object d3, with the object d1 and d2 as arguments. It can directly access the feet and inches variables of d3. The members of d1 and d2 can be accessed only by using the dot operator (like d1.feet and d1.inches) within the add() member. Figure 10.16 shows the two objects d1 and d2 being added together with the result stored in the recipient object d3. Any modification made to the data members of the objects d1 and d2 are not visible to the caller's actual parameters.

Passing Objects by Reference

Accessibility of the objects passed by reference is similar to those passed by value. Modifications carried out on such objects in the called function will also be reflected in the calling function. The method of passing objects as reference parameters to a function is illustrated in the program account.cpp. Given the account numbers and the balance of two accounts, this program transfers a specified sum from one of these accounts to the other and then, updates the balance in both the accounts.

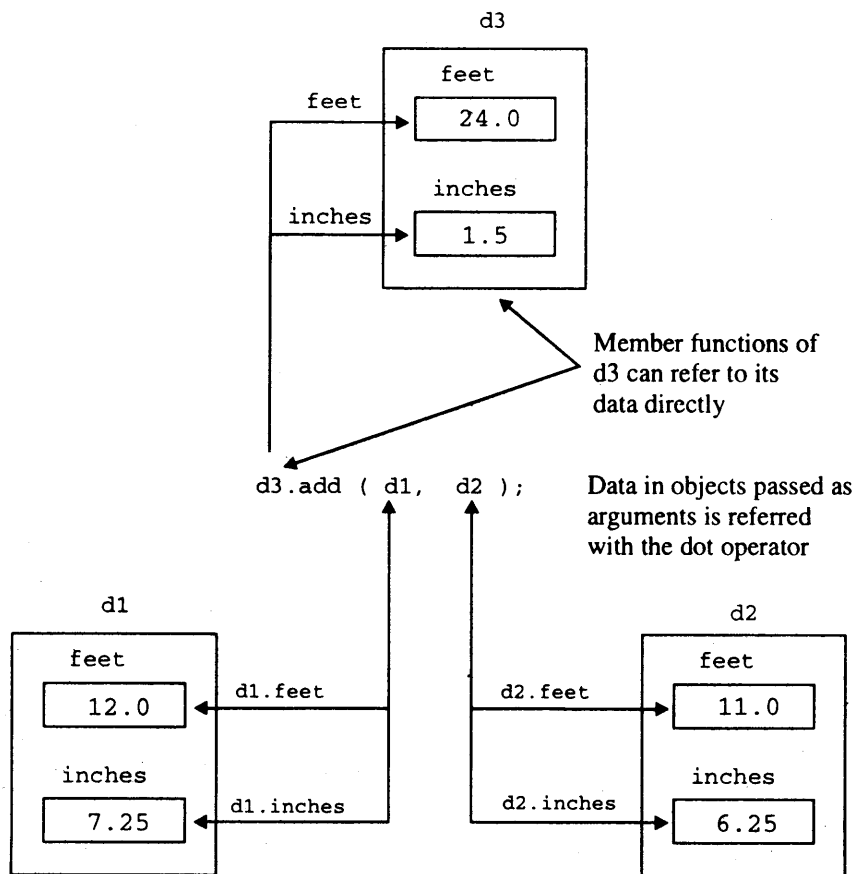


Figure 10.16: Objects of the distance class as parameters

```
// account.cpp: passing objects as parameters to functions
#include<iostream.h>
class AccClass
{
private:           // class data members
    int accno;
    float balance;
public:           // class function members
    void getdata()
    {
        cout << "Enter the account number for accl object: ";
        cin >> accno;
        cout << "Enter the balance: ";
        cin >> balance;
    }
}
```

```

void setdata( int accIn )
{
    accno = accIn;
    balance = 0;
}
void setdata( int accIn, float balanceIn )
{
    accno = accIn;
    balance = balanceIn;
}
void display()
{
    cout << "Account number is: " << accno << endl;
    cout << "Balance is: " << balance << endl;
}
void MoneyTransfer( AccClass & acc, float amount );
};
// acc1.MoneyTransfer( acc2, 100 ),transfers 100 rupees from acc1 to acc2
void AccClass::MoneyTransfer( AccClass & acc, float amount )
{
    balance = balance - amount; // deduct money from source
    acc.balance = acc.balance + amount; // add money to destination
}
void main()
{
    int trans_money;
    AccClass acc1, acc2, acc3;
    acc1.getdata();
    acc2.setdata( 10 );
    acc3.setdata( 20, 750.5 );
    cout << "Account Information..." << endl;
    acc1.display();
    acc2.display();
    acc3.display();
    cout << "How much money is to be transferred from acc3 to acc1: ";
    cin >> trans_money;
    acc3.MoneyTransfer(acc1,trans_money); //transfers money from acc3 to acc1
    cout << "Updated Information about accounts..." << endl;
    acc1.display();
    acc2.display();
    acc3.display();
}

```

Run

```

Enter the account number for acc1 object: 1
Enter the balance: 100
Account Information...
Account number is: 1
Balance is: 100
Account number is: 10
Balance is: 0

```

340 Mastering C++

```
Account number is: 20
Balance is: 750.5
How much money is to be transferred from acc3 to acc1: 200
Updated Information about accounts...
Account number is: 1
Balance is: 300
Account number is: 10
Balance is: 0
Account number is: 20
Balance is: 550.5
```

