Pointer to Object Definition

Pointers can be used to hold addresses of objects, just as they can hold addresses of primitive and user-defined data items. The need for using pointers to objects becomes clear when objects are to be created while the program is being executed, which is an instance of dynamic allocation of memory. The `new` operator can also be used to obtain the address of the allocated memory area besides allocating storage area to the objects of the given class. Thus, the address returned by the `new` operator may be used to initialize a pointer to an object.

The general format for defining a pointer to an object is shown in Figure 12.1, which is similar to the way in which pointers to other data types are declared and defined. A pointer can be made to point to an existing object, or to a newly created object using the `new` operator. The address operator & can be used to get the address of an object, which is defined statically during the compile time. In the following statement

```
ptr_to_object = & object;
```

The `&` operator in the expression `&object` returns the address of the `object` and the same is initialized to a pointer variable `ptr_to_object`.

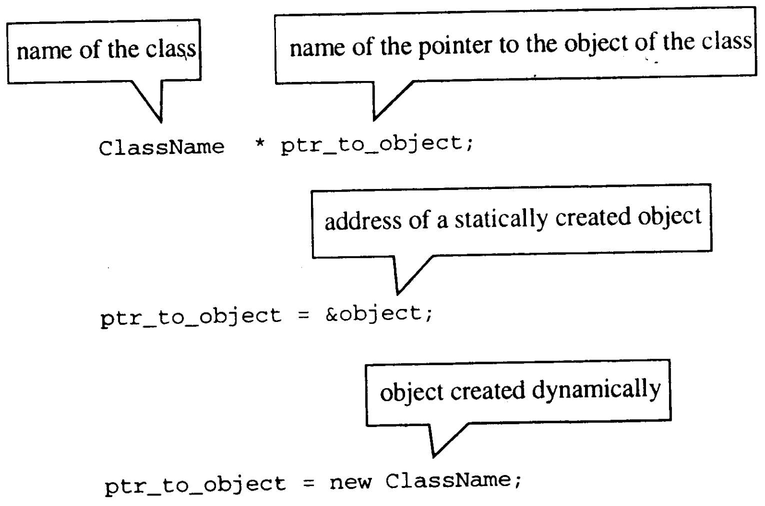

name of the class
name of the pointer to the object of the class
ClassName * ptr_to_object;
address of a statically created object
ptr_to_object = &object;
object created dynamically
ptr_to_object = new ClassName;

Figure 12.1: Syntax of defining pointer to object

Accessing Members of Objects

In order to utilize a pointer to an object, it is necessary to have some means by which the members of that object can be accessed and manipulated. As in the case of pointers to structures, there are two approaches to referring and accessing the members of an object whose address resides in a pointer. The operator -> can also be used to access member of an object using a pointer to objects. The expression to access a class member using pointer is as follows:

```
pointer_to_object -> member_name
```
or
```
*pointer_to_object.member_name
```

The member to be accessed through the object pointer can be either a data, or function member (see

Figure 12.2). The program `ptrobj1.cpp` illustrates the definition of pointers to objects and their usage in accessing members of a class.

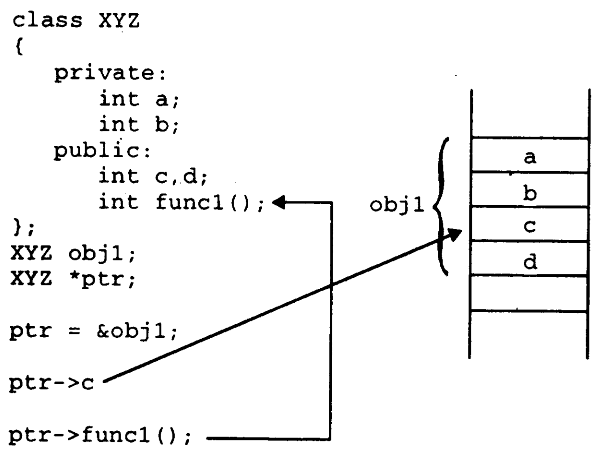


Figure 12.2: Pointer accessing class members

```
// ptrobj1.cpp: pointer to object, pointing to statically created objects
#include <iostream.h>
class someclass
{
   public:
      int data1;
      char data2;
      someclass()
      {
         cout << "Constructor someclass() is invoked\n";
         data1 = 1, data2 = 'A';
      }
      ~someclass()
      {
         cout << "Destructor ~someclass() is invoked\n";
      }
      void show()
      {
         cout << "data1 = " << data1;
         cout << " data2 = " << data2 << endl;
      }
};
void main(void)
{
   someclass *ptr;        // define a pointer to object of class someclass
   someclass object1;     // object of type someclass created statically
   ptr = &object1;
   cout << "Accessing object through object1.show()..." << endl;
```

```
    object1.show();
    cout << "Accessing object through ptr->show()..." << endl;
    ptr->show();    // it can be *ptr.show();
}
```

Run

```
Constructor someclass() is invoked
Accessing object through object1.show()...
data1 = 1 data2 = A
Accessing object through ptr->show()...
data1 = 1 data2 = A
Destructor ~someclass() is invoked
```

In `main()`, the statement,

```
ptr = &object1;
```

assigns the address of the object `object1` of the class `someclass` to the pointer `ptr`. The statement

```
ptr->show();
or
*ptr.show()
```

invokes the member function `show()` of the object pointed to by the pointer `ptr`. It points to the `object1`, and hence executes the function `show()` of the respective class.

Creating and Deleting Dynamic Objects

A dynamic object can be created by the execution of a new operator expression. The syntax for creating a dynamic object using the new operator is as follows:

new ClassName

It returns the address of a newly created object. The returned address of an object can be stored in a variable of type pointer to object (`ptr_to_object`) as follows:

ptr_to_object = new ClassName;

While creating a dynamic object, *if a class has the default constructor, it is invoked as a part of object creation activity.* Once a pointer is holding the address of a dynamic object, its members can be accessed by using -> operator.

The entity that executes the new expression is the dynamic object's creator. The creator may be a (member) function, an object, or a class. The creator of a dynamic object must be in a position to fully determine the object's lifetime. The creator cannot be inferred from the source code alone. Although, the creator is determined by the intent of the programmer, the language constrains the choice. In the program `ptrobj1.cpp`, the function `main()` is the creator of the object pointed to by variable `ptr_to_object` and hence, it is responsible for destroying it.

The syntax of the `delete` operator releasing memory allocated to dynamic object is as follows:

delete ptr_to_object;

It destroys the object pointed to by `ptr_to_object` variable. It also *invokes the destructor of the class if it exists as a part of object destruction activity* before releasing memory allocated to an object by the new operator.

The program ptrobj2.cpp illustrates the binding of dynamic objects' address to a pointer variable. The pointer defined is initialized with the address returned by the new operator, which actually creates the object.

```
// ptrobj2.cpp: pointer to object, pointing to dynamically created objects
# include <iostream.h>
class someclass
{
   public:
      int data1;
      char data2;
      someclass()
      {
         cout << "Constructor someclass() is invoked\n";
         data1 = 1, data2 = 'A';
      }
     ~someclass()
      {
         cout << "Destructor ~someclass() is invoked\n";
      }
      void show()
      {
         cout << "data1 = " << data1;
         cout << " data2 = " << data2 << endl;
      }
};
void main(void)
{
   someclass *ptr;        // define a pointer to object of class someclass
   cout << "Creating dynamic object..." << endl;
   ptr = new someclass;  // object created dynamically
   cout << "Accessing dynamic object through ptr->show()..." << endl;
   ptr->show();
   cout << "Destroying dynamic object..." << endl;
   delete ptr;          // object destroyed dynamically
}
```

Run

```
Creating dynamic object...
Constructor someclass() is invoked
Accessing dynamic object through ptr->show()...
data1 = 1 data2 = A
Destroying dynamic object...
Destructor ~someclass() is invoked
```

In main(), the statement

```
ptr = new someclass;  // object created dynamically
```

creates the *nameless object* of the class someclass dynamically and assigns its address to the object pointer ptr. It executes the constructor of the class someclass automatically during the creation of dynamic objects. The default argument constructor initializes the data members data1 and data2.

These data can be referenced by other member functions of its class. The statement

```
ptr->show();
```

invokes the member function `show()` of the object pointed to by the pointer variable `ptr`. It points to the object of the class `someclass` and hence, executes its member function `show()` as illustrated in Figure 12.3.

When the dynamic object pointed to by the variable `ptr` goes out of scope, the memory allocated to that object is not released automatically. It must be performed explicitly as follows:

```
delete ptr;
```

The above statement releases the memory allocated to the dynamically created object by the new operator. In addition to this, it also invokes the destructor function `~someclass()` to perform cleanup of resources allocated to the object's data members. In this class, object data members are not allocated with any resources dynamically and hence, no need to release them explicitly.

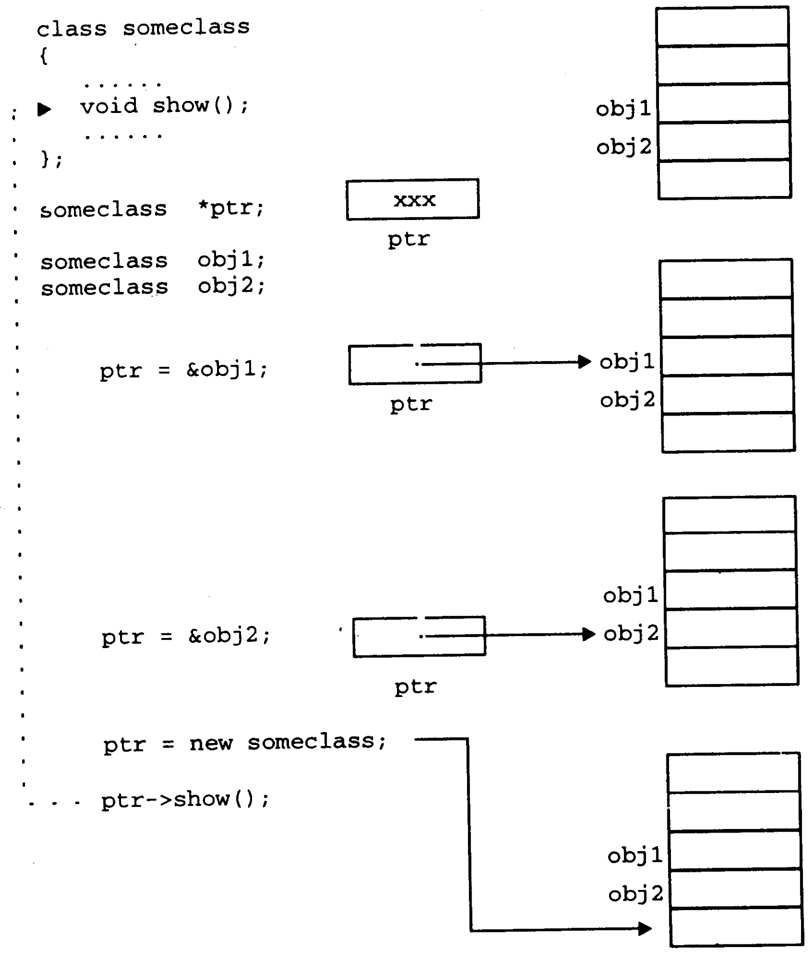
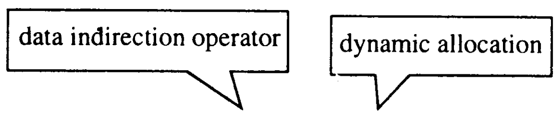


Figure 12.3: Object pointers and dynamic binding

Whenever it is necessary to determine the size of the memory area allocated to an object by the new operator, the `sizeof` operator may be used. For instance, the expression `sizeof(someclass)` returns the number of bytes required for the creation of an object of the class `someclass`.

