B.A. PART - III (SEMESTER-V) **PAPER : BAP-301**

COMPUTER APPLICATIONS **OBJECT ORIENTED PROGRAMMING USING C++**

UNIT-II

Department of Distance Education
Punjabi University, Patiala
(All Copyrights are Reserved)

LESSON NOs :

2.1 Classes and Objects
2.2 Member Access and Class Objects
2.3 Static Members and Friends
2.4 Constructor and Destructors
2.5 Dynamic Memory Allocation
2.6 Scopes
2.7 Inheritance
2.8 Pointers
2.9 Storage Classes

Note:- The students can download syllabus from departmental website **www.dccpbi.com**

LESSON NO. 2.1 **AUTHOR : KANWAL PREET SINGH**

CLASSES AND OBJECTS

2.1.1 Introduction

2.1.2 Objective

2.1.3 Creating New Types

2.1.4 Class

2.1.5 Creating Objects

2.1.6 Accessing Class Members

2.1.7 Defining Member Functions

2.1.8 The Complete Rectangle Program

2.1.9 Functions are Public, Data is Private

2.1.10 Inline Functions

2.1.11 Nesting Of Member Functions

2.1.12 An Example Program

2.1.13 Summary

2.1.14 Short Answer Type Questions

2.1.14 Long Answer Type Questions

2.1.15 Suggested Readings

2.1.1 Introduction

In C++, we can use various in build data types; as well we can create our own data types. A class is user defined data type. Once we define a class, we actually define a new data type and hence we can declare variables of that data type. These class variables are called objects. A class can contain both data and functions, that is, the data and functions are encapsulated inside a class. Data is prevented from open access and can be accessed only through member functions. This provides the concept of data hiding. We can use various access specifiers to give different access levels to the different members of the class.

2.1.2 Objective

After completing the lesson, you will have understanding of:

- Class
- Object
- Member variables and functions
- Inline Member Functions
- Member Access

2.1.3 Creating New Types

We have already learned about a number of variable types, including integers, floating point numbers and characters. The type of a variable tells you quite a bit about it. For example, if you declare Height and Width to be unsigned integers, you know that each one can hold a number between 0 and 65,535, assuming an integer is two bytes. I mean to say they are unsigned integers; trying to hold anything else in these variables causes an error. You can't store your name in an unsigned short integer. Just by declaring these variables to be unsigned short integers, you know that it is possible to add Height to Width and to assign that number to another number.

The type of these variables tells you:

- Their size in memory.
- What information they can hold.

- What actions can be performed on them.

For built-in types, this information is built into the compiler. But when you define a user-defined type in C++, you have to provide the same kind of information yourself. In exchange for this extra work, you gain the power and flexibility to custom fit new data types to match real-world requirements. More generally, a type is a category. Familiar types include car, house, person, fruit, and shape. In C++, the programmer can create any type needed, and each of these new types can have all the functionality and power of the built-in types.

The Need to Create a New Type?

Programs are usually written to solve real-world problems, such as keeping track of employee records or simulating the workings of a heating system. Although it is possible to solve complex problems by using programs written with only integers and characters, it is far easier to deal with large, complex problems if you can create representations of the objects that you are talking about. In other words, simulating the workings of a heating system is easier if you can create variables that represent rooms, heat sensors, thermostats, and boilers. The closer these variables correspond to reality, the easier it is to write the program.

2.1.4 Class

We make a new type by declaring a class**. A class is just a collection of variables, often of different types, combined with a set of related functions.** One way to think about a car is as a collection of wheels, doors, seats, windows, and so forth. Another way is to think about what a car can do: It can move, speed up, slow down, stop, park, and so on. A class enables you to encapsulate, or bundle, these various parts and various functions into one collection, which is called an object. Encapsulating everything you know about a car into one class has a number of advantages for a programmer. Everything is in one place, which makes it easy to refer to, copy, and manipulate the data. Likewise, clients of your class--that is, the parts of the program that use your class--can use your object without worry about what is in it or how it works. A class can consist of any combination of the variable types and also other class types.

The variables in the class are referred to as the member variables or data members. A Car class might have member variables representing the seats, radio type, tires, and so forth. Member variables are part of your class, just like the wheels and engine are part of your car.