In main(), the statement

```
acc3.MoneyTransfer( acc1, trans_money );
```

transfers the object acc1 by reference to the member function MoneyTransfer(). It is to be noted that when the MoneyTransfer() is invoked with acc1 as the object parameter, the data members of acc3 are accessed without the use of the class member access operator, while the data members of acc1 are accessed by using their names in association with the name of the object to which they belong. An object can also be passed to a non-member function of the class and that can have access to the public members only through the objects passed as arguments to it.

Passing Objects by Pointer

The members of objects passed by pointer are accessed by using the -> operator, and they have similar effect as those passed by value. The above program requires the following changes if parameters are to be passed by pointer:

1. The prototype of the member function MoneyTransfer() has to be changed to:

```
void MoneyTransfer( AccClass * acc, float amount );
```

2. The definition of the member function MoneyTransfer() has to be changed to:

```
void AccClass::MoneyTransfer( AccClass & acc, float amount )
{
    balance = balance - amount; // deduct money from source
    acc->balance = acc->balance + amount; // add money to destination
}
```

3. The statement invoking the member function MoneyTransfer() has to be changed to:

```
acc3.MoneyTransfer( &acc1, trans_money );
```

10.13 Returning Objects from Functions

Similar to sending objects as parameters to functions, it is also possible to return objects from functions. The syntax used is similar to that of returning variables from functions. The return type of the function is declared as the return object type. It is illustrated in the program complex.cpp.

```
// complex.cpp: Addition of Complex Numbers, class complex as data type
#include <iostream.h>
#include <math.h>
class complex
{
private:
    float real; // real part of complex number
    float imag; // imaginary part of complex number
```



```

public:
    void getdata()
    {
        cout << "Real Part ? ";
        cin >> real;
        cout << "Imag Part ? ";
        cin >> imag;
    }
    void outdata( char *msg )    // display number in x+iy form
    {
        cout << msg << real;
        if( imag < 0 )
            cout << "-i";
        else
            cout << "+i";
        cout << fabs(imag) << endl;
    }
    complex add( complex c2 );    // addition of complex numbers
};
complex complex::add( complex c2 ) // add default and c2 objects
{
    complex temp;                // object temp of complex class
    temp.real = real + c2.real;   // add real parts
    temp.imag = imag + c2.imag;   // add imaginary parts
    return( temp );              // return complex object
}
void main()
{
    complex c1, c2, c3;          // c1, c2, and c2 are objects of complex
    cout << "Enter Complex Number c1 .." << endl;
    c1.getdata();
    cout << "Enter Complex Number c2 .." << endl;
    c2.getdata();
    c3 = c1.add( c2 );           // add c1 and c2 assign to c3
    c3.outdata( "c3 = c1.add( c2 ): ");
}