Dereferencing Pointers

As the new operator returns a pointer to an area of memory that holds an object, it should be possible to refer to the original object by dereferencing the pointer. This method of memory allocation requires the use of both, the indirection operator * and the reference operator &. The general format for such a declaration is shown in Figure 12.4.



```
DataType & ReferenceVar=*(new DataType);
```

Figure 12.4: Syntax of dereferencing pointers

Such reference variables can be used like other variables without any special mechanism. The program `useref.cpp` illustrates the concept of binding reference variables at runtime.

```
// useref.cpp: Illustrates a variant usage of reference operator
#include <iostream.h>
void main(void)
{
   int & t1 = *(new int);   // Declares an integer variable using new
   int t2, t3;              // Regular int definitions
   t1 = t3 = 5;
   t2 = 10;
   t1 = t1 + t2;
   cout << "Sum of " << t3;  // Display old value of t1
   cout << " and " << t2 ;
   cout << " is : " << t1;   // Prints sum of t1 and t2
}
```

Run

```
Sum of 5 and 10 is : 15
```

Observe that the variable `t1` in the program is a variable of type reference to an integer. Also, the pointer returned by new is dereferenced, `*(new int)`, in order to refer to the original integer object which is finally associated with the reference variable `t1`. In the case of reference variables to class objects or structures, the members are accessed with the usual dot membership operator.

Reference to Dynamic Objects

The address of dynamic objects returned by the new operator can be dereferenced and reference to them can be created as follows:

```
ClassName &RefObj = *(new ClassName);
```

The reference to object `RefObj` can be used as a normal object; the memory allocated to such objects cannot be released except during the termination of the program. The program `refobj.cpp` illustrates the dereferencing of objects using reference pointers.

```
// refobj.cpp: reference to dynamic objects
#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 = *(new student); // reference to a dynamic object
   s1.setdata( 1, "Savithri" );
   s1.outdata();
   student &s2 = *(new student); // reference to a dynamic object
   s2.setdata( 2, "Bhavani" );
   s2.outdata();
   student s3;
   s3.setdata( 3, "Vani" );
   student &s4 = s3;     // reference to static object
   s3.outdata();
   s4.outdata();
}
```

Run

```
Roll No = 1
Name = Savithri
Roll No = 2
Name = Bhavani
Roll No = 3
Name = Vani
Roll No = 3
Name = Vani
```

In main(), the statement

```
student &s1 = *(new student);
```

creates a dynamic object of the class student and binds it to the reference variable s1. The expression *(new student) creates a dynamic object. The memory allocated to such objects cannot be released except during the termination of the program. The statement

```
s1.setdata( 1, "Savithri" );
```

accesses the member `setdata()` in the same way as normal objects accesses. The statement

```
student &s4 = s3;
```

creates the reference to normal object with the name `s4`. Note that, reference objects are accessed in the same way whether normal, or dynamic type objects.

12.3 Live Objects

The operator `new` allocates memory big enough to store an object and initializes it with the required data. Objects created dynamically with their data members initialized during creation are known as *live objects*. To create a live object, constructor must be invoked automatically which performs initialization of data members. Similarly, the destructor for an object must be invoked automatically before the memory for that object is deallocated. The syntax for creating a live object is as follows:

```
ptr_to_object = new ClassName( parameters )
```

A class whose live object is to be created must have *atleast one constructor*. The number of parameters passed specified at the point of creation of dynamic objects can be zero or more. If no arguments are specified, the default constructor (constructor with zero arguments) will be invoked automatically. If a class has more than one constructor, the constructor that matches with the parameters specified is invoked for initialization of the dynamic object. Note that there is no special syntax for releasing memory allocated to the objects, which are created and initialized by passing parameters. Hence, the syntax for destroying live objects is the same as that of normal dynamic objects.

The program `student3.cpp` illustrates the creation of *live* objects and their manipulation. It has a class called student having three constructor functions for initializing static or dynamic objects. The information required for initializing some dynamic objects is passed as parameters and some are initialized with information read at runtime.

```
// student3.cpp: manipulation of live objects
#include <iostream.h>
#include <string.h>
class student
{
   private:
      int roll_no;          // roll number
      char *name;           // name of a student
   public:
     // initializing data members using constructors
     student()   // constructor 0
     {
        char flag, str[50];
        cout << "Do you want to initialize the object (y/n): ";
        cin >> flag;
        if( flag == 'y' || flag == 'Y' )
        {
          cout << "Enter Roll no. of student: ";
          cin >> roll_no;
```

```
            cout << "Enter Name of student: ";
            cin >> str;
            name = new char[ strlen(str)+1 ];  // dynamic initialization
            strcpy( name, str );
         }
         else
         {
            roll_no = 0;
            name = NULL;
         }
      }
      student( int roll_no_in )  // constructor 1
      {
         roll_no = roll_no_in;
         name = NULL;
      }
      student( int roll_no_in, char *name_in )  // constructor 2
      {
         roll_no = roll_no_in;
         name = new char[ strlen(name_in)+1 ];
         strcpy( name, name_in );
      }
      ~student()
      {
         if( name )
            delete name;   // release memory allocated to name member
      }
      void set( int roll_no_in, char *name_in )
      {
         student( roll_no_in, name_in );
      }
      // display data members on the console screen
      void show()
      {
         if( roll_no )  // if( roll_no != 0 )
            cout << "Roll No: " << roll_no << endl;
         else
            cout << "Roll No: (not initialized)" << endl;
         if( name )  // if( name != NULL )
            cout << "Name: " << name << endl;
         else
            cout << "Name: (not initialized)" << endl;
      }
};
void main()
{
   student *s1, *s2, *s3, *s4;
   s1 = new student;    // will be initialized during run time by the user
   s2 = new student;    // will be initialized during run time by the user
   s3 = new student( 1 );  // partially live object
   s4 = new student( 2, "Bhavani" ); // fully live object
   cout << "Live objects contents..." << endl;
   // display contents of all live objects
   s1->show();
   s2->show();
   s3->show();
   s4->show();
   // release the memory allocated to dynamic objects s1, s2, s3, and s4
   delete s1;
   delete s2;
   delete s3;
   delete s4;
}
```

Run

```
Do you want to initialize the object (y/n): n
Do you want to initialize the object (y/n): y
Enter Roll no. of student: 5
Enter Name of student: Rekha
Live objects contents...
Roll No: (not initialized)
Name: (not initialized)
Roll No: 5
Name: Rekha
Roll No: 1
Name: (not initialized)
Roll No: 2
Name: Bhavani
```

In `main()`, the statement

```
        student *s1, *s2, *s3, *s4;
```

creates pointer variables to objects of the class `student`. The statements

```
        s1 = new student;
        s2 = new student;
```

create two objects dynamically and store their addresses in the variable `s1` and `s2` respectively. These objects are initialized by invoking the default constructor which reads the data entered by the user at runtime. The statement

```
        s3 = new student( 1 );
```

creates an object and initializes its first data member by invoking the one-argument constructor. The object `s3` is partially initialized object. The statement

```
        s4 = new student( 2, "Bhavani" );
```

creates an object named `s4` and initializes all its data members by invoking the two-argument constructor. The member function `show()` of the class `student` is invoked for all the objects pointed to by `s1`, `s2`, `s3`, and `s4` to display students' roll number and their name. All the objects created in this program are destroyed explicitly by using `delete` operator. The destructor is invoked automatically for each one of these objects to release the memory allocated to their string data member `name`. For instance, the statement,

```
   delete s2;
```

releases the memory allocated to the object pointed to by s2 and also invokes the destructor to cleanup.

12.4 Array of Objects

C++ allows the user to create an array of any data type including user-defined data types. Thus, an array of variables of a class data type can also be defined, and such variables are called an array of objects. An array of objects is often used to handle a group of objects, which reside contiguously in the memory. Consider the following class specification:

```
class student
{
  private:
    int roll_no;          // roll number
    char name[ 20 ];      // name of a student
  public:
    void setdata( int roll_no_in, char *name_in );
    void outdata();
};
```

The identifier `student` is a user-defined data type and can be used to create objects that relate to students of different courses. The following definition creates an array of objects of the `student` class:

```
student science[10]; // array of science course students
student medical[5];  // array of medical course students
student engg[25];    // array cf engineering course students
```

The array `science` contains ten objects, namely `science[0]`, ..., `science[9]` of type student class, the `medical` array contains 5 objects and the `engg` array contains 25 objects.

An array of objects is stored in the memory in the same way as a multidimensional array created at compile time. The representation of an array of engg objects is shown in Figure 12.5. Note that, only the memory space for data members of the objects is created; member functions are stored separately and shared by all the objects of `student` class.

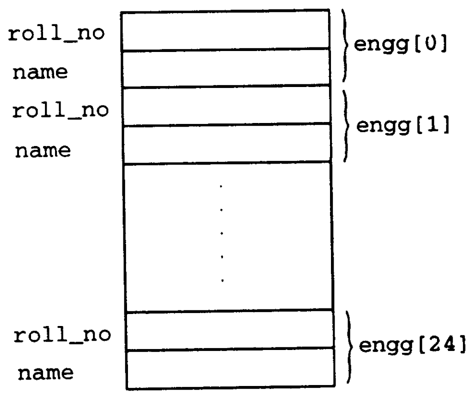


Figure 12.5: Storage for data items in an array of objects

An array of objects behaves similar to any other data-type array. The individual element of an array of objects is referenced by using its index, and member of an object is accessed using the *dot* operator.

For instance, the statement

```
        engg[i].setdata( 10, "Rajkumar" );
```

sets the data members of the i^{th} element of the array engg. Similarly, the statement

```
        engg[i].outdata();
```

will display the data of the i^{th} element of the array `engg[i]`. The program `student1.cpp` illustrates the use of the array of objects.

```
// student1.cpp: array of student data type
#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()
{
   int i, roll_no, count;
   char response, name[20];
   student s[10]; // array of 10 objects
   count = 0;
   for( i = 0; i < 10; i++ )
   {
      cout << "Initialize student object (y/n): ";
      cin >> response;
      if( response == 'y' || response == 'Y' )
      {
        cout << "Enter Roll no. of student: ";
        cin >> roll_no;
        cout << "Enter Name of student: ";
        cin >> name;
        s[i].setdata( roll_no, name );
        count++;
      }
      else
         break;
   }
   cout << "Student details..." << endl;
   for( i = 0; i < count; i++ )
      s[i].outdata();
}
```

Run

```
Initialize student object (y/n): y
Enter Roll no. of student: 1
Enter Name of student: Rajkumar
Initialize student object (y/n): y
Enter Roll no. of student: 2
Enter Name of student: Tejaswi
Initialize student object (y/n): y
Enter Roll no. of student: 3
Enter Name of student: Savithri
Initialize student object (y/n): n
Student details...
Roll No = 1
Name = Rajkumar
Roll No = 2
Name = Tejaswi
Roll No = 3
Name = Savithri
```

In `main()`, the statement

```
student s[10];
```

creates an array of 10 possible objects of the `student` class. It should be clearly understood that an array of objects allow better organization of the program instead of having 10 different variables and each one of them is the object of the `student` class. Note that the subscripted notation used for object is similar to the manner in which arrays of other data types are usually handled. The statement

```
s[i].outdata();
```

executes the `outdata()` member function in the `student` class for the `i`th object of the `s` array.