The functions in the class typically manipulate the member variables. They are referred to as member functions or methods of the class. Methods of the Car class might include Start() and Brake().Member functions are as much a part of your class as the member variables. They determine what the objects of your class can do.

Declaring a Class

Classes are generally declared using the keyword class, with the following format:

```
class class_name
{
 access_specifier_1:
   member1;
 access_specifier_2:
   member2;
 ...
} object_names;
```

where class_name is a valid identifier for the class, object_names is an optional list of names for objects of this class. The body of the declaration can contain members, which can be either data or function declarations, or optionally access specifiers.

All is very similar to the declaration of structures, except that we can now include functions as members and there is also a new thing called access specifier. An access specifier is one of the following three keywords: **private, public or protected**. These specifiers modify the access rights that the members following them acquire:

- **private** members of a class are accessible only from within other members of the same class or from their *friends*.
- **protected** members are accessible from members of their same class and from their friends, but also from members of their derived classes.
- **public** members are accessible from anywhere where the object is visible.

By default, all members of a class have private access. Therefore, any member that is declared before one other class specifier automatically has private access.

A Simple Example

Consider the following example:

```
class Rectangle
 {
   int x, y;
  public:
   void set_values (int,int);
   int area ();
 };
```

Declares a class (i.e., a type) called Rectangle. This class contains four members: two data members of type int (member x and member y) with private access (because private is the default access specifier) and two member functions with public access: set_values() and area(), of which for now we have only included their declaration, not their definition.

2.1.5 Creating Objects

As already discussed, when we create a class, we are actually creating a new data type. So, once a class has been created, we can create variables of that type by using the class name(like any other built in type variable). For example the statement:

int m;

declares **m** to be a variable of type int. Similarly the statement:

Rectangle rect1;

declares **rect1** to be a variable of type **Rectangle**. In C++, the class variables are known as **objects.** In the above example, rect1 is an object of type Rectangle. The syntax for declaring an object in C++ is:

Class_name Object_name;

Just as a data_type can have any number of variables, a class can have any number of objects. In fact, we can create more than one object in a single statement. For example:

Rectangle rect1,rect2,rect3;

declares three objects – rect1, rect2 and rect3 of class Rectangle.

Note: A class specification (class definition), like a structure, provides only a template and does not create any memory space. The memory space is reserved when you create an object of that class. For example the statement:

Rectangle rect1;

reserves a space of four bytes for object - rect1(2 bytes each for class variables x and y). Similarly the statement:

Rectangle rect1,rect2,rect3;

reserves a space of four bytes each for objects - rect1, rect2 and rect3.

2.1.6 Accessing Class Members

After the previous declarations of Rectangle and rect, we can refer within the body of the program to any of the public members of the object rect as if they were normal functions or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member. It is very similar to what we did with plain structures before. The general syntax for accessing the class members is:

Object_name.member_name;

Where object_name is the name of the object of type class and member_name is any public member variable or member function of type class. For example:

```
rect.set_values (3,4);
int myarea = rect.area();
```

The only members of **rect** that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of the same class.

Assign to Objects, Not to Classes

In C++ you don't assign values to types, you assign values to variables. For example, you would never write

```
int = 5;          // wrong
```

The compiler would flag this as an error, because you can't assign 5 to an integer. Rather, you must define an integer variable and assign 5 to that variable. For example:

```
int  x;           // define x to be an int
x = 5;            // set x's value to 5
```

This is a shorthand way of saying, "Assign 5 to the variable x, which is of type int." In the same way, you wouldn't write

```
Rectangle.x=5;        // wrong
```

The compiler would flag this as an error, because you can't assign 5 to the x part of the Rectangle. Rather, you must define a Rectangle object and assign 5 to that object.

For example,

```
Rectangle rect1;        // just like  int x;
rect1.x = 5;      // just like  x = 5;
```

In this case, the compiler will generate an error message here also because x is declared as private and we can not access a private member outside its class. To access x directly, we will have to declare it as public.

2.1.7 Defining Member Functions

Member functions can be defined in two places:

Inside the class definition

We can define a member function inside the class of which it is the member. For example we can define the Rectangle class as:

```
class Rectangle
 {
   int x, y;
  public:
   void set_values (int a, int b)
     {
        x = a;
        y = b;
      }
   int area ()
      {
         return x*y;
       }
 };
```

When a function is defined inside a class, it is treated as an inline function. Therefore, all the restrictions that apply to inline functions are also applicable here. Normally, we define a function outside the class and only small functions are defined inside.

