C++ → C++ is an object oriented programming language. It was developed by Bjarne Stroustrup at AT & T Bell laboratories, in early 1980's. C++ is an extension of C with a major addition of the class construct feature of Simula67. C++ is a superset of C.

OOPS L
IV Sem
(CS)
Qs 2

An example →

```
#Include <iostream.h>          // Header file
Int main()
{
cout<<" Hello world \n";
return 0;
}
```

28/—

Output Operator → The statement

```
cout<<" Hello world \n";
```

Causes the string in quotation marks to be displayed on the screen. The identifier cout is a predefined object that represents the standard output stream in C++. The standard output stream represents the screen. It is also possible to redirect the output to other output device.

<< operator is called the insertion or put to operator. It inserts (or sends) the contents of the variable on its right to the object on its left.

<< → bit wise left shift operator [operator overloading]

Input Operator → The statement

```
cin>> n;
```

is an input statement and causes the program to wait for the user to type in a number. The number keyed is placed in variable n.

The identifier cin is a predefined object in C++ that corresponds to the standard input stream.

The operator >> is known as extraction or get from operator. It extracts the value from the keyboard and assigns the variable on its right.

Cascading of I/O operators- The statement

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

first sends the string "Sum=" to cout and then sends the value of sum. The multiple use of << in one statement is called cascading.

The iostream file- #include <iostream>

This directive causes the preprocessor to add the contents of the iostream file to the program. It contains declarations for the identifier cout and the operator <<. Some old versions of C++ use a header file called iostream.h. This is one of the changes introduced by ANSI C++.

Namespace- Namespace is a new concept introduced by ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. It is used as,

using namespace std;

Std is the namespace where ANSI C++ standard class libraries are defined.

Some operators in C++ → All C operators are valid in C++. In addition C++ adds some new operators eg.

::	scope resolution operator
delete	Memory release operator
endl	line feed operator
new	Memory allocation operator

Scope Resolution operator — C++ is a block structured language. The same variable name can be used to have different meanings in different blocks. eg.

```
{
int x=10;
}
{
int x=1;
}
```

The two declarations of x refer to two different memory locations containing different values.

In C, the global version of a variable can't be accessed from within the inner block. C++ resolves this by introducing a new operator :: called scope resolution operator. It is used as,

:: variable_name

```
#include <iostream.h>
int m=10;
int main()
{
 int m=20;
 {
  int k=m;
  int m=30;
  cout<<" k=" <<k <<"\n";          // k=20
  cout<<" m=" <<m<<"\n";          // m=30
  cout<<"::m=" <<::m <<"\n";      // ::m=10
 }
 cout<<"m=" <<m<<"\n";            //m=20
 cout<<" ::m=" <<::m<<"\n";       //::m=10
 return 0;
}
```

Memory Management operators- C uses malloc() and calloc() functions to allocate memory dynamically at run time. Similarly, it uses the function free() to free dynamically allocated memory. C++ supports these functions, it also defines two unary operators new and delete that perform the task of allocating and freeing the memory.

A data object created inside a block with new, will remain in existence until it is explicitly destroyed by using delete of

pointer variable = new datatype;

The new operator allocates sufficient memory to hold a data object of type data-type & returns the address of the object of

```
p= new int;
q= new float;
```

Here p & q should be of ~~type~~ pointer of type int & pointer of type float. It can also be written as,

```
int *p = new int;
float *q = new float;
```

Values can be assigned as,

```
*p = 25;
*q = 7.5;
```

Also it can be done,

```
int *p = new int (25);
float *q = new float (7.5);
```

new can also be used for any datatype including arrays, structures, classes etc.

```
int *p = new int[10];
```

creates a memory space for an array of 10 integers.

delete operator ⇒ when a data object is no longer needed, it is destroyed to release the memory space for reuse.

```
delete pointer_variable;
```

eg.

```
delete p;
delete q;
```

for array,

```
delete [size] pointer_variable;
delete [] p;
```

Abstract Datatypes – A class is a way to bind the data and its associated functions together. It allows the data and functions to be hidden from external use. A class specification has two parts:

① Class declaration → type and scope of members.

② Class function definitions. – class function implementation.

General form of class declaration

```
class class_name
{
private:
    variable declarations;      → data members
    function declarations;      → member functions
public:
    variable declarations;
    function declarations;
};
```

visibility labels

private → These members can be accessed only from within the class.

public → These members can be accessed from outside the class also.

Creating Objects → Once a class is declared, we can create variables of that type by using the classname. eg.

```
Classname objectname;
```

The necessary memory space is allocated to an object at this stage. Objects can also be created like

```
class student
{
---
---
} s1, s2, s3;
```

Array of objects

Accessing Class Members-

Objectname. function name (actual arguments);

eg. s1. getdata ();
s1. putdata ();

Defining Member functions-

① Outside the class

Returntype classname :: functionname (argument declaration)
{
==
}

② Inside the class-

```
class A
{
  int a;
public:
  void getdata()      //Inline function
  {
  ---
  }
};
```

Making an outside function inline →

Inline function- When a function is called, it takes a lot of extra time in executing a series of instruction for tasks such as jumping to the function, saving registers, pushing arguments into stack & returning to the calling function. In C, the solution is macros.

C++ has a different solution. C++ proposes a new feature called inline function. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code.

We can define a member function outside the class & still make it inline by using inline keyword.

```
class A
{
==
public:
  void getdata();
};
inline void A :: getdata()
{
==
}
```

Memory Allocation for objects - The memory space for objects is allocated when they are declared & not when the class is specified. The member functions are created & placed in memory only once when they are defined as a part of a class specification. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created. Only space for member variables is allocated separately for each object. Separate memory locations for the objects are essential, bcz the member variables will hold different data values for different objects.

Static Data Members ⇒ A static member variable has certain special characteristics. These are -

① It is initialized to zero when the first object of its class is created.

② Only one copy of that member is created for the entire class and is shared by all the objects of that class.

③ It is visible only within the class, but its lifetime is the entire program.

```
#include<iostream>
using namespace std;
class Item
{
static int count;
 int number;
public:
 void getdata (int a)
 {
   number = a;
    count++;
 }
void getcount (void)
{
 cout<< "count: ";
 cout << count <<"\n";
};
int item :: count=0;
int main()
{
Item a, b, c;
a.getcount();
b.getcount();
c.getcount();
a.getdata (100);
b.getdata (200);
c.getdata (300);
cout<<"After reading data";
a.getcount();
b.getcount();
c.getcount();
return 0;
}
```

o/p :-

```
count : 0
count : 0
count : 0
After reading data
count : 3
count : 3
count : 3
```

The type and scope of each static member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object. static data members are also known as class variables.