12.5 Array of Pointers to Objects

An array of pointers to objects is often used to handle a group of objects, which need not necessarily reside contiguously in memory, as in the case of a static array of objects. This approach is more flexible, in comparison with placing the objects themselves in an array, because objects could be dynamically created as and when they are required. The syntax for defining an array of pointers to objects is the same as any of the fundamental types. The program `student2.cpp` illustrates the concept of array of pointers to objects.

```
// student2.cpp: array of pointers to student
#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()
{
   int i, roll_no, count;
   char response, name[20];
   student * s[10];   // array of pointers to objects
   count = 0;
   for( i = 0; i < 10; i++ )
   {
      cout << "Create student object (y/n): ";
      cin >> response;
      if( response == 'y' || response == 'Y' )
      {
         cout << "Enter Roll no. of student: ";
         cin >> roll_no;
         cout << "Enter Name of student: ";
         cin >> name;
         s[i] = new student;  // dynamically creating objects
         s[i]->setdata( roll_no, name );
         count++;
      }
      else
         break;
   }
   cout << "Student details..." << endl;
   for( i = 0; i < count; i++ )
      s[i]->outdata();
   for( i = 0; i < count; i++ )    // release memory allocated to all objects
      delete s[i];
}
```

Run

```
Create student object (y/n): y
```

```
Enter Roll no. of student: 1
Enter Name of student: Rajkumar
Create student object (y/n): Y
Enter Roll no. of student: 2
Enter Name of student: Tejaswi
Create student object (y/n): y
Enter Roll no. of student: 3
Enter Name of student: Savithri
Create student object (y/n): n
Student details...
Roll No = 1
Name = Rajkumar
Roll No = 2
Name = Tejaswi
Roll No = 3
Name = Savithri
```

In main(), the statement

```
student * s[10];
```

creates an array of pointers of 10 possible student objects. It should be clearly understood that the space required for an array of 10 pointers to student objects is certainly less than the space for an array of 10 student objects. Hence, the student class objects are created by the program as and when they are needed (see Figure 12.6).

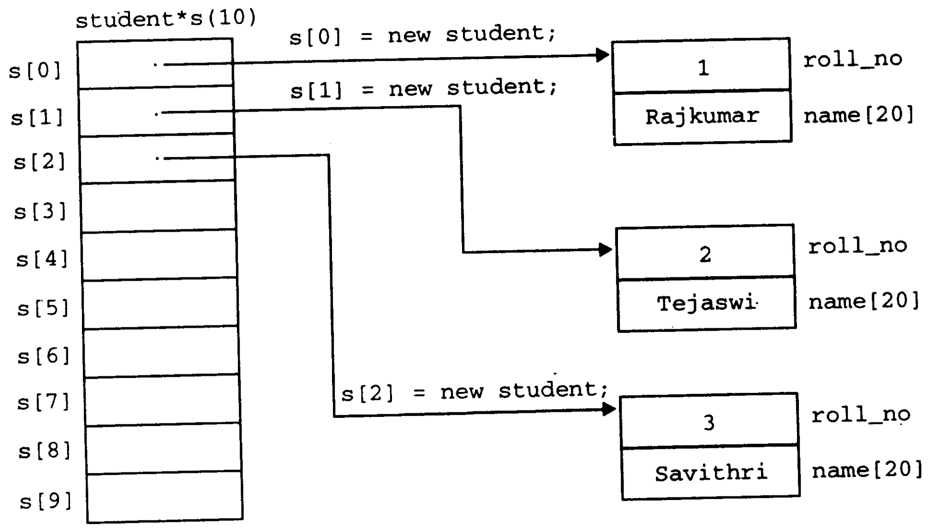


Figure 12.6: Array of pointers to objects and dynamic binding

Note that the subscripted notation used for object pointers is similar to the manner in which arrays of other data types are usually handled. Thus, s[count] is same as *(s + count) in the program. Similarly the statement

```
s[i]->outdata();
```

executes the `outdata()` member function in the `student` class for the i^{th} object of the `s` array. Pointers to objects could be effectively used to create and manipulate data structures like linked-lists, stacks, queues, etc.

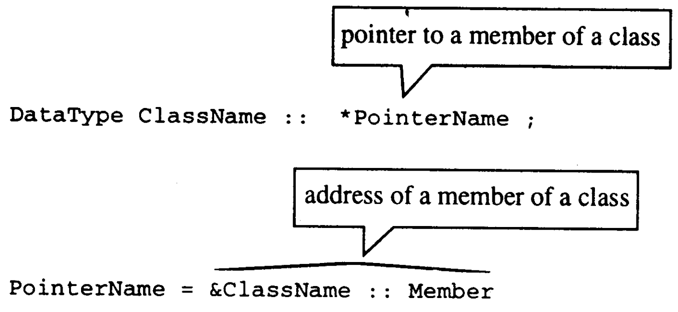
12.6 Pointers to Object Members

Whenever an object is created, memory is allocated to it. The data defining the object is held in the space allocated to it, i.e., the data and member functions of the object reside at specific memory locations subsequent to the creation of the object. Thus, a pointer to an object member can be obtained by applying the address-of operator (`&`) to a fully qualified class member-name (which may be a data item or a member function). A fully qualified member name is used to refer to a member of a class without any ambiguity. For instance, the declaration

```
<class_name>::<member_name>;
```

is a fully qualified declaration naming the member `<member_name>` of the class `<class_name>`. Preceding the above member reference with an `&` operator causes the address of the member `<member_name>` of the class `<class_name>` to be returned.

Members of a class can be accessed using either pointer to an object, or pointer to members itself. The address of a member can be obtained by using the address operator (`&`) to a *fully qualified* member name of a class similar to variables. A pointer to class members is declared using the operator `::*` with the class name. The syntax for defining the pointer to class members is shown in Figure 12.7.


pointer to a member of a class
address of a member of a class

```
DataType ClassName ::  *PointerName ;

PointerName = &ClassName :: Member
```

Figure 12.7: Syntax of defining pointer to class members

A variable of type *pointer to a member of class X* can be defined as follows:

```
DataType X::*ptr_name;
```

The `ptr_name` is a pointer to a data member of class `X`, which is of type `DataType`. A pointer to a member function can be defined as follows:

```
ReturnType ( X::* fn_ptr ) ( arguments );
```

It defines a pointer variable `fn_ptr` as a pointer to a member function of the class `X` which takes one or more arguments as specified by `arguments` and returns a value of type `ReturnType`. Consider the following specification of the class `X`:

```
class X
{
   private:
      int y;
```

```
        public:
            int a;
        public:
            int b;
            int init( int z );
    };
```

A pointer to the member a or b is defined as follows:

```
    int X::*ip;
```

The address of the member a can be assigned by

```
    ip = &X::a;
```

Similarly, the address of the member b can be assigned by

```
    ip = &X::b;
```

The address of the member a can also be assigned to a pointer during its definition as

```
    int X::*ip = &X::a;
```

The pointer variable ip, acts like the class member so that it can be invoked with a class object. In the above statement, the phrase X::* implies *pointer-to-member of the class* X. The phrase &X::a implies *address of the member* a *of the class* X.

The address of the private member y cannot be assigned by using the statement

```
    ip = &X::y;
```

Private members have the same access control privilege even with a pointer to the class members.

Normal pointer variable cannot be used as a pointer to the class member. Hence, the statement

```
    int *ptr = &X::a;
```

is invalid; The pointer and the variable have meaning only when they are associated with the class to which they belong. The scope resolution operator must be applied to both the pointer and the member.

Like pointers to data members, pointers to member functions can also be defined and invoked using the dereferencing operators. A pointer to the member function init() is defined as follows:

```
    int (X::*init_ptr)(int);
```

The address of the member init() can be assigned by

```
    init_ptr = &X::init;
```

to the pointer variable init_ptr. The different methods of accessing class members is shown in Figure 12.8.

Access through Objects

C++ provides operator, .* (dot-star) exclusively for use with pointers to members called *member dereferencing operator.* This operator is used to access class members using a pointer to members and it must be used with the objects of the class. The following statement,

```
    X obj1;
```

creates the object obj1 of the class X. Using the pointer variable ip, the following statement accesses the data member variable.

```
    obj1.*ip = 20;      // if ip is bound to a, it is same as the obj1.a;
    cout << obj1.*ip;
    int k = obj1.*ip;
```

Member functions can also be accessed using the operator `.*` as follows:

```
(obj1.*init_ptr)( 5 );       // same as the obj1.init() call
int k = (obj1.*init_ptr)( 5 );
```

The general format can be deduced to the following:

```
(object-name.*pointer-to-member-function)(arguments);
```

In such calls, the parentheses must be used explicitly, since the precedence of `()` is higher than the dereferencing `.*` operator.

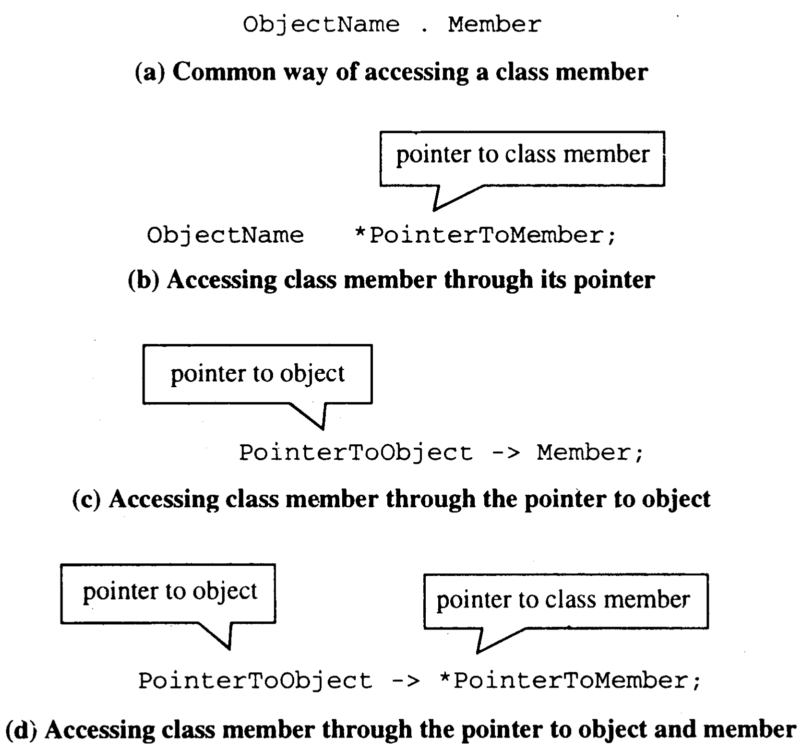

ObjectName . Member
(a) Common way of accessing a class member
pointer to class member
ObjectName *PointerToMember;
(b) Accessing class member through its pointer
pointer to object
PointerToObject -> Member;
(c) Accessing class member through the pointer to object
pointer to object
pointer to class member
PointerToObject -> *PointerToMember;
(d) Accessing class member through the pointer to object and member