Outside the class definition

In this case, the functions are only declared inside the class, but are defined outside the class. Their definitions are very much like the normal functions. The general syntax for defining a function outside its class is:

```
return_type  class_name :: function_name(argument list)
{
        Function Body;
}
```

In this outside declaration, we must use the operator of scope (::) to specify that we are defining a function that is a member of the class and not a regular global function. The scope operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. The class Rectangle can be modified to define its functions outside in the following way:

```
class Rectangle
 {
   int x, y;
  public:
   void set_values (int,int);
   int area ();
 };
void Rectangle::set_values (int a, int b)
{
  x = a;
  y = b;
}
```

```
void Rectangle:: area ()
{
  return x*y;
}
```

The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior.

2.1.8 The Complete Rectangle Program

The complete Rectangle class program discussed so far is as follows:

```
#include <iostream.h>
class Rectangle
{
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};
void  Rectangle::set_values (int a, int b)
{
  x = a;
  y = b;
}
int main ()
```

```
{
  Rectangle rect1, rect2;
  rect1.set_values (3,4);
  rect2.set_values (5,6);
  cout << "rect1 area: " << rect1.area() << endl;
  cout << "rect2 area: " << rect2.area() << endl;
  return 0;
}
```

You may notice that the definition of the member function area() has been included directly within the definition of the Rectangle class given its extreme simplicity, whereas set_values() has only its prototype declared within the class, but its definition is outside. In this outside declaration, we must use the operator of scope (::) to specify that we are defining a function that is a member of the class Rectangle and not a regular global function.

The scope operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, in the function set_values() of the previous code, we have been able to use the variables x and y, which are private members of class Rectangle, which means they are only accessible from other members of their class.

Members x and y have private access (remember that if nothing else is said, all members of a class defined with keyword class have private access). By declaring them private we deny access to them from anywhere outside the class. This makes sense, since we have already defined a member function to set values for those members within the object: the member function set_values(). Therefore, the rest of the program does not need to have direct access to them.

In this example, the class which we are talking about is **Rectangle,** of which there are two instances or objects**: rect1** and **rect2**. Each one of them has its own member variables and member functions. Notice that the call to **rect1.area()** does not give the same result as the call to **rect2.area().** This is because each

object of class Rectangle has its own variables **x** and **y**, as they, in some way, have also their own function members **set_value()** and **area()** that each uses its object's own variables to operate.

That is the basic concept of *object-oriented programming*: Data and functions are both members of the object. We no longer use sets of global variables that we pass from one function to another as parameters, but instead we handle objects that have their own data and functions embedded as members. Notice that we have not had to give any parameters in any of the calls to rect1.area or rect2.area. Those member functions directly used the data members of their respective objects rect1 and rect2.

Note: In this lesson and the ones following it, we have used modern C++. Many C++ compilers still don't support it. In such compilers use the statement:

#include<stdio.h>

in place of

#include<iostream>

using namespace std;

2.1.9 Functions are Public, Data is Private

Usually the data within a class is private and the functions are public. This is a result of how classes are used. The data is hidden so it will be safe from accidental manipulation, while the functions that operate on the data are public so they can be accessed from outside the class. However, there is no rule that data must be private and functions public. In some circumstances you may find you'll need to use private functions and public data.

2.1.10 Inline Functions

The primary distinction between normal functions and inline functions is not in how you code them but in how the C++ compiler incorporates them into a program. The final product of the compilation process is an executable program, which consists of a set of machine language instructions. When you start a program, the operating system loads these instructions into the computer's

memory, so each instruction has a particular memory address. The computer then goes through these instructions step-by-step. Sometimes, as when you have a loop or a branching statement, program execution will skip over instructions, jumping back or forward to a particular address. Normal function calls also involve having a program jump to another address (the function's address) and then jump back when the function terminates.

Let's look at a typical implementation of that process in a little more detail. When the program reaches the function call instruction, the program stores the memory address of the instruction immediately following the function call, copies arguments to the stack (a block of memory reserved for that purpose), jumps to the memory location that marks the beginning of the function, executes the function code (perhaps placing a return value in a register), and then jumps back to the instruction whose address it saved. Jumping back and forth and keeping track of where to jump means that there is an overhead in elapsed time to using functions.