Static Member functions - A member function that is declared static has the following properties-

① A static function can have access to only other static members declared in the same class.

② A static member function can be called using the classname

```
class test
{
  int code;
  static int count;
public:
void setcode (void)
{
 code = ++ count;
}
void showcode (void)
{
 cout<< " object : " << code << "\n";
}
static void showcount (void)
{
 cout << "count : " << count << "\n";
}};
int test :: count;
```

```
int main()
{
test t1, t2;
  t1.setcode();
  t2.setcode();
test::showcount();
  test t3;
  t3.setcode();
  test::showcount();
  t1.showcode();
  t2.showcode();
  t3.showcode();
  return 0;
}
```

Op

count: 2
count: 3
Object: 1
Object: 2
Object: 3

Objects as Function Arguments → Objects can be passed in two ways-

① Pass-by-value
② Pass-by-reference

Pass-by-value - A copy of the object is passed to the function any changes made to the object inside the function don't affect the object to call the function.

Pass-by-reference - Address of the object is passed, the called function works directly on the actual object used in the call.

eg.

```
class A
{
int a, b;
public:
  void getdata (int c, int d)
  {
   a = c;
   b = d;
  }
  Void putdata ()
  {
  Cout << a << b << "\n";
  }
  void sum( A, A);
};
void A :: sum (A ob1, A ob2)
{
  a = ob1.a + ob2.a;
  b = ob1.b + ob2.b;
}
```

```
main()
{
A obj1, obj2, obj3;
obj1.getdata(2, 4);
obj2.getdata( 3, 6);
obj3.sum(obj1, obj2);
cout<<"obj1= "<< obj1.putdata();
cout << "obj2= " << obj2.putdata();
Cout << "obj3= " << obj3.putdata();
}
```

Friend functions → A nonmember function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. eg.

```
class ABC
{
  ==
public:
  ==
friend void xyz(void);
};
```

The function definition does not ~~need~~ use either the keyword friend or the scope operator ::. A function can be declared as a friend in any number of classes. A friend function possesses certain special characteristics—

(1) It is not in the scope of the class to which it has been declared as friend.
(2) It can't be called using the object of that class.
(3) It can be invoked like a normal function.
(4) It can be declared either in the public or the private part of a class without affecting its meaning.
(5) It has objects as arguments.

Friend function as Bridge-

```
class A;
class B
{
  int b;
public:
  void set()
  {
    b=10;
  }
friend void Add(B, A);
};
class A
{
  int a;
public:
  void set()
  {
    a=20;
  }
friend void Add(B,A);
};
void Add(B b1, A a1)
{
  int c;
  c = b1.b + a1.a;
  cout << "Sum is=" << c;
}
```

```
int main()
{
 A a2;
a2.set();
 B b2;
b2.set();
add(b2, a2);
return 0;
}
```

```
class sample
{
int a;
int b;
public:
void setvalue()
{
a=25;
b=40;
}
friend float mean(sample s);
};
float mean(sample s)
{
return float(s.a+s.b)/2.0;
}
int main()
{
sample X;
X.setvalue();
cout<<" mean value="<<mean(X)<<"\n";
return 0;
}
```

Friend classes - We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a friend class. This can be specified as -

```
class Z
{
friend class X;
};
```

Alan Kay, one of the promoters of the object-oriented paradigm and the principal designer of Smalltalk, has said: "*As complexity increases, architecture dominates the basic material*". To build today's complex software it is just not enough to put together a sequence of programming statements and sets of procedures and modules; we need to incorporate sound construction techniques and program structures that are easy to comprehend, implement and modify.

Since the invention of the computer, many programming approaches have been tried. These include techniques such as *modular programming, top-down programming, bottom-up programming and structured programming*. The primary motivation in each has been the concern to handle the increasing complexity of programs that are reliable and maintainable. These techniques have become popular among programmers over the last two decades.

With the advent of languages such as C, structured programming became very popular and was the main technique of the 1980s. Structured programming was a powerful tool that enabled programmers to write moderately complex programs fairly easily. However, as the programs grew larger, even the structured approach failed to show the desired results in terms of bug-free, easy-to-maintain, and reusable programs.

Object-Oriented Programming (OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language. However, not all languages are suitable to implement the OOP concepts easily.

1.3 A LOOK AT PROCEDURE-ORIENTED PROGRAMMING

Conventional programming, using high level languages such as COBOL, FORTRAN and C, is commonly known as *procedure-oriented programming (POP)*. In the procedure-oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks. The primary focus is on functions. A typical program structure for procedural programming is shown in Fig. 1.4. The technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem.

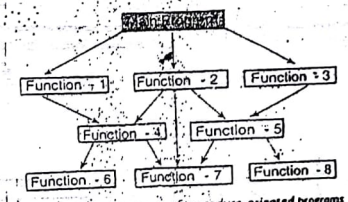


Fig. 1.4 *Typical structure of procedure-oriented programs*

Procedure-oriented programming basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as *functions*. We normally use a *flowchart* to organize these actions and represent the flow of control from one action to another. While we concentrate on the development of functions, very little attention is given to the data that are being used by various functions. What happens to the data? How are they affected by the functions that work on them?

In a multi-function program, many important data items are placed as *global* so that they may be accessed by all the functions. Each function may have its own *local data*. Figure 1.5 shows the relationship of data and functions in a procedure-oriented program.

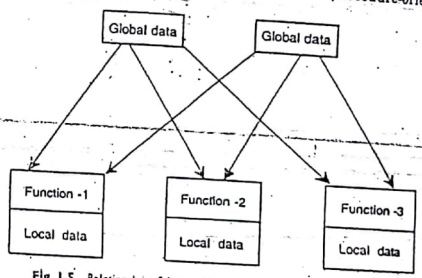


Fig. 1.5 *Relationship of data and functions in procedural programming*

Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.

Another serious drawback with the procedural approach is that it does not model real world problems very well. This is because functions are action-oriented and do not really correspond to the elements of the problem.

Some characteristics exhibited by procedure-oriented programming are:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs *top-down* approach in program design.

1.4 OBJECT-ORIENTED PROGRAMMING PARADIGM

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions. OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects. The organization of data and functions in object-oriented programs is shown in Fig. 1.6. The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.