Figure 12.8: Different ways of accessing class members

Access through Object Pointers

C++ provides another operator `->*` for use exclusively with pointers to members called member dereferencing operator. This operator is used to access a member using a pointer to it with *pointer to the object*. The following statement

```
X obj1;
X *pobj;
```

create the object `obj1` of the class `X` and the pointer `pobj` to the objects of the class `X`. Using the pointer variable `ip` (defined earlier), the following statements access the member variables.

```
pobj->*ip = 20;       // accesses a if ip is bound to data member a
cout << pobj->*ip;    // display data member a
int k = pobj->*ip;    // k = data member a's contents
```

Member functions can also be accessed using the operator `->*` as follows.

```
    (pobj.*init_ptr)( 5 );
    int k = (pobj->*init_ptr)( 5 );
```

The general format can be deduced to the following

```
    (pointer-to-object->*pointer-to-member-function)(arguments);
```

In such calls, the parentheses must be used explicitly, since the precedence of `()` is higher than the dereferencing `->*` operator. The program `ptrmemb.cpp` illustrates the concept of a pointer to class members.

```
// ptrmemb.cpp: pointer to class members
#include <iostream.h>
class X
{
  private:
     int y;   // through pointer it cannot be accessed
  public:   // all public members can be accessed through pointers
     int a;
     int b;
     int init( int z )
     {
        a = z;
        return z;
     }
};
void main()
{
   X obj;
   int X::*ip; // pointer to data member
   ip = &X::a; // address of data member a is assigned to pointer
   // access through object
   obj.*ip = 10;
   cout << "a in obj, after obj.*ip = 10 is " << obj.*ip << endl;
   X *pobj; // pointer to object of the class X
   pobj = &obj;
   // access through object pointer
   pobj->*ip = 10;
   cout << "a in obj, after pobj->*ip = 10 is " << pobj->*ip << endl;
   int (X::*ptr_init)(int);   // pointer to member function
   ptr_init = &X::init; // pointer to member function init()
   // access through object
   (obj.*ptr_init)(5);
   cout << "a in obj, after (obj.*ptr_init)(5) = " << obj.a << endl;
   // access through object pointer
   (pobj->*ptr_init)(5);
   cout << "a in obj, after (pobj->*ptr_init)(5) = " << obj.a << endl;
}
```

Run

```
a in obj, after obj.*ip = 10 is 10
```

```
a in obj, after pobj->*ip = 10 is 10
a in obj, after (obj.*ptr_init)(5) = 5
a in obj, after (pobj->*ptr_init)(5) = 5
```

Access Through Friend Functions

The friend functions can access private data members of a class although it is not in the scope of the class. Similarly, members of any access privilege can be accessed using pointers to members. Both the dereferencing operators `.*` and `->*` can be used to access class members. The program `friend.cpp` illustrates the concept of accessing class members through pointers from `friend` functions.

```
// friend.cpp: friend functions and pointer to members
#include <iostream.h>
class X
{
   private:
      int a;
      int b;
   public:
      X()
      {
         a = b = 0;
      }
      void SetMembers( int a1, int b1 )
      {
         a = a1;
         b = b1;
      }
      friend int sum( X x );
};
int sum( X objx )
{
   int X::*pa = &X::a;      // pointer to member a
   int X::*pb = &X::b;      // pointer to member b
   X *pobjx = &objx;        // pointer to object of the class X
   int result;
   // the member a is accessed through objects
   // and the member b is accessed through object pointer
   result = objx.*pa + pobjx->*pb;  // sum a and b;
   return result;
}
void main()
{
   X objx;
   void (X::*pfunc) (int, int);
   pfunc = &X::SetMembers;
   (objx.*pfunc)(5, 6);        // equivalent to objx.SetMembers(5, 6)
   cout << "Sum = " << sum( objx ) << endl;
   X *pobjx;                   // pointer to object of the class X
   pobjx = &objx;
   (pobjx->*pfunc)(7, 8);      // equivalent to pobjx->SetMembers(5, 6)
```

```
    cout << "Sum = " << sum( objx ) << endl;
}
```

Run

```
Sum = 11
Sum = 15
```

12.7 Function `set_new_handler()`

The C++ run-time system makes sure that when memory allocation fails, an error function is activated. By default, this function returns the value 0 to the caller of `new`, so that the pointer which is assigned by `new` is set to zero. The error function can be redefined, but it must comply with a few prerequisites, which are, unfortunately, compiler-dependent.

The function `set_new_handler()`, sets the function to be called when a request for memory allocation through the operator `new()` function cannot be satisfied. Its prototype is

```
        void ( * set_new_handler(void (* my_handler)() ))();
```

If `new()` cannot allocate the requested memory, it invokes the handler set by `set_new_handler()`. The user defined function, `my_handler()` should specify the actions to be taken when `new()` cannot satisfy a request for memory allocation.

If `my_handler()` returns, `new()` will again attempt to satisfy the request. Ideally, `my_handler` would release the memory and return. `new()` would then be able to satisfy the request and the program would continue. However, if `my_handler()` cannot provide memory for `new()`, `my_handler` must terminate the program. Otherwise, an infinite loop will be created.

The default handler is reset by `set_new_handler(0)`. Preferably, it is advisable to overload the `new()` to take appropriate actions as per the application requirement.

The function `set_new_handler` returns the old handler, if it has been defined. By default, no handler is installed. The user-defined argument function, `my_handler`, should not return a value.

The program `memhnd.cpp` demonstrates the implementation of user-defined function (in Borland C++) to handle memory resource shortage error.

```
// memhnd.cpp:user-defined handler to handle out-of-memory issue
#include <iostream.h>
#include <new.h>
#include <process.h>
void out_of_memory ()
{
    cout << "Memory exhausted, cannot allocate";
    exit( 1 );  // terminate the program
}
void main ()
{
    int *ip;
    long total_allocated = 0L;
    // install error function
    set_new_handler( out_of_memory );
    // eat up all memory
    cout << "Ok, allocating.." << endl;
    while (1)
    {
        ip = new int [100];
        total_allocated += 100L;
        cout << "Now got a total of " << total_allocated << " bytes" << endl;
    }
}
```

Run

```
Ok, allocating..
Now got a total of 100 bytes
Now got a total of 200 bytes
.....
.....
Now got a total of 29900 bytes
Memory exhausted, cannot allocate
```

The advantage of an allocation error function lies in the fact that once installed, `new` can be used without bothering whether the memory allocation has succeeded or not: upon failure, the error function is automatically invoked and the program terminates. It is a good practice to install a new handler in each C++ program, even when the actual code of the program does not allocate memory. Memory allocation can also fail in code which is not directly visible to the programmer, e.g., when streams are used or when strings are duplicated by low-level functions.

Most often, even standard C functions, which allocate memory such as `strdup()`, `malloc()`, `realloc()`, etc., trigger (invoke) the new handler when the memory allocation fails. That is, once a new handler is installed, such functions can be used in a C++ program without testing for errors. However, compilers exit where the C functions do not trigger the new handler.

12.8 `this` Pointer

It is observed that a member function of a given class is always invoked in the context of some object of the class; there is always an *implicit substrate* (implicitly defined) for the function to act on. C++ has a keyword `this` to address this substrate (it is not available in the static member functions) . The keyword `this` is a pointer variable, which always contains the address of the object in question. The `this` pointer is implicitly defined in each member function (whether public or private); therefore, it appears as if each member function of the class `Test` contains the following declaration:

```
        extern Test *this;
```

Every member function of a class is born with a pointer called `this`, which points to the object with which the member function is associated.

Thus, member function of every object has access to a pointer named `this`, which points to the object itself. When a member function is invoked, it comes into existence with the value of `this` set to the address of the object for which it is called. The `this` pointer can be treated like any other pointer to an object. Using a `this` pointer, any member function can find out the address of the object of which it is a member. Method of accessing a member of a class from within a class using `this` pointer is shown in Figure 12.9.

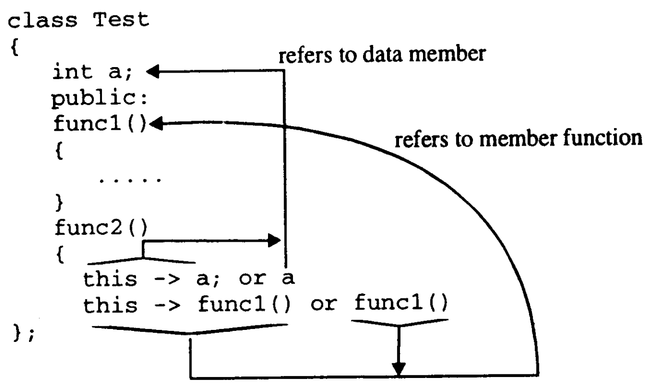


Figure 12.9: Accessing class members using this pointer

The this pointer can also be used to access the data in the object it points to. The program this.cpp illustrates the working of this pointer.

```
// this.cpp:accessing data members through this pointer
#include <iostream.h>
class Test
{
  private:
     int a;
  public:
     void setdata( int init_a )
     {
       a = init_a;        // normal way to set data
       cout<<"Address of my object, this in setdata(): "<< this <<endl;
       this->a = init_a; // another way to set data
     }
     void showdata()
     {
       // normal way to show data
       cout << "Data accessed in normal way: " << a << endl;
       cout<<"Address of my object, this in showdata(): "<< this<<endl;
       // data access through this pointer
       cout << "Data accessed through this->a: " << this->a;
     }
};
void main()
{
  Test my;
  my.setdata( 25 );
  my.showdata();
}
```

Run

```
Address of my object, this in setdata(): 0xfff2
Data accessed in normal way: 25
```

```
Address of my object, this in showdata(): 0xfff2
Data accessed through this->a: 25
```

A more practical use of `this` pointer is in returning values from member functions. When an object is local to the function, the object will be destroyed when the function terminates. It necessitates the need for a more permanent object while returning it by reference. Consider the member function `add()` of the class `complex`:

```
complex complex::add( complex c2 )
{
   real = real + c2.real;          // add real parts
   imag = imag + c2.imag;          // add imaginary parts
   return complex( real, imag );     // create an object and return
}
```

It adds the object `c2` to a default object and returns the updated default object by explicitly creating a nameless object using the statement

```
return complex( real, imag );
```

It can be replaced by the statement

```
return *this;
```

without the loss of functionality. The modified definition of `add()` appears as follows:

```
complex complex::add( complex c2 )
{
   real = real + c2.real;          // add real parts
   imag = imag + c2.imag;          // add imaginary parts
   return *this;
}
```

Since `this` is a pointer to the object of which the function is a member, `*this` naturally refers to the object pointed to by `this` pointer. The statement

```
return *this;
```

returns this object by value.