The C++ inline function provides an alternative. **This is a function whose compiled code is "in line" with the other code in the program. That is, the compiler replaces the function call with the corresponding function code.** With inline code, the program doesn't have to jump to another location to execute the code and then jump back. Inline functions thus run a little faster than regular functions, but there is a memory penalty. If a program calls an inline function ten times, then the program winds up with ten copies of the function inserted into the code.

You should be selective about using inline functions. The speed gain usually is minimal unless the function itself is so short that the time needed to execute the function is comparable to the time spent jumping to and from the function. In that case, the function already is fast, so about the only time you would get much of a benefit is if the function were the main time-consumer in a crucial loop.

To use this feature, you have to do two things:

- Preface the function definition with the keyword **inline**.
- Place the function definition above all functions that call it.

Note that you have to place the entire definition (meaning the function header and all the function code), not just the prototype, above the other functions.

The compiler does not have to honor your request to make a function inline. It might decide the function is too large or notice that it calls itself (recursion is not allowed for inline functions), or the feature might not be implemented for your particular compiler. Following example illustrates the inline technique with an inline **square()** function that squares its argument. Note that we've placed the entire definition on one line. That's not required, but if the definition doesn't fit on one line, the function probably is a poor candidate for an inline function.

```
#include <iostream>
 using namespace std;
 // an inline function must be defined before first use
 inline double square(double x) { return x * x; }
 int main()
 {
    double a, b;
    double c = 13.0;
    a = square(5.0);
    b = square(4.5 + 7.5);   // can pass expressions
    cout << "a = " << a << ", b = " << b << "\n";
    cout << "c = " << c;
    cout << ", c squared = " << square(c++) << "\n";
    cout << "Now c = " << c << "\n";
    return 0;
 }
```

Here's the output:

a = 25, b = 144

c = 13, c squared = 169

Now c = 14

Making Member Functions Inline

You may either define a member function inside its class definition, or you may define it outside if you have already declared (but not defined) the member function in the class definition. A member function that is defined inside its class member list is called an inline member function. Member functions containing a few lines of code are usually declared inline. If you define a member function outside of its class definition, it must appear in a namespace scope enclosing the class definition. You must also qualify the member function name using the scope resolution (::) operator.

An equivalent way to declare an inline member function is to either declare it in the class with the **inline** keyword (and define the function outside of its class) or to define it outside of the class declaration using the **inline** keyword.

In the following example, member function **Y::f()** is an inline member function:

```
class Y
 {
  private:
   char  a;
  public:
   char f() { return a; }
};
```

The following example is equivalent to the previous example, **Y::f()** is an inline member function:

```
class Y
 {
```

```
  private:
   char a;
  public:
   char f();
 };
inline char Y::f() { return a; }
```

2.1.11 Nesting Of Member Functions

Earlier we had discussed that a member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as the nesting of member functions. For example the **Rectangle** program can be modified to include another function show as:

```
#include <iostream.h>
class Rectangle
{
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
    void show(){cout<<"\n Area of rectangle is: "<<area();}
};
void  Rectangle::set_values (int a, int b)
{
  x = a;
```

```
  y = b;
}
int main ()
{
  Rectangle rect1, rect2;
  rect1.set_values (3,4);
  rect2.set_values (5,6);
  rect1.show();
  rect2.show();
  return 0;
}
```

In the above program we define another function show(). The function show() displays the area of the rectangle. The show() function does not calculate the area of the rectangle itself. Instead it just calls the function area(), which actually calculates the area of the rectangle. This is nesting of member functions.

2.1.12 Another Example Program

Write a Program to create a class TTime to display any date and time.

```
#include <iostream.h>
#include <stdio.h>
class TTime
{
private:
  int year;
  int month;
  int day;
```

```
  int hour;
  int minute;
public:
  void Display(void);
  void SetTime(int m, int d, int y, int hr, int min);
};
void TTime::Display(void)
{
  char s[32];
  sprintf(s, "Date: %02d/%02d/%04d  Time: %02d:%02d\n",
    month, day, year, hour, minute);
  cout << s;
}
void TTime::SetTime(int m, int d, int y, int hr, int min)
{
  month = m;     // Assign arguments to data members
  day = d;
  year = y;
  hour = hr;
  minute = min;
}
void main()
{
  TTime appointment;
```

```
  int month, day, year, hour, minute;
  appointment.SetTime(09, 24, 2008, 9, 15);
  cout << "Appointment == ";
  appointment.Display();
}
```