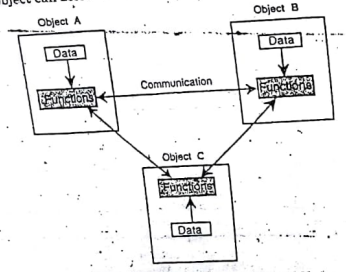


Fig 1.6 Organization of data and functions in OOP

Some of the striking features of object-oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- *Follows bottom-up* approach in program design.

Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people. It is therefore important to have a working definition of object-oriented programming before we proceed further. We define object-oriented programming as *an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.*" Thus, an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

1.5 BASIC CONCEPTS OF OBJECT-ORIENTED PROGRAMMING

It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

We shall discuss these concepts in some detail in this Section.

Objects

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in terms of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in C.

When a program is executed, the objects interact by sending messages to one another. For example, if "customer" and "account" are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance. Each object contains data, and code to manipulate the data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted, and the type of response returned by the objects. Although different authors represent them differently, Fig. 1.7 shows two notations that are popularly used in object-oriented analysis and design.

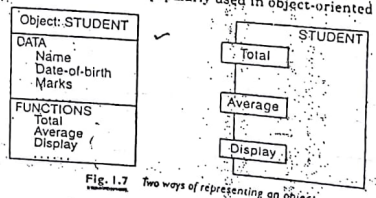


Fig. 1.7 Two ways of representing an object

Classes

We just mentioned that objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a *class*. In fact, objects are variables of the type *class*. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects of similar type. For example, mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language. The syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement

```
fruit mango;
```

will create an object mango belonging to the class fruit.

Data Abstraction and Encapsulation

The wrapping up of data and functions into a single unit (called class) is known as *encapsulation*. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called *data hiding or information hiding*.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract *attributes* such as size, weight and cost, and *functions* to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called *data members* because they hold information. The functions that operate on these data are sometimes called *methods or member functions*.

Since the classes use the concept of data abstraction, they are known as *Abstract Data Types* (ADT).

Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of *hierarchical classification*. For example, the bird 'robin' is a part of the class 'flying bird' which is again a part of the class 'bird'. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in Fig. 1.8.

In OOP, the concept of inheritance provides the idea of *reusability*. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it

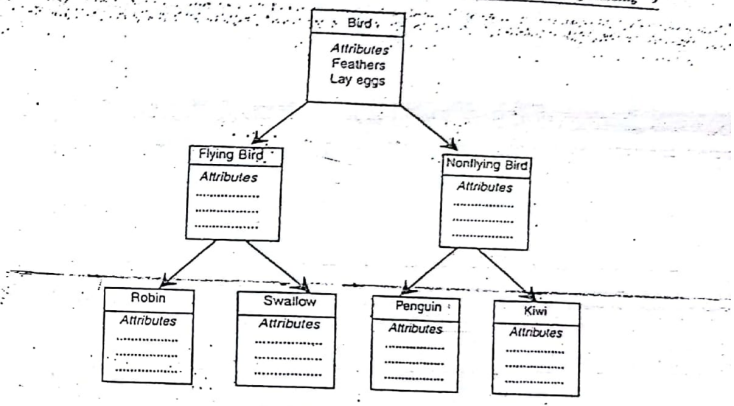


Fig. 1.8 *Property inheritance*

allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side-effects into the rest of the classes.

Note that each sub-class defines only those features that are unique to it. Without the use of classification, each class would have to explicitly include all of its features.

Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviours in different instances is known as *operator overloading*.

Figure 1.9 illustrates that a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having

several different meanings depending on the context. Using a single function name to perform different types of tasks is known as *function overloading*.

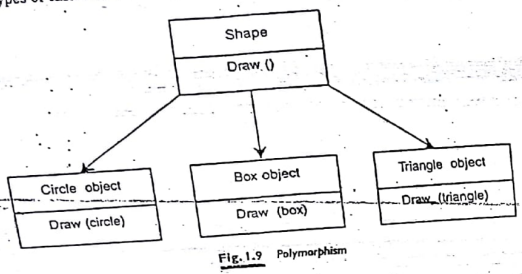


Fig. 1.9 Polymorphism

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. *Dynamic binding* (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Consider the procedure "draw" in Fig. 1.9. By inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

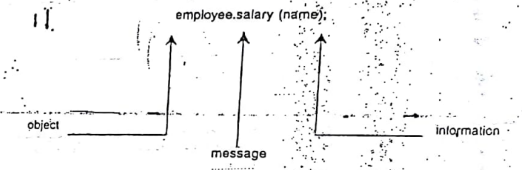
Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

1. Creating classes that define objects and their behaviour,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result. *Message passing* involves specifying the name of the object, the name of the function (message) and the information to be sent. Example:



Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

1.6 BENEFITS OF OOP

OOP offers several benefits to both the program designer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

Ques:- Differentiate between C and C++.

Ans:-

C	C++
(1) C is a middle level language.	- C++ is a high level language
(2) C is a structured language	- C++ is an object oriented language
(3) C doesn't support privacy of data	- C++ supports privacy of data using the concept of data abstraction
(4) A C program have a '.c' extension	- A C++ program have a '.cpp' extension.
C is a subset of C++	- C++ is a superset of C
(6) In 'C' we include 'stdio.h' for input / output operations	- In C++ we include 'iostream.h' for input, output operations
(7) In 'C' 'scanf' is used to get data from Keyboard	- In C++ 'cin' (console input) is used to get data from Keyboard.

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer. There are a number of issues that need to be tackled to reap some of the benefits stated above. For instance, object libraries must be available for reuse. The technology is still developing and current products may be superseded quickly. Strict controls and protocols need to be developed if reuse is not to be compromised.

Developing a software that is easy to use makes it hard to build. It is hoped that the object-oriented programming tools would help manage this problem.

1.7 OBJECT-ORIENTED LANGUAGES

Object-oriented programming is not the right of any particular language. Like structured programming, OOP concepts can be implemented using languages such as C and Pascal. However, programming becomes clumsy and may generate confusion when the programs grow large. A language that is specially designed to support the OOP concepts makes it easier to implement them.

The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

1. Object-based programming languages, and
2. Object-oriented programming languages.

Object-based programming is the style of programming that primarily supports encapsulation and object identity. Major features that are required for object-based programming are:

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

Languages that support programming with objects are said to be object-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

Object-oriented programming incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statement:

Object-based features + inheritance + dynamic binding

Languages that support these features include C++, Smalltalk, Object Pascal and Java. There are a large number of object-based and object-oriented programming languages. Table 1.1 lists some popular general purpose OOP languages and their characteristics.