```

Run

```

Enter Complex Number c1 ..
Real Part ? 1.5
Imag Part ? 2
Enter Complex Number c2 ..
Real Part ? 3
Imag Part ? -4.3
c3 = c1.add( c2 ): 4.5-i2.3

```

In main(), the statement
`c3 = c1.add(c2);` // add c1 and c2 assign to c3
invokes the function add() of the class complex by passing the object c2 as a parameter. The statement in this function,
`return(temp);` // return complex object
returns the object temp as a return object.

10.14 Friend Functions and Friend Classes

The concept of encapsulation and data hiding dictate that non-member functions should not be allowed to access an object's private and protected members. The policy is, *if you are not a member you cannot get it*. Sometimes this feature leads to considerable inconvenience in programming. Imagine that the user wants a function to operate on objects of two different classes. At such times, it is required to allow functions outside a class to access and manipulate the private members of the class. In C++, this is achieved by using the concept of *friends*.

One of the convenient and a controversial feature of C++ is allowing non-member functions to access even the private members of a class using friend functions or friend classes. It permits a function or all the functions of another class to access a different class's private members. The accessibility of class members in various forms is shown in Figure 10.17.

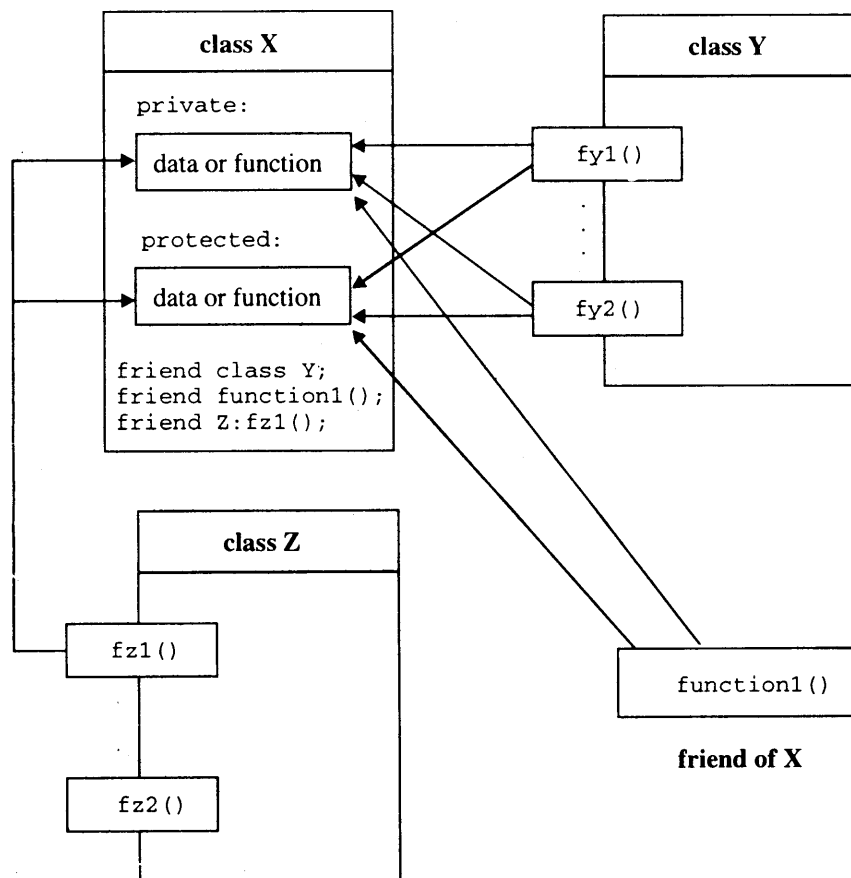


Figure 10.17: Class members accessibility in various forms

The function declaration must be prefixed by the keyword `friend` whereas the function definition must not. The function could be defined anywhere in the program similar to any normal C++ function. The functions that are declared with the keyword `friend` are called *friend functions*. A function can be a friend to multiple classes. A friend function possesses the following special characteristics:

- ◆ The scope of a friend function is not limited to the class in which it has been declared as a friend.
- ◆ A friend function cannot be called using the object of that class; it is not in the scope of the class. It can be invoked like a normal function without the use of any object.
- ◆ Unlike class member functions, it cannot access the class members directly. However, it can use the object and the dot operator with each member name to access both the private and public members.
- ◆ It can be either declared in the private part or the public part of a class without affecting its meaning.

Consider the following skeleton of the program code to illustrate friend functions.

```
class A
{
    private:
        int value; // value is private data
    public:
        void setval( int v )
        { value = v; }
        int getval ()
        { return( value ); }
};
// function decrement: tries to alter A's private data
void decrement( A &a )
{
    a.value--; // Error:: not allowed to access private data
}
class B // class B: tries to access A's private data
{
    public:
        void touch (A &a)
        { a.value++; }
};
```

This code will not compile, since the function `decrement()` and the function `touch()` of the class B attempt to access a private data member of the class A.

The function can be allowed explicitly to access A's data and class B members can be allowed to access the class A's data. To accomplish this, the offending classless function `decrement()` and the class B are declared to be friends of the class A as illustrated in the following code:

```
class A
{
    public:
        friend class B; // B is my friend, I trust him
        friend void decrement (A &what); // decrement() is also a good pal
};
```

Concerning friendship between classes, the following should be noted:

- ◆ Friendship is not mutual by default. That is, once B is declared as a friend of A, this does not give A the right to access the private members of the class B.
- ◆ Friendship, when applied to program design, is an escape mechanism which creates exceptions to the rule of data hiding. Usage of friend classes should, therefore, be limited to those cases where it is absolutely essential.

Bridging Classes with Friend Functions

Consider a situation of operating on objects of two different classes. In such a situation, friend functions can be used to bridge the two classes. It is illustrated in the program `friend1.cpp`. The syntax of defining friend non-member function is shown in Figure 10.18.

```

class Testclass
{
    int num1, num2;
    .....
    public:
        // public members
        friend float sum ( Testclass obj );
    .....
};

```

keyword

No friend keyword

No scope resolution operator, Testclass :: sum cannot be made

```

float sum (Testclass obj)
{
    float result;
    result = obj.num1 + obj.num2;
    return result;
}

```

private data member

Figure 10.18: Friend function of a class

```

// friend1.cpp: Normal function accessing object's private members
#include <iostream.h>

class two; // advance declaration like function prototype
class one
{
    private:
        int data1;
    public:
        void setdata( int init )
        {
            data1 = init;
        }
        friend int add_both( one a, two b ); // friend function
};

class two
{
    private:
        int data2;
    public:
        void setdata( int init )
        {
            data2 = init;
        }
        friend int add_both( one a, two b ); // friend function
};

```

```

// friend function of class one and two
int add_both( one a, two b )
{
    return a.data1 + b.data2; // a.data1 and b.data2 are private
}
void main()
{
    one a;
    two b;
    a.setdata( 5 );
    b.setdata( 10 );
    cout << "Sum of one and two: " << add_both( a, b );
}

```

Run

Sum of one and two: 15

The above program, contains two classes named `one` and `two`. To allow the normal function `add_both()` to have an access to private data members of objects of these classes, it must be declared as a friend function. It has been declared with the `friend` keyword in both the classes as:

```
friend int add_both( one a, two b );
```

This declaration can be placed either in the private or the public section of the class.

An object of each class has been passed as an argument to the function `add_both()`. Being a friend function, it can access the private members of both classes through these arguments.

Observe the following declaration at the beginning of the program

```
class two; // advance declaration like function prototype
```

It is necessary, since a class cannot be referred until it has been declared before the class `one`. It informs the compiler that the class `two`'s specification will appear later.