For a given class `X`, in each one of its member functions, the pointer `this` is implicitly declared as

```
X *const this;
```

and initialized to point to the object for which the member function is invoked. As the pointer `this` is declared as `* const`, it cannot be changed for a particular object ensuring that the access to the object is not lost, even accidentally. However, the value of `this` is different for every individual object declared or created in the program. The compiler treats `this` as a keyword (reserved word) as a result of which it cannot be explicitly declared. Further, it (the compiler) also places a restriction which prevents the keyword `this` from being used outside a class member function body.

12.9 Self-Referential Classes

Many of the frequently used dynamic data structures like stacks, queues, linked-lists, etc., use self-referential members. Classes can contain one or more members which are pointers to other objects of the same class. This pointer holds an address of the next object in a data structure. Such a feature is essential for implementing dynamic data structures such as linked lists, stack, trees, etc.

Linked List

A list having node, which is a pointer to the next node in a list is called linked list. The pictorial representation of a linked list having pointer to the next object of the same class is shown in Figure 12.10. The program listed in `list.cpp` implements a linked list of integers using such a self-referential class. The program uses a pointer called `this` pointer.

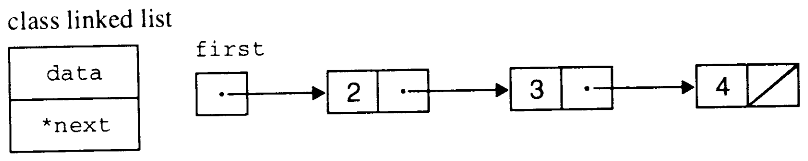


Figure 12.10: Linked list with self-referential classes

```
// list.cpp: Linked list having self reference
#include <iostream.h>
#include <process.h>
// linked list class
class list
{
  private:
    int data;     // data of a node
    list *next;   // pointer to next node
  public:
     list()
     {
        data = 0;
        next = NULL;
     }
     list(int dat)
     {
        data = dat;
        next = NULL;
     }
    ~list() {}
     int get() { return data; }
     void insert(list *node);      // Inserts new node at list
     friend void display(list *);  // Display list
};
// Inserts node. If list empty the first node is created else the
// new node is inserted at the end of a list
void list::insert( list *node )
{
   list *last = this;   // this node pointer to catch last node
   while( last->next )  // if node-next != NULL, it is not last node
      last = last->next;
   last->next = node;   // make last node point to new node
}
// Displays the doubly linked list in both forward and reverse order by
// making use of the series of next and prev pointers.
```

```
void display( list *first )
{
   list *traverse;
   cout << "List traversal yields: ";
   // scan for all the elements
   for( traverse = first; traverse; traverse = traverse->next )
     cout << traverse->data << ", ";
   cout << endl;
}
void main(void)
{
   int choice, data;
   list *first = NULL;  // initially points to NULL
   list *node;    // pointer to new node to be created
   while( 1 )
   {
      cout << "Linked List..." << endl;
      cout << "1.Insert" << endl;
      cout << "2.Display" << endl;
      cout << "3.Quit" << endl;
      cout << "Enter Choice: ";
      cin >> choice;
      switch (choice)
      {
         case 1:
            cout << "Enter Data: ";
            cin >> data;
            node = new list( data );
            if( first == NULL )
               first = node;
            else
               first->insert( node );
            break;
         case 2:
            display( first );
            break; // Display list.
         case 3:
            exit( 1 );
         default:
            cout << "Bad option selected" << endl;
         continue;
      }
   }
}
```

Run

```
Linked List...
1.Insert
2.Display
3.Quit
```

```
Enter Choice: 1
Enter Data: 2
Linked List...
1.Insert
2.Display
3.Quit
Enter Choice: 2
List traversal yields: 2,
Linked List...
1.Insert
2.Display
3.Quit
Enter Choice: 1
Enter Data: 3
Linked List...
1.Insert
2.Display
3.Quit
Enter Choice: 1
Enter Data: 4
Linked List...
1.Insert
2.Display
3.Quit
Enter Choice: 2
List traversal yields: 2, 3, 4,
Linked List...
1.Insert
2.Display
3.Quit
Enter Choice: 3
```

The use of a self-referential class is inevitable in the above program, since each node in the stack has a pointer to another node of its own type, which is its predecessor (in the case of the stack).

Several problems whose solutions are based on the use of data structures like trees, graphs and lists make extensive use of self-referential class.

Doubly Linked List

Using `this` pointer when referring to a member of its own class is often unnecessary, as illustrated earlier; the major use of the `this` pointer is for writing member functions that manipulate pointers directiy. The doubly linked list has two pointer nodes: one pointing to the next node in the list and another pointing to the previous node in the list. The pictorial representation of a doubly linked list is shown in Figure 12.11.

The program `dll.cpp` makes use of the data structure, doubly linked list, illustrating the typical use of the `this` pointer at relevant points. The `this` pointer is particularly used as a pointer to the first node while traversing through the entire list.

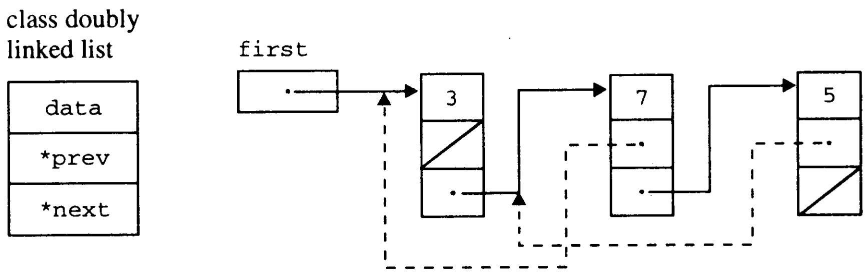


Figure 12.11: Doubly linked list representation

```
// dll.cpp: doubly linked list
#include <iostream.h>
#include <process.h>
class dll            // doubly linked list class
{
  private:
    int data;    // data of a node
    dll *prev;   // pointer to previous node
    dll *next;   // pointer to next node
  public:
    dll()
    {
      data = 0;
      prev = next = NULL;
    }
    dll( int data_in )
    {
      data = data_in;
      prev = next = NULL;
    }
    ~dll()
    {
      cout << "->" << data;
    }
    int get() { return data; }
    void insert(dll *node);          // Inserts new node at list
    friend void display(dll *);      // Display list
    void FreeAllNodes();
};
// Inserts node. If list empty the first node is created else the
// new node is inserted immediately after the first node.
void dll::insert( dll *node )
{
  dll *last;
  // find out last node. this points to first node
  for( last = this; last->next; last = last->next );
  // insert new node at the end of list
```

```
   node->prev = last;
   node->next = last->next;
   last->next = node;
}
void dll::FreeAllNodes()
{
   cout << "Freeing the node with data: ";
   // this points to first node, use it to release all the nodes
   for( dll *first = this; first; first = first->next )
      delete first;
}
// Displays the doubly linked list in both forward and reverse order making
// use of the series of next and prev pointers.
void display( dll *first )
{
   dll *traverse = first;
   if( traverse == NULL)
   {
      cout << "Nothing to display !" << endl; // along the list.
      return;
   }
   else
   {
      cout << "Processing with forward -> pointer: ";
      // scan for all the elements in forward direction
      for( ;traverse->next; traverse = traverse->next )
        cout << "->" << traverse->data;
      // display last element
      cout << "->" << traverse->data << endl;
      cout << "Processing with backward <- pointer: ";
      // scan for all the elements in reverse direction
      for( ;traverse->prev; traverse = traverse->prev )
           cout << "->" << traverse->data;
      // display first element
      cout << "->" << traverse->data << endl;
   }
}
dll * InsertNode( dll *first, int data )
{
   dll *node;
   node = new dll( data );
   if( first == NULL )
      first = node;
   else
      first->insert( node );
   return first;
}
void main( void )
{
   int choice, data;
   dll *first = NULL;   // initially points to NULL
   cout << "Double Linked List Manipulation..." << endl;
   while( 1 )
   {
      cout << "Enter Choice ([1] Insert, [2] Display, [3] Quit): ";
      cin >> choice;
      switch (choice)
      {
         case 1:
            cout << "Enter Data: ";
            cin >> data;
            first = InsertNode( first, data );
            break;
         case 2:
            display( first );
            break; // Display list.
         case 3:
            first->FreeAllNodes();  // release all nodes
            exit( 1 );
         default:
            cout << "Bad option selected" << endl;
         continue;
      }
   }
}
```

Run

```
Double Linked List Manipulation...
Enter Choice ([1] Insert, [2] Display, [3] Quit): 1
Enter Data: 3
Enter Choice ([1] Insert, [2] Display, [3] Quit): 2
Processing with forward -> pointer: ->3
Processing with backward <- pointer: ->3
Enter Choice ([1] Insert, [2] Display, [3] Quit): 1
Enter Data: 7
Enter Choice ([1] Insert, [2] Display, [3] Quit): 2
Processing with forward -> pointer: ->3->7
Processing with backward <- pointer: ->7->3
Enter Choice ([1] Insert, [2] Display, [3] Quit): 1
Enter Data: 5
Enter Choice ([1] Insert, [2] Display, [3] Quit): 2
Processing with forward -> pointer: ->3->7->5
Processing with backward <- pointer: ->5->7->3
Enter Choice ([1] Insert, [2] Display, [3] Quit): 0
Bad option selected
Enter Choice ([1] Insert, [2] Display, [3] Quit): 3
Freeing the node with data: ->3->7->5
```

Besides handling dynamic data structures, the `this` pointer finds extensive application in the following contexts:

- Member functions returning pointers to their respective objects.

- Overloaded operators which return object values by reference.
- Virtual functions wherein decisions, as to which version of an overloaded function is to be executed, is taken only during runtime (late binding).

12.10 Guidelines for Passing Object Parameters

The parameters to normal functions or member functions, of a class can be passed either by value, pointer, or reference. However, passing some objects by pointers or reference is much efficient when compared to passing by value even though modification in a callee need not be reflected in the caller. A few guidelines that help in taking decision on choosing appropriate parameter passing scheme are the following:

[1] If a function does not modify an argument, which is a built-in type or a "small" user-defined type (class objects), pass arguments by value. The meaning of "small" refers to data-type, which require few bytes to represent its objects and it is system dependent.

[2] If a function modifies an argument, which is a built-in type, pass arguments by a pointer. It makes processing of data explicit to anyone reading the code, which modifies built-in type variables.

[3] If a function modifies or does not modify a "large" user-defined type, pass arguments by reference. Any function, which modifies private data (and hence protected) of an object must either be a member function, or a friend function. This is justifiable, since the "class" has control over the functions which modify class's private data. In this case, just because the address of an object is handed over to a function does not mean the function can secretly modify the private data of an object. As far as object data members are concerned, it is very clear and straight forward to answer "who has permission to modify this object ?" Hence, it is advisable to pass reference to an object instead of value or a pointer.

Review Questions

12.1 What is the difference between dynamic memory allocation and dynamic objects ?

12.2 Justify the need of object cleanup and initialization facility for creating live objects

12.3 Explain why C++ is treated as the middle ground between static and dynamic binding languages.

12.4 What is the difference between stack based and heap-based objects ?

12.5 What is dereferencing of objects ? Write a program for illustrating the use of object references.

12.6 What are self-referential classes ? Write a program to create an ordered linked list.

12.7 What are live objects ? Write a program to illustrate live objects supporting different ways of creating them. Will an object created using `new` operator occupy more space than necessary ?