Table 1.1 Characteristics of Some OOP Languages

Characteristics	Simula *	Smalltalk *	Objective C	C++	Ada **	Object Pascal	Turbo Pascal	Eiffel *	Java *
Binding (early or late)	Both	Late	Both	Both	Early	Late	Early	Early	Both
Polymorphism	✓	✓	✓	✓	✓	✓	✓	✓	✓
Data hiding	✓	✓	✓	✓	✓	✓	✓	✓	✓
Concurrency	✓	✓	✓	✓	✓	✓	✓	✓	✓
Inheritance	✓	Poor	Poor	Poor	Difficult	No	No	Promised	✓
Multiple Inheritance	No	✓	✓	✓	No	✓	✓	✓	✓
Garbage Collection	✓	✓	✓	✓	No	—	—	✓	No
Persistence	No	Promised	No	No	No	✓	✓	✓	✓
Genericity	No	No	No	✓	like 3GL	No	No	Some Support	✓
Object Libraries	✓	✓	✓	✓	Not much	No	No	✓	No

* *Pure object-oriented languages*
** *Object-based languages*
Others are extended conventional languages

As seen from Table 1.1, all languages provide for polymorphism and data hiding. However, many of them do not provide facilities for concurrency, persistence and genericity. Eiffel, Ada and C++ provide generic facility which is an important construct for supporting reuse. However, persistence (a process of storing objects) is not fully supported by any of them. In Smalltalk, though the entire current execution state can be saved to disk, yet the individual objects cannot be saved to an external file.

Commercially, C++ is only 10 years old, Smalltalk and Objective C 13 years old, and Java only 5 years old. Although Simula has existed for more than two decades, it has spent most of its life in a research environment. The field is so new, however, that it should not be judged too harshly.

Use of a particular language depends on characteristics and requirements of an application, organizational impact of the choice, and reuse of the existing programs. C++ has now become the most successful, practical, general purpose OOP language, and is widely used in industry today.

1.8 APPLICATIONS OF OOP

OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP. Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as *windows*. Hundreds of windowing systems have been developed, using the OOP techniques.

(8) In 'C' we can't use 'clrscr()' before the declaration of variables in main().	- In C++ we can use 'clrscr()' before and after the declaration of variables in main().
(9) In 'C' we use 'printf' to show output on terminal.	- In C++ we use 'cout' to show output on terminal.
(10) In 'C', one has to remember so many format strings like %d, %f etc that are used in scanf and printf functions.	- In C++, there is no need to learn such format strings.
(11) In 'C' we use & (Ampersand) operator to get data from keyboard using scanf.	- In C++, compiler takes care of all the addresses / locations and we need not to use & operator.

(12) In 'C' structures, structure variable is defined as:
struct abc
{
};
struct abc s;
↳ structure variable
	- In C++ structures, structure variable is defined as:
	struct abc
	{
	};
	abc s;
	↳ structure variable

(13) In 'c' structures, we can have only data types.

(14) In 'c' we don't have classes, only structures are there which have data as public by default.

(15) In 'c' we can't initialise variables inside the definition of structure.

(16) In 'c' the width of variable name is limited to 32 ~~characters~~.

(17) In 'c' we use 'calloc' and 'malloc' as dynamic memory allocator functions.

(18) In 'c' we have '\n' for new line.

In c++ structures we can have data types as well as functions called Data functions.

In c++ we have classes where we can have data as private by default, public and protected.

- In c++ we can initialise variables inside the definition of structure.

- In c++ there is no upper limit to the width of variable name.

- In c++ we use 'new' and 'Delete' as dynamic memory allocator functions.

- In c++ we have '\n' as well as 'Endl' for moving to a next line.

(19) In 'c' we include comments as:- /* comment ... */	- In c++ we can include comment as:- // comment and /* comment ... */ as c++ is a superset of c.
(20) In 'c' after comment we can also continue our programming code.	- In c++ nothing can be written after comment because it will automatically move the data to the next line.
(21) c doesnot support function overloading.	- c++ supports function overloading.
(22) In c memory management operator requires the typecast.	In c++, memory management operation all automatically typecast.

6

Constructors and Destructors

Key Concepts

- Constructing objects
- Constructors
- Constructor overloading
- Default argument constructor
- Copy constructor
- Constructing matrix objects
- Automatic initialization
- Parameterized constructors
- Default constructor
- Dynamic initialization
- Dynamic constuctor
- Destructors

6.1 INTRODUCTION

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as **putdata()** and **setvalue()** to provide initial values to the private member variables. For example, the following statement

```
A.input();
```

invokes the member function **input()**, which assigns the initial values to the data items of object A. Similarly, the statement

```
x.getdata(100,299.95);
```

passes the initial values as arguments to the function **getdata()**, where these values are assigned to the private variables of object x. All these 'function call' statements are used with the appropriate objects that have already been created. These functions cannot be used to initialize the member variables at the time of creation of their objects.

Providing the initial values as described above does not conform with the philosophy of the language. We stated earlier that one of the aims of C++ is to create user-defined data types such as class, that behave very similar to the built-in types. This means that we should be able to initialize a class type variable (object) when it is declared, much the same way as the initialization of an ordinary variable. For example,

```
int m = 20;
float x = 5.75;
```

are valid initialization statements for basic data types.

Similarly, when a variable of built-in type goes out of scope, the compiler automatically destroys the variable. But it has not happened with the objects we have so far studied. It is therefore clear that some more features of classes need to be explored that would enable us to initialize the objects when they are created and destroy them when their presence is no longer necessary.

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as *automatic initialization* of objects. It also provides another member function called the *destructor* that destroys the objects when they are no longer required.

6.2 CONSTRUCTORS

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

A constructor is declared and defined as follows:

```
// class with a constructor

class integer
{
      int m, n;
  public:
      integer(void);              // constructor declared
      .....
      .....
};
integer :: integer(void)          // constructor defined
{
    m = 0; n = 0;
}
```

are created. The constructors that can take arguments are called *parameterized constructors*.

The constructor integer() may be modified to take arguments as shown below:

```
class integer
{
    int m, n;
  public:
    integer(int x, int y);      // parameterized constructor
    .....
    .....
};
integer :: integer(int x, int y)
{
    m = x; n = y;
}
```

When a constructor has been parameterized, the object declaration statement such as

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the first method:

```
integer int1 = integer(0,100);    // explicit call
```

This statement creates an integer object int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100);              // implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor. Program 6.1 demonstrates the passing of arguments to the constructor functions.