Though friend functions add flexibility to the language and make programming convenient in certain situations, they are controversial; it goes against the philosophy that only member functions can access a class's private data. Friend functions should be used sparingly. If a program uses many friend functions, it can easily be concluded that there is a basic flaw in the design of a program and it would be better to redesign such programs. However, friend functions are very useful in certain situations. One such example is when a friend is used to increase the versatility of overloaded operators, which will be discussed in the chapter *Operator Overloading*.



Friend functions are useful in the following situations:

- ◆ Function operating on objects of two different classes. This is the ideal situation where the friend function can be used to bridge two classes.
- ◆ Friend functions can be used to increase the versatility of overloaded operators.
- ◆ Sometimes, a friend allows a more obvious syntax for calling a function, rather than what a member function can do.

Friend Classes

Friend functions permit an exception to the rules of data encapsulation. The `friend` keyword allows a function, or all the functions of another class to manipulate the private members of the original class. The syntax of declaring *friend* class is shown in Figure 10.19.

```

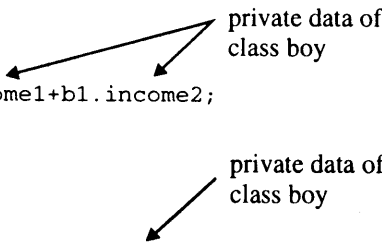
class boy
{
    private:  private specifier
        int income1;
        int income2;
    public:  public specifier
        int gettotal()
        {
            return income1 + income2;
        }

        friend class girl; //class girl can access private members
};

class girl
{
    // all the members of class girl can access attributes of boy
    .....
    public:
        int girlfunc(boy b1)
        {
            result = b1.income1+b1.income2;
            return result;
        }

        void show()
        {
            boy b1;
            cout << "Income1: " << b1.income1; // private data of boy
        }
};

```


Figure 10.19: girl class is a friend of class boy

All the member functions of one class can be friend functions of another class. The program friend2.cpp demonstrates the method of bridging classes using friend class.

```

// friend2.cpp: class girl is declared as a friend of class boy
#include <iostream.h>
// forward declaration of class girl; is optional
class boy
{
    private: // private members
        int income1;
        int income2;
    public:
        void setdata( int in1, int in2 )
        {
            income1 = in1;
            income2 = in2;
        }
        friend class girl; // class girl can access private data
};

```

```

class girl
{
    int income; // income is private data member
public:
    int girlfunc( boy b1 )
    {
        return b1.income1+b1.income2;
    }
    void setdata( int in )
    {
        income = in;
    }
    void show()
    {
        boy b1;
        b1.setdata( 100, 200 );
        cout << "boy's Income1 in show(): " << b1.income1 << endl;
        cout << "girl's income in show(): " << income << endl;
    }
};

void main()
{
    boy b1;
    girl g1;
    b1.setdata( 500, 1000 );
    g1.setdata( 300 );
    cout << "boy b1 total income: " << g1.girlfunc(b1) << endl;
    g1.show();
};

```

Run

```

boy b1 total income: 1500
boy's Income1 in show(): 100
girl's income in show(): 300

```

The statement in the class boy

```
friend class girl; // class girl can access private data members
```

declares that all the member functions of the class `girl` are friend functions of class `boy` but not the other way. (Thus in C++, class `girl`, the friend class of the class `boy`, does not mean that the class `boy` is the friend of the class `girl`). The objects of the class `girl` can access all the members of the class `boy` irrespective of their access privileges.

The function `show()` in the `girl` class

```
cout << "boy's Income1 in show(): " << b1.income1 << endl;
```

accesses the private data member `income1` of the `boy` class.