12.8 Write a program to access members of a student class using pointer to object members.

12.9 Justify the need for "allowing pointers to class members accessing private members of a class"

12.10 Explain how memory allocation failure can be handled in C++ ?.

12.11 What is `this` pointer ? What is your reaction to the statement:

```
delete this;
```

Write a program demonstrating the use of `this` pointer.

12.12 Write an interactive program for creating a doubly linked list. The program must support ordered insertion and deletion of a node.

13
Operator Overloading

13.1 Introduction

The operators such as `+`, `-`, `+=`, >, >>, etc., are designed to operate only on standard data types in structured programming languages such as C. The + operator can be used to perform the addition operation on integer, floating-point, or mixed data types as indicated in the expression `(a+b)`. In this expression, the data type of the operands `a` and `b` on which the + operator is operating, is not mentioned explicitly. In such cases, the compiler implicitly selects suitable addition operation (integer, floating-point, double, etc.,) depending on the data type of operands without any assistance from the programmer. Consider the following statements:

```
int a, b, c;
float x, y, z;
c = a + b;    // 1: integer addition and assignment
z = x + y;    // 2: floating-point addition and assignment
x = a + b;    // 3: integer addition and floating point assignment
```

The operators = and + behave quite differently in the above statements: the first statement does integer addition and assigns the result to `c`, the second performs floating-point addition and assigns the result to `z`, and the last performs integer addition and assigns the result to the floating-point variable `x`. It indicates that, the + operator is overloaded implicitly to operate on operands of any standard data type supported by the language. Unlike C, in C++, such operators can also be overloaded explicitly to operate on operands of user-defined data types. For instance, the statement

```
c3 = AddComplex( c1, c2 );
```

performs the addition of operands `c1` and `c2` belonging to the user defined data type and assigns the result to `c3` (which is also operand of the user defined data type). In C++, by overloading the + operator, the above statement can be changed to an easily readable form:

```
c3 = c1 + c2;
```

It tries to make the user-defined data types behave in a manner similar (and have the same *look and feel*) to the built-in data types, thereby allowing the user to redefine the language itself. Operator overloading, thus allows to provide additional meaning to operators such as +,*, >=,+=,etc., when they are applied to user defined data types. It allows the user to program (develop solution to) the problems as perceived in the real world.

The operator overloading feature of C++ is one of the methods of realizing *polymorphism*. The word polymorphism is derived from the Greek words *poly* and *morphism* (*polymorphism = poly + morphism*). Here, *poly* refers to many or multiple and *morphism* refers to actions, i.e., performing many actions with a single operator. As stated earlier, the + operator performs integer addition if the operands are of integer type and floating point addition if the operands are of real type.

The concept of operator overloading can also be applied to data conversion. C++ offers automatic conversion of primitive data types. For example, in the statement `x=a+b`, the compiler implicitly converts the integer result to floating-point representation and then assigns to the `float` variable `x`. But the conversion of user defined data types requires some effort on the part of the programmer. Thus, operator overloading concepts are applied to the following two principle areas:

- Extending capability of operators to operate on user defined data.
- Data conversion.

Operator overloading extends the *semantics* of an operator without changing its *syntax*. The grammatical rules defined by C++ that govern its use such as the number of operands, precedence, and associativity of the operator remain the same for overloaded operators. Therefore, it should be remembered that the overloaded operator should not change its original meaning. However, semantics (meaning) can be changed, but it is advisable to retain the predefined logical meaning.

13.2 Overloadable Operators

C++ provides a wide variety of operators to perform operations on various operands. The operators are classified into *unary* and *binary* operators based on the number of arguments on which they operate. C++ allows almost all operators to be overloaded in which case atleast one operand must be an instance of a class (object). It allows overloading of the operators listed in Table 13.1.

The precedence relation of overloadable operators and their expression syntax remains the same even after overloading. Even if there is a provision to change the operator precedence or the expression syntax, it does not offer any advantage. For instance, it is improper to define a unary division (/) or a binary complement (~), since the change of precedence or syntax leads to ambiguity. For example, defining an operator `**` to represent exponentiation as in the case of Fortran language, appears to be obvious, however, interpretation of the expression `a**b`, leads to confusion; whether to interpret it as `a*(*b)` or `(a)**(b)`, because, C++ already interprets it as `a*(*b)`.

Operator Category	Operators
Arithmetic	`+, -, *, /, %`
Bit-wise	`&, |, ~, ^`
Logical	`&&, ||, !`
Relational	`>, <, ==, !=, <=, >=`
Assignment or Initialization	`=`
Arithmetic Assignment	`+=, -=, *=, /=, %=, &=, |=, ^=`
Shift	`<<, >>, <<=, >>=`
Unary	`++, --`
Subscripting	`[]`
Function Call	`()`
Dereferencing	`->`
Unary Sign Prefix	`+, -`
Allocate and Free	`new, delete`

Table 13.1: C++ overloadable operators

13.3 Unary Operator Overloading

Consider an example of class `Index` which keeps track of the index value. The program `index1.cpp` having class members to maintain the index value is listed below:

```
// index1.cpp: Index class with functions to keep track of index value
#include <iostream.h>
class Index
{
   private:
      int value;                // Index Value
   public:
      Index()                   // No argument constructor
      {
         value = 0;
      }
      int GetIndex()             // Index Access
      {
         return value;
      }
      void NextIndex()          // Advance Index
      {
         value = value + 1;
      }
};
void main()
{
   Index idx1, idx2;        // idx1 and idx2 are objects of Index class
   // Display index values
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
   // Advance Index objects
   idx1.NextIndex();
   idx2.NextIndex();
   idx2.NextIndex();
   // Display index values
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
}
```

Run

```
Index1 = 0
Index2 = 0
Index1 = 1
Index2 = 2
```

The function `NextIndex()` advances (increments) the index value. Instead of using such functions, the operators like ++ (increment operator) can be used to perform the same job. It enhances the program readability without the loss of functionality. A new version of the class program `index1.cpp`, is rewritten using overloaded increment operator. The program `index2.cpp` illustrates overloading of ++ operator.

```
// index2.cpp: Index class with operator overloading
#include <iostream.h>
class Index
{
  private:
     int value;              // Index Value
  public:
     Index()                 // No argument constructor
     {
        value = 0;
     }
     int GetIndex()          // Index Access
     {
        return value;
     }
     void operator ++()      // prefix or postfix increment operator
     {
        value = value + 1;   // value++;
     }
};
void main()
{
   Index idx1, idx2;       // idx1 and idx2 are objects of Index class
   // Display index values
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
   // Advance Index objects with ++ operators
   ++idx1;            // equivalent to idx1.operator++();
   idx2++;
   idx2++;
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
}
```

Run

```
Index1 = 0
Index2 = 0
Index1 = 1
Index2 = 2
```

In `main()`, the statements

```
++idx1;            // equivalent to idx1.operator++();
idx2++;
```

invoke the overloaded ++ operator member function defined in the class `Index`:

```
void operator ++()        // prefix or postfix increment operator
```

The name of this overloaded function is ++. The word `operator` is a keyword and is preceded by the return type `void`. The operator to be overloaded is written immediately after the keyword `operator`. This declarator informs the compiler to invoke the overloaded operator **function** ++ whenever the unary increment operator is prefixed or postfixed to an object of the `Index` **class**.

The variables `idx1` and `idx2` are the objects of the class `Index`. The index value is advanced by using statements such as `++idx1; idx2++;` instead of explicitly invoking the member function `NextIndex()` as in the earlier program. The operator is applied to objects of the `Index` class. Yet, operator function ++ takes no arguments. It increments the data member `value` of the Index class's objects. Figure 13.1 shows the `Index` class representation and invocation of its member functions when they are accessed implicitly (constructor function) or explicitly (other members).

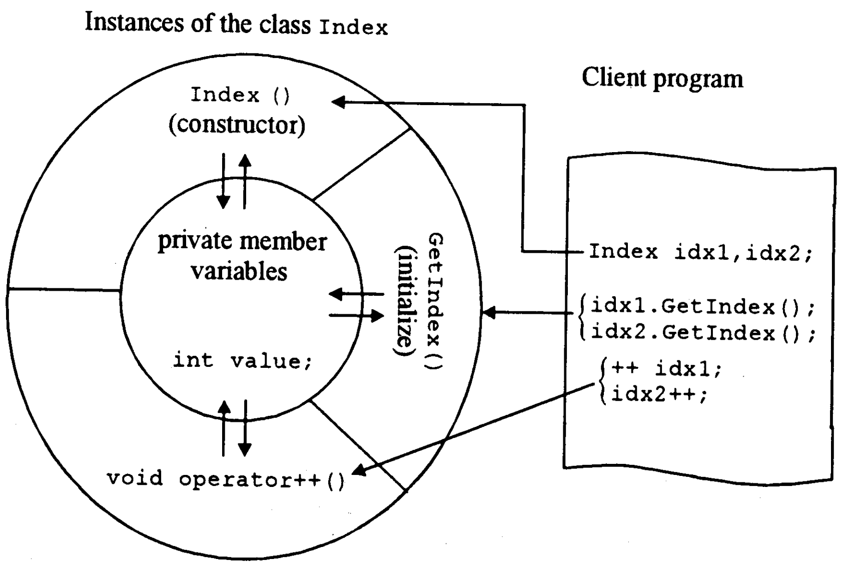
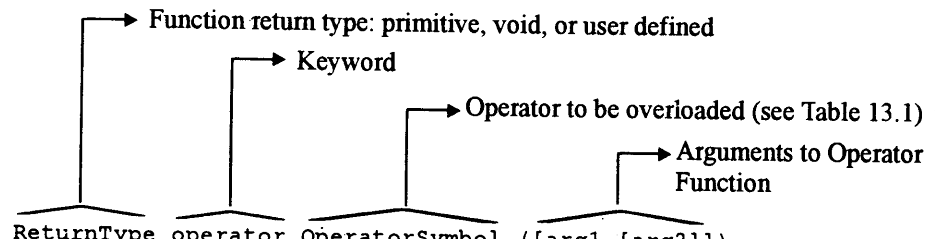


Figure 13.1: Index class and ++ operator overloading

13.4 `operator` Keyword

The keyword `operator` facilitates overloading of the C++ operators. The general format of operator overloading is shown in Figure 13.2. The keyword `operator` indicates that the *operator symbol* following it, is the C++ operator to be overloaded to operate on members of its class. The operator overloaded in a class is known as *overloaded operator function*.