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration

```
integer int1;          // object int1 created
```

not only creates the object int1 of type integer but also initializes its data members m and n to zero. There is no need to write any statement to invoke the constructor function (as we do with the normal member functions). If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately. This would be very inconvenient, if there are a large number of objects.

A constructor that accepts no parameters is called the *default constructor*. The default constructor for class A is A::A(). If no such constructor is defined, then the compiler supplies a default constructor. Therefore a statement such as

```
A a;
```

invokes the default constructor of the compiler to create the object a.

The constructor functions have some special characteristics. These are :

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be **virtual**. (Meaning of virtual will be discussed later in Chapter 9.)
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make 'implicit calls' to the operators **new** and **delete** when memory allocation is required.

Remember, when a constructor is declared for a class, initialization of the class objects becomes mandatory.

6.3 PARAMETERIZED CONSTRUCTORS

The constructor integer(), defined above, initializes the data members of all the objects to zero. However, in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects

CLASS WITH CONSTRUCTORS

```
#include <iostream>

using namespace std;

class integer
{
    int m, n;
  public:
    integer(int, int);              // constructor declared

    void display(void)
    {
      cout << " m = " << m << "\n";
      cout << " n = " << n << "\n";
    }
};

integer :: integer(int x, int y)    // constructor defined
{
    m = x; n = y;
}

int main()
{
  integer int1(0,100);              // constructor called implicitly

  integer int2 = integer(25, 75);   // constructor called explicitly

  cout << "\nOBJECT1" << "\n";
  int1.display();

  cout << "\nOBJECT2" << "\n";
  int2.display();

  return 0;
}
```

PROGRAM 6.1

Program 6.1 displays the following output:

```
OBJECT1
m = 0
n = 100
```

```
OBJECT2
m = 25
n = 75
```

The constructor functions can also be defined as inline functions. Example:

```
class integer
{
    int m, n;
  public:
    integer(int x, int y)  // Inline constructor
    {
        m = x; y = n;
    }
    .....
    .....
};
```

The parameters of a constructor can be of any type except that of the class to which it belongs. For example,

```
class A
{
    .....
    .....
  public:
    A(A);
};
```

is illegal.

However, a constructor can accept a *reference* to its own class as a parameter. Thus, the statement

```
Class A
{
    .....
    .....
  public:
    A(A&);
};
```

is valid. In such cases, the constructor is called the *copy constructor*.

6.4 MULTIPLE CONSTRUCTORS IN A CLASS

Constructor overloading.

So far we have used two kinds of constructors. They are:

```
integer();              // No arguments
integer(int, int);      // Two arguments
```

In the first case, the constructor itself supplies the data values and no values are passed by the calling program. In the second case, the function call passes the appropriate values from main(). C++ permits us to use both these constructors in the same class. For example, we could define a class as follows:

```
class integer
{
      int m, n;
  public:
     integer(){m=0; n=0;}              // constructor 1
     integer(int a, int b)
     {m = a; n = b;}                   // constructor 2
     integer(integer & i)
     {m = i.m;  n = i.n;}              // constructor 3
};
```

This declares three constructors for an integer object. The first constructor receives no arguments, the second receives two integer arguments and the third receives one integer object as an argument. For example, the declaration

```
integer I1;
```

would automatically invoke the first constructor and set both m and n of I1 to zero. The statement

```
integer I2(20,40);
```

would call the second constructor which will initialize the data members m and n of I2 to 20 and 40 respectively. Finally, the statement

```
integer I3(I2);
```

would invoke the third constructor which copies the values of I2 into I3. In other words, it sets the value of every data element of I3 to the value of the corresponding data element of I2. As mentioned earlier, such a constructor is called the *copy constructor*. We learned in Chapter 4 that the process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor function is defined in a class, we say that the constructor is overloaded.

Program 6.2 shows the use of overloaded constructors.

OVERLOADED CONSTRUCTORS

```
#include <iostream>

using namespace std;

class complex
{
    float x, y;
  public:
    complex(){ }                          // constructor no arg
    complex(float a) {x = y = a;}         // constructor-one arg
    complex(float real, float imag)       // constructor-two args
    {x = real; y = imag;}

    friend complex sum(complex, complex);
    friend void show(complex);
};
complex sum(complex c1, complex c2)       // friend
{
    complex c3;
    c3.x = c1.x + c2.x;
    c3.y = c1.y + c2.y;
    return(c3);
}
void show(complex c)                      // friend
{
    cout << c.x << " + j" << c.y << "\n";
}

int main()
{
    complex A(2.7, 3.5);                  // define & initialize
    complex B(1.6);                       // define & initialize
    complex C;                            // define

    C = sum(A, B);                        // sum() is a friend
    cout << "A = "; show(A);              // show() is also friend
    cout << "B = "; show(B);
    cout << "C = "; show(C);

// Another way to give initial values (second method)
    complex P,Q,R;                        // define P, Q and R
```

(Contd)

```
P = complex(2.5,3.9);     // initialize P
Q = complex(1.6,2.5);     // initialize Q
R = sum(P,Q);

cout << "\n";
cout << "P = "; show(P);
cout << "Q = "; show(Q);
cout << "R = "; show(R);

return 0;
}
```

PROGRAM 6.2

The output of Program 6.2 would be:

```
A = 2.7 + j3.5
B = 1.6 + j1.6
C = 4.3 + j5.1

P = 2.5 + j3.9
Q = 1.6 + j2.5
R = 4.1 + j6.4
```

There are three constructors in the class complex. The first constructor, which takes no arguments, is used to create objects which are not initialized; the second which takes one argument, is used to create objects and initialize them; and the third, which takes two arguments, is also used to create objects and initialize them to specific values. Note that the second method of initializing values looks better.

Let us look at the first constructor again.

```
complex(){ }
```

It contains the empty body and does not do anything. We just stated that this is used t create objects without any initial values. Remember, we have defined objects in the earlie examples without using such a constructor. Why do we need this constructor now?. As pointe out earlier, C++ compiler has an *implicit constructor* which creates objects, even though i was not defined in the class.

This works fine as long as we do not use any other constructors in the class. However, one we define a constructor, we must also define the "do-nothing" implicit constructor. Thi constructor will not do anything and is defined just to satisfy the compiler.