Class Friend to a Specified Class Member

When only specific member function of one class should be friend function of another class, it must be specified explicitly using the scope resolution operator as shown in Figure 10.20. The function `girlfunc()` is a member function of class `girl` and a friend of class `boy`.

```

class boy
{
    private:
        int income1;
        int income2;
    public:
        int gettotal()
        {
            return income1 + income2;
        }
};

friend girl :: girlfunc(boy b1); // class girl's girlfunc() is allowed to
// access data and functions of class boy

class girl
{
    public:
        int girlfunc( boy b1 )
        {
            result = b1.income1 + b1.income2;
            return result;
        }
        void show () // cannot access private members of boy
        {
            boy b1; // only public members can be accessed
        }
};

```

Annotations in the original image:

- A curved arrow points from the text "private specifier" to the `private:` keyword in the `boy` class.
- A curved arrow points from the text "public specifier" to the `public:` keyword in the `boy` class.
- A straight arrow points from the text "class name to which this function is a member" to the `girl` class name in the `friend` declaration.
- Two straight arrows point from the text "private data members of class boy" to the `income1` and `income2` variables in the `girlfunc` function of the `girl` class.

Figure 10.20: Member function to which class boy is a friend

In the class `girl`, only function `girlfunc()` is allowed to access the private data and functions of the class `boy`. So only this function could be specifically made a friend in the class `boy` as illustrated in the program `friend3.cpp`.

```

// friend3.cpp: specific member function class girl is friend of boy
#include <iostream.h>
class boy; // advance declaration like function prototype
class girl
{
    int income; // income is private data member
    public:
        int girlfunc( boy b1 );
        void setdata( int in )
        {
            income = in;
        }
        void show()
        {
            cout << "girl income: " << income;
        }
};

```



```

class boy
{
    private:          // private members
        int incomel;
        int income2;
    public:
        void setdata( int in1, int in2 )
        {
            incomel = in1;
            income2 = in2;
        }
        // only this function can access private data of boy
        friend int girl::girlfunc( boy b1 );
};
// only this function can access private data of the boy class
int girl::girlfunc( boy b1 )
{
    return b1.incomel+b1.income2;
}
void main()
{
    boy b1;
    girl g1;
    b1.setdata( 500, 1000 );
    g1.setdata( 300 );
    cout << "boy b1 total income: " << g1.girlfunc(b1) << endl;
    g1.show();
}

```

Run

```

boy b1 total income: 1500
girl income: 300

```

The null-body class declaration statement,

```
class boy; // advance declaration like function prototype
```

appears in the beginning of the program; a class cannot be referred until it has been declared before the class `girl`. It informs the compiler that the class `boy` is defined later. The statement in the class `boy`

```
friend int girl::girlfunc( boy b1 );
```

declares that only member function `girlfunc()` of the class `girl` can access private data and member functions of the class `boy`.

10.15 Constant Parameters and Member Functions

Certain member functions of a class, access the class data members without modifying them. It is advisable to declare such functions as `const` (constant) functions. The syntax for declaring `const` member functions is shown in Figure 10.21. A `const` member function is used to indicate that it does not alter the data fields of the object, but only inspects them.

Keyword

```
ReturnType FunctionName(arguments) const
```

Figure 10.21: Syntax of declaring a constant member function

A member function, which does not alter any data members in the class can be declared as `const` member function. The following statements illustrate the same:

```
void showname() const;
float divide() const;
```

The qualifier `const` is suffixed to the function in both the declaration and the definition. The compiler will generate an error message if such functions attempt to alter the class data members. The concept of constant member functions is illustrated in the program `constmem.cpp`.

```
// constmem.cpp: person class with const member functions
#include <iostream.h>
#include <string.h>
class Person
{
    private:
        char *name;           // name of person
        char *address;       // address field
        char *phone;        // telephone number
    public:
        void init();
        void clear();
        // functions to set fields
        void setname(char const *str);
        void setaddress(char const *str);
        void setphone(char const *str);
        // functions to inspect fields
        char const *getname(void) const;
        char const *getaddress(void) const;
        char const *getphone(void) const;
};
// initialize class data members to NULL
inline void Person::init()
{
    name = address = phone = 0;
}
// release memory allocated to class data members
inline void Person::clear()
{
    delete name;
    delete address;
    delete phone;
}
```