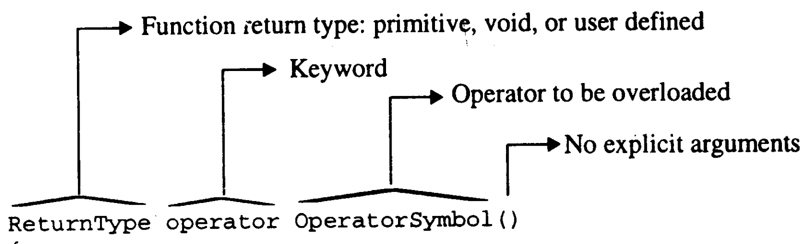

Function return type: primitive, void, or user defined
Keyword
Operator to be overloaded (see Table 13.1)
Arguments to Operator Function

```
ReturnType operator OperatorSymbol ([arg1,[arg2]])
{
   // body of Operator function

}
```

Figure 13.2: Syntax of operator overloading

Overloading without explicit arguments to an operator function is known as *unary operator overloading* and overloading with a single explicit argument is known as *binary operator overloading*. However, with friend functions, unary operators take one explicit argument and binary operators take two explicit arguments. The syntax of overloading the unary operator is shown in Figure 13.3.


Function return type: primitive, void, or user defined
Keyword
Operator to be overloaded
No explicit arguments

```
ReturnType operator OperatorSymbol()
{
   // body of Operator function

}
```

Figure 13.3: Syntax for overloading unary operator

The following examples illustrate the overloading of unary operators:

```
(1)     Index operator +();
(2)     int operator -();
(3)     void operator ++();
(4)     void operator --();
(5)     int operator *();
```

Similar to other member functions of a class, an overloaded operator member function can be either defined within the body of a class or outside the body of a class. The following class specification defines an overloaded operator member function within the body of a class:

```
class MyClass
{
   // class data or function stuff
   int operator++()       // member function definition
   {
      // body of a function
   }
};
```

A skeleton of the same class having the operator member function definition outside its body is as follows:

```
class MyClass
{
    // class data or function stuff
     int operator ++(); // prototype declaration
};
// overloaded member function definition
int MyClass::operator++()
{
     // body of a function
}
```

The process of operator overloading generally involves the following steps:

1. Declare a class (that defines the data type) whose objects are to be manipulated using operators.
2. Declare the operator function, in the *public* part of the class. It can be either a normal member function or a friend function.
3. Define the operator function either within the body of a class or outside the body of the class (however, the function prototype must exist inside the class body).

The syntax for invoking the overloaded unary operator function is as follows:

object operand
operator object

The first syntax can be used to invoke a prefix operator function, for instance, `++idx1`, and the second syntax can be used to invoke a postfix operator function, for instance, `idx1++`.

The syntax for invoking the overloaded binary operator function is as follows:

object1 operator object2

For instance, the expression `idx1+idx2` invokes the overloaded member function + of the `idx1` object's class by passing `idx2` as the argument. Note that, in an expression invoking the binary operator function, one of the operands must be the object. The above syntax is interpreted as follows:

object1.operator OperatorSymbol(object2)

Operator Arguments

In `main()` of `index2.cpp` program, `operator++()` is applied to the object of the class `Index` as in the expression `idx2++`; it can be observed that the `operator++()` takes no arguments explicitly. The execution of the expression `idx2++` invokes a member function `operator++()` defined in the class `Index`. In this function, the data members of the object `idx2` are manipulated.

13.5 Operator Return Values

The operator function in the program `index2.cpp` has a subtle defect. An attempt to use an expression such as

```
idx1 = idx2++;
```

will lead to a compilation error like *Improper Assignment,* because the return type of `operator++` is defined as `void` type. The above assignment statement tries to assign the void return type to the object (`idx1`) of the `Index` class. Such an assignment operation can be permitted after modifying the return type of the `operator++()` member function of the `Index` class in the `index2.cpp` program. A program with required modifications is listed in `index3.cpp`.

```
// index3.cpp: Index class with overloaded operator returning an object
#include <iostream.h>
class Index
{
  private:
     int value;              // Index Value
  public:
     Index()                 // No argument constructor
     {
        value = 0;
     }
```

```
      int GetIndex()              // Index Access
      {
         return value;
      }
      Index operator ++()         // Returns Index object
      {
         Index temp;              // temp object
         value = value + 1;       // update index value
         temp.value = value;      // initialize temp object
         return temp;             // return temp object
      }
};
void main()
{
   Index idx1, idx2;  // idx1 and idx2 are objects of class Index
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
   idx1 = idx2++;    //returned object of idx2++ is assigned to idx1
   idx2++;                  // returned object of idx2++ is unused
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
}
```

Run

```
Index1 = 0
Index2 = 0
Index1 = 1
Index2 = 2
```

In `main()`, the statement

```
   idx1 = idx2++;    //returned object of idx2++ is assigned to idx1
```

invokes the overloaded operator function and assigns the return value to the object `idx1` of the class `Index`. The operator `++()` function creates a new object of the class `Index` called `temp` to be used as a return value; it can be assigned to another object. The `value` data member of the implicit object `idx2` is incremented and then assigned to the `temp` object which is returned to the caller. The returned object is assigned to the destination object `idx1`.

13.6 Nameless Temporary Objects

In the program `index3.cpp`, an intermediate (a temporary) object `temp` is created as a return object. A convenient way to return an object is to create a nameless object in the return statement itself. The program `index4.cpp`, illustrates the overloaded operator function returning a nameless object.

```
// index4.cpp: Index class with overloaded operator returning nameless object
#include <iostream.h>
class Index
{
   private:
      int value;                  // Index Value
```

```
    public:
       Index()                   // No argument constructor
       {  value = 0;  }
       Index( int val )          // Constructor with one argument
       {
          value = val;
       }
       int GetIndex()             // Index Access
       {
          return value;
       }
       Index operator ++() // Returns nameless object of class Index
       {
          value = value + 1;
          return Index( value );    // calls one-argument constructor
       }
};
void mair()
{
   Index idx1, idx2;    // idx1 and idx2 are the objects of Index
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
   idx1 = idx2++; // return object idx2++ is assigned to object idx1
   idx2++;            // return object idx2++ is unused
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
}
```

Run

```
Index1 = 0
Index2 = 0
Index1 = 1
Index2 = 2
```

In the program `index3.cpp`, the statements used to return an object are the following:

```
Index temp;
value = value + 1;
temp.value = value;
return temp;
```

In this program, the statements,

```
value = value + 1;
return Index( value );
```

perform the same operation as achieved by the above four statements. It creates a nameless object by passing an initialization value. To perform this operation, the following parameterized constructor is added as the constructor member function to the `Index` class:

```
Index( int val )
{
   value = val;
}
```

13.7 Limitations of Increment/Decrement Operators

The prefix notation causes a variable (of type standard data type) to be updated before its value is used in the expression, whereas the postfix notation causes it to be updated after its value is used. However, the statement (built using user-defined data types and overloaded operator),

```
idx1 = ++idx2;
```

has exactly the same effect as the statement

```
idx1 = idx2++;
```

When ++ and -- operators are overloaded, there is no distinction between the prefix and postfix overloaded operator function. This problem is circumvented in advanced implementations of C++, which provides additional syntax to express and distinguish between prefix and postfix overloaded operator functions. A new syntax to indicate postfix operator overloaded function is:

```
operator ++( int )
```

The program `index5.cpp` illustrates the invocation of prefix and postfix operator functions. Note that the old syntax is used to overload prefix operator function.

```
// index5.cpp: Index class with overloaded prefix and postfix unary operators
#include <iostream.h>
class Index
{
   private:
      int value;              // Index Value
   public:
      Index()                 // No argument constructor
      {  value = 0;   }
      Index( int val )        // Constructor with one argument
      {
         value = val;
      }
      int GetIndex()           // Index Access
      {
         return value;
      }
      // Operator overloading for prefix operator
      Index operator ++()
      {
         // Object is created with the ++value, hence object is
         // created with a new value of 'value' and returned
         return Index( ++value );
      }
      // Operator overloading for postfix operator
      Index operator ++(int)
      {
         // Object is created with the value++, hence object is
         // created with old value of 'value' and returned
         return Index( value++ );
      }
};
```

```
void main()
{
   Index idx1(2), idx2(2), idx3, idx4;
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
   idx3 = idx1++;   // postfix increment
   idx4 = ++idx2;   // prefix increment
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex3 = " << idx3.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
   cout << "\nIndex4 = " << idx4.GetIndex();
}
```

Run

```
Index1 = 2
Index2 = 2
Index1 = 3
Index3 = 2
Index2 = 3
Index4 = 3
```

In the postfix `operator ++(int)` function, first a nameless object with the old index value is created and then, the index value is updated to achieve the intended operation. The compiler will just make a call to this function for postfix operation, but the responsibility of achieving this rests on the programmer.

The above discussion on *unary plus* overloading is also applicable to overloading of unary decrement and negation operators. It is illustrated by the program `index6.cpp`.

```
// index6.cpp: Index class with unary operator overloading -, ++, and --
#include <iostream.h>
class Index
{
  private:
     int value;               // Index Value
  public:
     Index()                  // No argument constructor
     {  value = 0;   }
     Index( int val )         // Constructor with one argument
     {
        value = val;
     }
     int GetIndex()            // Index Access
     {
        return value;
     }
     Index operator -()       // Negation of Index Value
     {
        return Index( -value );
     }
```

```
      Index operator ++()   // Prefix increment
      {
         ++value;
         return Index( value );
      }
      Index operator --()       // Prefix decrement
      {
         --value;
         return Index( value );
      }
};
void main()
{
   Index idx1, idx2;
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
   idx2++;
   idx1 = -idx2;     // negate idx2 and assign to idx1
   ++idx2;
   --idx2;          // prefix decrement
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
}
```

Run

```
Index1 = 0
Index2 = 0
Index1 = -1
Index2 = 1
```

Overloading of unary operator does not necessarily mean that it is overloaded to operate on a class's object, which has a single data member. Within the body of a overloaded unary operator function, any amount of data can be manipulated. One of the best example is manipulation of `date` object data members. A class called `date` can have three data members `day`, `month`, and `year`. To increment date by one, it may necessitate updation of all the fields on the date class. It depends on the current values of `date` class's object data members as illustrated in the program `mydate.cpp`. It has overloaded unary increment operator function to update date object's data members.

```
// mydate.cpp: overloading ++ operator to increment date
#include <iostream.h>
class date
{
   int day;
   int month;
   int year;
   public:
      date()
      {
         day = 0; month = 0; year = 0;
      }
      date( int d, int m, int y )
      {
         day = d; month = m; year = y;
      }
      void read()
      {
         cout << "Enter date <dd mm yyyy>: ";
         cin >> day >> month >> year;
      }
      void show()
      {
         cout << day << ":" << month << ":" << year;
      }
      int IsLeapYear()
      {
         if( (year % 4 == 0 && year % 100 != 0 ) || (year % 400 == 0) )
            return 1;
         else
            return 0;
      }
      int thisMonthMaxDay()
      {
         int m[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
         if( month == 2 && IsLeapYear() )
            return 29;  // February month with leap year will have 28 days
         else
            return m[month-1];
      }
      // unary increment operator overloading
      void operator ++()
      {
         ++day;
         // adjust all fields of date according to current day
         // so that they hold valid date
         if( day > thisMonthMaxDay() )
         {
            // set day to 1 and increment month
            day = 1;
            month++;
         }
         if( month > 12 )
         {
            // month to January (1) and increment year
            month = 1;
            year++;
         }
      }
};
void nextday( date & d )
{
   cout << "Date "; d.show();
   ++d; // invokes operator function
   cout << " on increment becomes "; d.show();
   cout << endl;
}
void main()
{
   date d1( 14, 4, 1971 );
   date d2( 28, 2, 1992 ); // leap year
   date d3( 28, 2, 1993 );
   date d4( 31, 12, 1995 );
   nextday( d1 );
   nextday( d2 );
   nextday( d3 );
   nextday( d4 );
   date today;
   today.read();
   nextday( today );
}
```

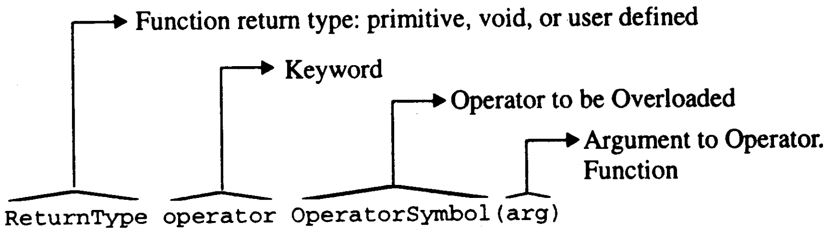
Run

```
Date 14:4:1971 on increment becomes 15:4:1971
Date 28:2:1992 on increment becomes 29:2:1992
Date 28:2:1993 on increment becomes 1:3:1993
Date 31:12:1995 on increment becomes 1:1:1996
Enter date <dd mm yyyy>: 11 9 1996
Date 11:9:1996 on increment becomes 12:9:1996
```

The updation of date requires to take care of conditions such as whether the year is a leap year or not. If it is leap year and month is February, it will have 29 days instead of usual 28 days. Such cases need to be handled explicitly (see the second and third output line in ***Run***).