6.5 CONSTRUCTORS WITH DEFAULT ARGUMENTS

It is possible to define constructors with default arguments. For example, the constructor complex() can be declared as follows:

```
complex(float real, float imag=0);
```

The default value of the argument imag is zero. Then, the statement

```
complex C(5.0);
```

assigns the value 5.0 to the real variable and 0.0 to imag (by default). However, the statement

```
complex C(2.0,3.0);
```

assigns 2.0 to real and 3.0 to imag. The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor A::A() and the default argument constructor A::A(int = 0). The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

```
A a;
```

The ambiguity is whether to 'call' A::A() or A::A(int = 0).

6.6 DYNAMIC INITIALIZATION OF OBJECTS

Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during run time. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Consider the long term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment. Program 6.3 illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

DYNAMIC INITIALIZATION OF CONSTRUCTORS

```
// Long-term fixed deposit system
#include <iostream>
using namespace std;
```

(Contd)

```
class Fixed_deposit
{
        long int P_amount;          // Principal amount
        int      Years;             // Period of investment
        float    Rate;              // Interest rate
        float    R_value;           // Return value of amount
public:
        Fixed_deposit() { }
        Fixed_deposit(long int p, int y, float r=0.12);
        Fixed_deposit(long int p, int y, int r);
        void display(void);
};
Fixed_deposit :: Fixed_deposit(long int p, int y, float r)
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;
        for(int i = 1; i <= y; i++)
                R_value = R_value * (1.0 + r);
}

Fixed_deposit :: Fixed_deposit(long int p, int y, int r)
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;

        for(int i=1; i<=y; i++)
                R_value = R_value*(1.0+float(r)/100);
}

void Fixed_deposit :: display(void)
{
        cout << "\n"
             << "Principal Amount = " << P_amount << "\n"
             << "Return Value     = " << R_value  << "\n";
}

int main()
{
        Fixed_deposit FD1, FD2, FD3;  // deposits created

        long int p;                   // principal amount
```

(Cont

```
    int     y;                          // investment period, years
    float   r;                          // interest rate, decimal form
    int     R;                          // interest rate, percent form

    cout << "Enter amount,period,interest rate(in percent)"<<"\n";
    cin >> p >> y >> R;
    FD1 = Fixed_deposit(p,y,R);

    cout << "Enter amount,period,interest rate(decimal form)" << "\n";
    cin >> p >> y >> r;
    FD2 = Fixed_deposit(p,y,r);

    cout << "Enter amount and period" << "\n";
    cin >> p >> y;
    FD3 = Fixed_deposit(p,y);

    cout << "\nDeposit 1";
    FD1.display();

    cout << "\nDeposit 2";
    FD2.display();

    cout << "\nDeposit 3";
    FD3.display();

    return 0;
}
```

PROGRAM 6.3

The output of Program 6.3 would be:

```
Enter amount,period,interest rate(in percent)
10000 3 18
Enter amount,period,interest rate(in decimal form)
10000 3 0.18
Enter amount and period
10000 3

Deposit 1
Principal Amount = 10000
Return Value     = 16430.3

Deposit 2
Principal Amount = 10000
Return Value     = 16430.3
```

The member function **join( )** concatenates two strings. It estimates the combined length of the strings to be joined, allocates memory for the combined string and then creates the same using the string functions **strcpy( )** and **strcat( )**. Note that in the function **join( )**, **length** and **name** are members of the object that calls the function, while **a.length** and **a.name** are members of the argument object **a**. The **main( )** function program concatenates three strings into one string. The output is as shown below:

```
Joseph Louis Lagrange
```

6.9 CONSTRUCTING TWO-DIMENSIONAL ARRAYS

We can construct matrix variables using the class type objects. The example in Program 6.6 illustrates how to construct a matrix of size m x n.

```
// CONSTRUCTING MATRIX OBJECTS
#include <iostream>
using namespace std;

class matrix
{
        int **p;          // pointer to matrix
        int d1,d2;        // dimensions
    public:
        matrix(int x, int y);
        void get_element(int i, int j, int value)
          {p[i][j]=value;}
        int & put_element(int i, int j)
          {return p[i][j];}
};
matrix :: matrix(int x, int y)
{
        d1 = x;
        d2 = y;
        p = new int *[d1];              // creates an array pointer
        for(int i = 0; i < d1; i++)
            p[i] = new int[d2];         // creates space for each row
}

int main()
{
        int m, n;

        cout << "Enter size of matrix: ";
        cin >> m >> n;
```

(Contd)

```
    matrix A(m,n);          // matrix object A constructed
    cout << "Enter matrix elements row by row \n";
    int i, j, value;

    for(i = 0; i < m; i++)
        for(j = 0; j < n; j++)
        {
                cin >> value;
                A.get_element(i,j,value);
        }
    cout << "\n";
    cout << A.put_element(1,2);

    return 0;
}
```

PROGRAM 6.6

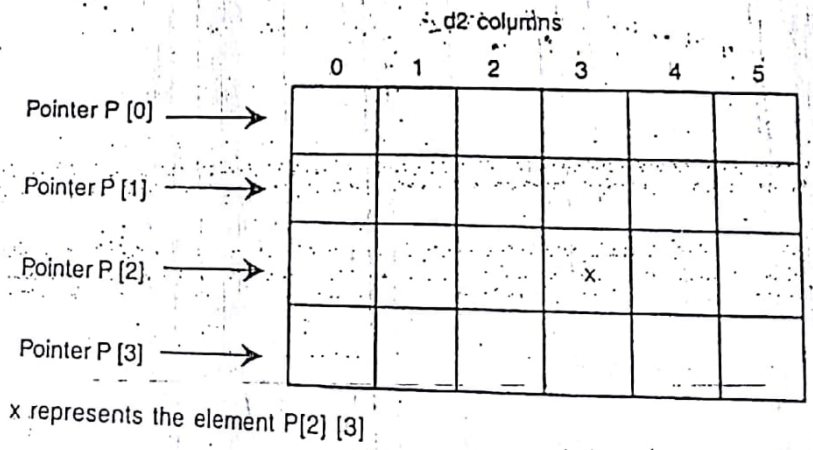
The output of a sample run of Program 6.6 is as follows.

```
Enter size of matrix: 3 4
Enter matrix elements row by row
11 12 13 14
15 16 17 18
19 20 21 22

17
```

17 is the value of the element (1,2).

The constructor first creates a vector pointer to an **int** of size **d1**. Then, it allocates, iteratively an **int** type vector of size **d2** pointed at by each element **p[i]**. Thus, space for the elements of a d1 X d2 matrix is allocated from free store as shown below:



x represents the element P[2] [3]

6.10 const OBJECTS

We may create and use constant objects using **const** keyword before object declaration. For example, we may create X as a constant object of the class **matrix** as follows:

```
const matrix X(m,n); // object X is constant
```

Any attempt to modify the values of m and n will generate compile-time error. Further, a constant object can call only **const** member functions. As we know, a **const** member is a function prototype or function definition where the keyword const appears after the function's signature.