13.8 Binary Operator Overloading

The concept of overloading unary operators applies also to the binary operators. The syntax for overloading a binary operator is shown in Figure 13.4.


Function return type: primitive, void, or user defined
Keyword
Operator to be Overloaded
Argument to Operator.
Function

```
ReturnType operator OperatorSymbol(arg)
{
   // body of Operator function

}
```

Figure 13.4: Syntax for overloading a binary operator

The binary overloaded operator function takes the first object as an implicit operand and the second operand must be passed explicitly. The data members of the first object are accessed without using the dot operator whereas, the second argument members can be accessed using the dot operator if the argument is an object, otherwise it can be accessed directly. Note that, the overloaded binary operator function is a member function defined in the first object's class.

The following examples illustrate the overloading of binary operators:

```
complex operator + ( complex c1 );
int operator - ( int a );
void operator * ( complex c1 );
void operator / ( complex c1 );
complex operator += ( complex c1 );
```

Similar to unary operators, binary operators also have to return values so that cascaded assignment expressions can be formed. The programs illustrating the overloading of binary operators are discussed in the following sections.

13.9 Arithmetic Operators

Consider an example involving operations on complex numbers to illustrate the concept of binary operator overloading. Complex numbers consists of two parts: real part and imaginary part. It is represented as (x+iy), where x is the real part and y is the imaginary part. The process of performing the addition operation is illustrated below. Let c1, c2, and c3 be three complex numbers represented as follows:

```
c1 = x1 + i y1;
c2 = x2 + i y2;
```

The operation c3 = c1 + c2 is given by

```
c3 = ( c1.x1 + c2.x2 ) + i ( c1.y1 + c2.y2 );
```

The program complex1.cpp performs addition of complex numbers without operator overloading.

```
// complex1.cpp: Addition of Complex Numbers
#include <iostream.h>
class complex
{
   private:
      float real;    // real part of complex number
      float imag;    // imaginary part of complex number
   public:
      complex()   // no argument constructor
      {
         real = imag = 0.0;
      }
      void getdata()
      {
         cout << "Real Part ? ";
         cin >> real;
         cout << "Imag Part ? ";
         cin >> imag;
      }
      complex AddComplex( complex c2 );     // Add complex numbers
      void outdata( char *msg )                // display complex number
      {
         cout << endl << msg;
```

```
            cout << "(" << real;
            cout << ", " << imag << ")";
        }
};
// adds default and c2 complex objects
complex complex::AddComplex( complex c2 )
{
    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, c3 are object of complex class
    cout << "Enter Complex Number c1 .." << endl;
    c1.getdata();
    cout << "Enter Complex Number c2 .." << endl;
    c2.getdata();
    c3 = c1.AddComplex( c2 );     // add c1 and c2 and assign the result to c3
    c3.outdata( "c3 = c1.AddComplex( c2 ): ");
}
```

Run

```
Enter Complex Number c1 ..
Real Part ? 2.5
Imag Part ? 2.0
Enter Complex Number c2 ..
Real Part ? 3.0
Imag Part ? 1.5
c3 = c1.AddComplex( c2 ): (5.5, 3.5)
```

In `main()`, the statement

```
c3 = c1.AddComplex( c2 );
```

invokes the member function `AddComplex()` of the `c1` object's class and adds `c2` to it and then the returned result object is assigned to `c3`. By overloading the + operator, this clumsy and dense-looking statement can be represented in the simplified standard (usual) form as follows:

```
c3 = c1 + c2;
```

The program `complex2.cpp` illustrates the overloading of the binary operator + in order to perform addition of complex numbers.

```
// complex2.cpp: Complex Numbers operations with operator overloading
#include <iostream.h>
class complex
{
    private:
        float real;              // real part of complex number
        float imag;              // imaginary part of complex number
```

```
   public:
      complex()                  // no argument constructor
      {
         real = imag = 0.0;
      }
      void getdata()             // read complex number
      {
         cout << "Real Part ? ";
         cin >> real;
         cout << "Imag Part ? ";
         cin >> imag;
      }
      complex operator + ( complex c2 );  // complex addition
      void outdata( char *msg )     // display complex number
      {
         cout << endl << msg;
         cout << "(" << real;
         cout << ", " << imag << ")";
      }
};
// add default and c2 complex objects
complex complex::operator + ( complex c2 )
{
   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, c3 are object of complex class
   cout << "Enter Complex Number c1 .." << endl;
   c1.getdata();
   cout << "Enter Complex Number c2 .." << endl;
   c2.getdata();
   c3 = c1 + c2; // add c1 and c2 and assign the result to c3
   c3.outdata("c3 = c1 + c2: " ); // display result
}
```

Run

```
Enter Complex Number c1 ..
Real Part ? 2.5
Imag Part ? 2.0
Enter Complex Number c2 ..
Real Part ? 3.0
Imag Part ? 1.5
c3 = c1 + c2: (5.5, 3.5)
```

In the class `complex`, the `operator+()` function is declared as follows.

```
complex operator + ( complex c2 );
```

This function takes one explicit argument of type `complex` and returns the result of `complex` type. In a statement such as

```
c3 = c1 + c2;       // c3 = c1.operator+( c2 );
```

it is very important to understand the mechanism of returning a value and relating the arguments of the operator to its objects. When the compiler encounters such expressions, it examines the argument types of the operator. In this case, since the first argument is of type `complex`, the compiler realizes that it must invoke the operator member `+()` function defined in the `complex` class (Figure 13.5).

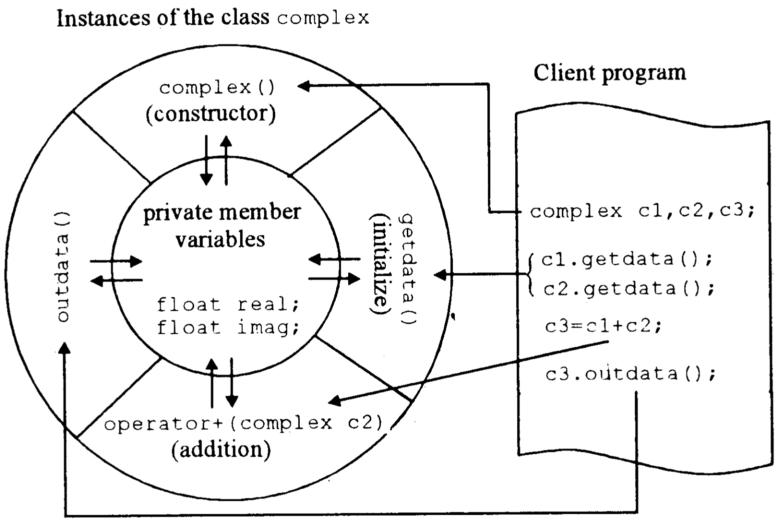


Figure 13.5: Complex numbers and operator overloading

The argument on the left side of the operator (`c1` in this case) is the object of a class having overloaded operator function as its member function. The object on the right side (`c2` in this case) of the operator is passed as the actual argument to the overloaded operator function. The operator returns a value (complex object `temp` in this case), which can be assigned to another object (c3 in this case) or can be used in other ways (as argument or term in an expression, etc.).

The expression `c1+c2` invokes operator `+()` member function, `c1` object's data members are accessed directly since, this is the object of which the operator function is a member. The right operand is treated as an argument to the function and its members are accessed using the member access dot operator (as `c2.real` and `c2.imag`).

In the overloading of binary operators, as a rule, the *left-hand* operand is used to invoke the operator function and the *right-hand* operand is passed as an argument to the operator function. The mechanism of handling operands of an overloaded binary operator is illustrated in Figure 13.6.

Similarly, functions can be created to overload other operators to perform addition, subtraction, multiplication, division, etc. The program `complex3.cpp` illustrates the overloading of various arithmetic operators for manipulating complex numbers.

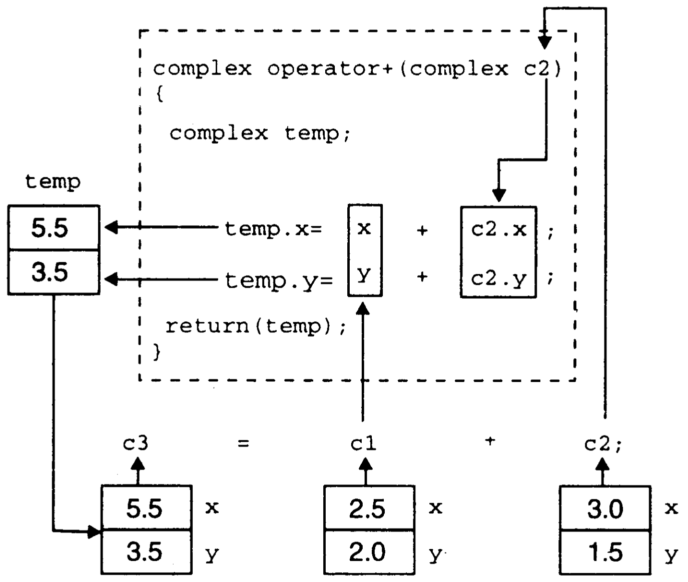


Figure 13.6: Operator overloading in class complex

```
// complex3.cpp: Manipulation of Complex Numbers
#include <iostream.h>
class complex
{
   private:
      float real;
      float imag;
   public:
      complex()
      {
         real = imag = 0;
      }
      void getdata()       // read complex number
      {
         cout << "Real Part ? ";
         cin >> real;
         cout << "Imag Part ? ";
         cin >> imag;
      }
      void outdata( char *msg )     // display complex number
      {
         cout << endl << msg;
         cout << "(" << real;
         cout << ", " << imag << ")";
      }
      complex operator + ( complex c2 );
```