Whenever **const** objects try to invoke non-**const** member functions, the compiler generates errors.

6.11 DESTRUCTORS

A *destructor*, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example, the destructor for the class integer can be defined as shown below:

```
~integer(){ }
```

A destructor never takes any argument nor does it return any value. It will be invoke implicitly by the compiler upon exit from the program (or block or function as the case ma be) to clean up storage that is no longer accessible. It is a good practice to declare destructo in a program since it releases memory space for future use.

Whenever **new** is used to allocate memory in the constructors, we should use **delete** free that memory. For example, the destructor for the **matrix** class discussed above may defined as follows:

```
matrix :: ~matrix()
{
    for(int i=0; i<dl; i++)
    delete p[i];
    delete p;
}
```

This is required because when the pointers to objects go out of scope, a destructor is not implicitly.

The example below illustrates that the destructor has been invoked implicitly compiler.

IMPLEMENTATION OF DESTRUCTORS:

```
#include <iostream>

using namespace std;

int count = 0;

class alpha
{
  public:
    alpha()
    {
        count++;
        cout << "\nNo.of object created " << count;
    }

    ~alpha()
    {
        cout << "\nNo.of object destroyed " << count;
        count--;
    }
};

int main()
{
    cout << "\n\nENTER MAIN\n";

    alpha A1, A2, A3, A4;
    {
        cout << "\n\nENTER BLOCK1\n";
        alpha A5;
    }

    {
        cout << "\n\nENTER BLOCK2\n";
        alpha A6;
    }
    cout << "\n\nRE-ENTER MAIN\n";

    return 0;
}
```

PROGRAM 6.7

Throughout the history of computing, there has been an ongoing debate concerning the best way to manage the use of dynamically allocated memory. Dynamically allocated memory is memory that is obtained during runtime from the *heap*, which is a region of free memory that is available for program use. The heap is also commonly referred to as *free store* or *dynamic memory*. Dynamic allocation is important because it enables a program to obtain, use, release, and then reuse memory during execution. Because nearly all real-world programs use dynamic allocation in some form, the way it is managed has a profound effect on the architecture and performance of programs.

In general, there are two ways that dynamic memory is handled. The first is the manual approach, in which the programmer must explicitly release unused memory in order to make it available for reuse. The second relies on an automated approach, commonly referred to as *garbage collection*, in which memory is automatically recycled when it is no longer needed. There are advantages and disadvantages to both approaches, and the favored strategy has shifted between the two over time.

C++ uses the manual approach to managing dynamic memory. Garbage collection is the mechanism employed by Java and C#. Given that Java and C# are newer languages, the current trend in computer language design seems to be toward garbage collection. This does not mean, however, that the C++ programmer is left on the "wrong side of history." Because of the power built into C++, it is possible—even easy—to create a garbage collector for C++. Thus, the C++ programmer can have the best of both worlds: manual control of dynamic allocation when needed and automatic garbage collection when desired.

This chapter develops a complete garbage collection subsystem for C++. At the outset, it is important to understand that the garbage collector does not replace C++'s built-in approach to dynamic allocation. Rather, it supplements it. Thus, both the manual and garbage collection systems can be used within the same program.

Aside from being a useful (and fascinating) piece of code in itself, a garbage collector was chosen for the first example in this book because it clearly shows the unsurpassed power of C++. Through the use of template classes, operator overloading, and C++'s inherent ability to handle the low-level elements upon which the computer operates, such as memory addresses, it is possible to transparently add a core feature to C++. For most other languages, changing the way that dynamic allocation is handled would require a change to the compiler itself. However, because of the unparalleled power that C++ gives the programmer, this task can be accomplished at the source code level.

The garbage collector also shows how a new type can be defined and fully integrated into the C++ programming environment. Such *type extensibility* is a key component of C++, and it's one that is often overlooked. Finally, the garbage collector testifies to C++'s ability to "get close to the machine" because it manipulates and manages pointers. Unlike some other languages which prevent access to the low-level details, C++ lets the programmer get as close to the hardware as necessary.

that memory is no longer needed, it is released by **delete**. Thus, each dynamic allocation follows this sequence:

```
p = new some object;
// ...
delete p;
```

In general, each use of **new** must be balanced by a matching **delete**. If **delete** is not used, the memory is not released, even if that memory is no longer needed by your program.

Garbage collection differs from the manual approach in one key way: it automates the release of unused memory. Therefore, with garbage collection, dynamic allocation is a one-step operation. For example, in Java and C#, memory is allocated for use by **new**, but it is never explicitly freed by your program. Instead, the garbage collector runs periodically, looking for pieces of memory to which no other object points. When no other object points to a piece of dynamic memory, it means that there is no program element using that memory. When it finds a piece of unused memory, it frees it. Thus, in a garbage collection system, there is no **delete** operator, nor a need for one, either.

At first glance, the inherent simplicity of garbage collection makes it seem like the obvious choice for managing dynamic memory. In fact, one might question why the manual method is used at all, especially by a language as sophisticated as C++. However, in the case of dynamic allocation, first impressions prove deceptive because both approaches involve a set of trade-offs. Which approach is most appropriate is decided by the application. The following sections describe some of the issues involved.

The Pros and Cons of Manual Memory Management

The main benefit of manually managing dynamic memory is efficiency. Because there is no garbage collector, no time is spent keeping track of active objects or periodically looking for unused memory. Instead, when the programmer knows that the allocated object is no longer needed, the programmer explicitly frees it and no additional overhead is incurred. Because it has none of the overhead associated with garbage collection, the manual approach enables more efficient code to be written This is one reason why it was necessary for C++ to support manual memory management: it enabled the creation of high-performance code.

Another advantage to the manual approach is control. Although requiring the programmer to handle both the allocation and release of memory is a burden, the benefit is that the programmer gains complete control over both halves of the process. You know precisely when memory is being allocated and precisely when it is being released. Furthermore, when you release an object via **delete**, its destructor is executed at that point rather than at some later time, as can be the case with garbage collection. Thus, with the manual method you can control precisely when an allocated object is destroyed.

Although it is efficient, manual memory management is susceptible to a rather annoying type of error: the memory leak. Because memory must be freed manually, it is possible (even easy) to forget to do so. Failing to release unused memory means that the memory will remain allocated even if it is no longer needed. Memory leaks cannot occur in a garbage collection

environment because the garbage collector ensures that unused objects are eventually freed. Memory leaks are a particularly troublesome problem in Windows programming, where the failure to release unused resources slowly degrades performance.

Other problems that can occur with C++'s manual approach include the premature releasing of memory that is still in use, and the accidental freeing of the same memory twice. Both of these errors can lead to serious trouble. Unfortunately, they may not show any immediate symptoms, making them hard to find.

The Pros and Cons of Garbage Collection

There are several different ways to implement garbage collection, each offering different performance characteristics. However, all garbage collection systems share a set of common attributes that can be compared against the manual approach. The main advantages to garbage collection are simplicity and safety. In a garbage collection environment, you explicitly allocate memory via **new**, but you never explicitly free it. Instead, unused memory is automatically recycled. Thus, it is not possible to forget to release an object or to release an object prematurely. This simplifies programming and prevents an entire class of problems. Furthermore, it is not possible to accidentally free dynamically allocated memory twice. Thus, garbage collection provides an easy-to-use, error-free, reliable solution to the memory management problem.

Unfortunately, the simplicity and safety of garbage collection come at a price. The first cost is the overhead incurred by the garbage collection mechanism. All garbage collection schemes consume some CPU cycles because the reclamation of unused memory is not a cost-free process. This overhead does not occur with the manual approach.

A second cost is loss of control over when an object is destroyed. Unlike the manual approach, in which an object is destroyed (and its destructor called) at a known point in time—when a **delete** statement is executed on that object—garbage collection does not have such a hard and fast rule. Instead, when garbage collection is used, an object is not destroyed until the collector runs and recycles the object, which may not occur until some arbitrary time in the future. For example, the collector might not run until the amount of free memory drops below a certain point. Furthermore, it is not always possible to know the order in which objects will be destroyed by the garbage collector. In some cases, the inability to know precisely when an object is destroyed can cause trouble because it also means that your program can't know precisely when the destructor for a dynamically allocated object is called.

For garbage collection systems that run as a background task, this loss of control can escalate into a potentially more serious problem for some types of applications because it introduces what is essentially nondeterministic behavior into a program. A garbage collector that executes in the background reclaims unused memory at times that are, for all practical purposes, unknowable. For example, the collector will usually run only when free CPU time is available. Because this might vary from one program run to the next, from one computer to next, or from one operating system to the next, the precise point in program execution at which the garbage collector executes is effectively nondeterministic. This is not a problem for many programs, but it can cause havoc with real-time applications in which the unexpected allocation of CPU cycles to the garbage collector could cause an event to be missed.

You Can Have It Both Ways

As the preceding discussions explained, both manual management and garbage collection maximize one feature at the expense of another. The manual approach maximizes efficiency and control at the expense of safety and ease of use. Garbage collection maximizes simplicity and safety but pays for it with a loss of runtime performance and control. Thus, garbage collection and manual memory management are essentially opposites, each maximizing the traits that the other sacrifices. This is why neither approach to dynamic memory management can be optimal for all programming situations.

Although opposites, the two approaches are not mutually exclusive. They can coexist. Thus, it is possible for the C++ programmer to have access to both approaches, choosing the proper method for the task at hand. All one needs to do is create a garbage collector for C++, and this is the subject of the rest of this chapter.

Creating a Garbage Collector in C++

Because C++ is a rich and powerful language, there are many different ways to implement a garbage collector. One obvious, but limited, approach is to create a garbage collector base class, which is then inherited by classes that want to use garbage collection. This would enable you to implement garbage collection on a class-by-class basis. This solution is, unfortunately, too narrow to be satisfying.

A better solution is one in which the garbage collector can be used with any type of dynamically allocated object. To provide such a solution, the garbage collector must:

1. Coexist with the built-in, manual method provided by C++.
2. Not break any preexisting code. Moreover, it must have no impact whatsoever on existing code.
3. Work transparently so that allocations that use garbage collection are operated on in the same way as those that don't.
4. Allocate memory using **new** in the same way that C++'s built-in approach does.
5. Work with all data types, including the built-in types such as **int** and **double**.
6. Be simple to use.

In short, the garbage collection system must be able to dynamically allocate memory using a mechanism and syntax that closely resemble that already used by C++ and not affect existing code. At first thought, this might seem to be a daunting task, but it isn't.

Understanding the Problem

The key challenge that one faces when creating a garbage collector is how to know when a piece of memory is unused. To understand the problem, consider the following sequence:

```
int *p;
p = new int(99);
p = new int(100);
```

Here, two int objects are dynamically allocated. The first contains the value 99 and a pointer to this value is stored in p. Next, an integer containing the value 100 is allocated, and its address is also stored in p, thus overwriting the first address. At this point, the memory for int(99) is not pointed to by p (or any other object) and can be freed. The question is, how does the garbage collector know that neither p nor any other object points to int(99)?

Here is a slight variation on the problem:

```
int *p, *q;

p = new int(99);

q = p; // now, q points to same memory as p

p = new int(100);
```

In this case, q points to the memory that was originally allocated for p. Even though p is then pointed to a different piece of memory, the memory that it originally pointed to can't be freed because it is still in use by q. The question: how does the garbage collector know this fact? The precise way that these questions are answered is determined by the garbage collection algorithm employed.

Choosing a Garbage Collection Algorithm

Before implementing a garbage collector for C++, it is necessary to decide what garbage collection algorithm to use. The topic of garbage collection is a large one, having been the focus of serious academic study for many years. Because it presents an intriguing problem for which there is a variety of solutions, a number of different garbage collection algorithms have been designed. It is far beyond the scope of this book to examine each in detail. However, there are three archetypal approaches: *reference counting*, *mark and sweep*, and *copying*. Before choosing an approach, it will be useful to review these three algorithms.

Reference Counting

In reference counting, each dynamically allocated piece of memory has associated with it a reference count. This count is incremented each time a reference to the memory is added and decremented each time a reference to the memory is removed. In C++ terms, this means that each time a pointer is set to point to a piece of allocated memory, the reference count associated with that memory is incremented. When the pointer is set to point elsewhere, the reference count is decremented. When the reference count drops to zero, the memory is unused and can be released.

The main advantage of reference counting is simplicity—it is easy to understand and implement. Furthermore, it places no restrictions on the organization of the heap because the reference count is independent of an object's physical location. Reference counting adds overhead to each pointer operation, but the collection phase is relatively low-cost. The main