Output:

Appointment == Date: 09/24/2008 Time: 9:15

2.1.13 Summary

By declaring a class, we make a new type**.** A class is a collection of variables, often of different types, combined with a set of related functions. Once a class has been created, we can create variables of that type by using the class name. These variables are called objects. Inline functions are those which are preceded by the keyword **inline.** When a function is declared as inline, the compiler replaces the function call with the corresponding function code. A member function can be called by using its name inside another member function of the same class. This is known as the nesting of member functions.

2.1.14 Short Answer Type Questions

1. What is a user defined data type?
2. What is a class?
3. What is an object?

2.1.15 Long Answer Type Questions

1. Explain in detail the concept of classes and objects in C++.
2. What are the different ways of defining member functions? How can you make an outside function inline?
3. Explain member access with the help of an example.

2.1.16 Suggested Readings

1. Object Oriented Programming in C++ — Robert Lafore
2. The Complete Reference C++ — Herbert Schildt
3. Object Oriented Programming with C++ — E Balagurusamy

LESSON NO. 2.2 AUTHOR : KANWAL PREET SINGH

MEMBER ACCESS AND CLASS OBJECTS

2.2.1 Introduction

2.2.2 Objective

2.2.3 Member Accesss

2.2.4 Constructing Class Objects

2.2.5 Functions Returning Objects

2.2.6 Arrays of Objects

2.2.7 This Pointer

2.2.8 Summary

2.2.9 Short Answer Type Questions

2.2.10 Long Answer Type Questions

2.2.11 Suggested Readings

2.2.1 Introduction

In this lesson, we will study about Access Specifiers. C++ provides three access specifiers – public, private and protected, which can be attached with the class members. These specifiers determine that where the class members can be accessed. After that we discuss the different types of objects in C++. Like other

variables, objects can be passed as arguments to functions and functions can also return objects. We can also declare an array of objects. In the end, we discuss the this pointer, which is simply a pointer to the current object.

2.2.2 Objective

After reading this lesson you will be able to understand:

- Member Access
- Global and Local Objects
- Pointer to Objects
- Reference to Objects
- Passing object to functions
- Functions returning Objects
- Array of Objects
- this pointer

2.2.3 Member Access

Member access determines if a class member is accessible in an expression or declaration. Suppose x is a member of class A. Class member x can be declared to have one of the following levels of accessibility:

- **public:** x can be used anywhere without the access restrictions defined by private or protected.
- **private:** x can be used only by the members and friends of class A.
- **protected:** x can be used only by the members and friends of class A, and the members and friends of classes derived from class A.

Members of classes declared with the keyword class are private by default. Members of classes declared with the keyword struct or union are public by default.

To control the access of a class member, you use one of the access specifiers *public, private,* or *protected* as a label in a class member list. The following example demonstrates these access specifiers:

```
class A
 {
   friend class C;
   private:
    int a;
   public:
    int b;
   protected:
    int c;
};
class B : A
 {
  void f()
  {
    // a = 1;
     b = 2;
     c = 3;
  }
};
class C
 {
   void f(A x)
   {
     x.a = 4;
```

```
    x.b = 5;
    x.c = 6;
  }
};
int main() {
  A y;
//  y.a = 7;
    y.b = 8;
//  y.c = 9;
  B z;
//  z.a = 10;
    z.b = 11;
//  z.c = 12;
}
```

The following Table-1 lists the access of data members A::a, A::b, and A::c in various scopes of the above example.

Table-1

Scope	A::a	A::b	A::c
function B::f()	No access. Member A::a is private.	Access. Member A::b is public.	Access. Class B inherits from A.
function C::f()	Access. Class C is a friend of A.	Access. Member A::b is public.	Access. Class C is a friend of A.
object y in main()	No access. Member y.a is private.	Access. Member y.a is public.	No access. Member y.c is protected.

Scope	A::a	A::b	A::c
object z in main()	No access. Member z.a is private.	Access. Member z.a is public.	No access. Member z.c is protected.

An access specifier specifies the accessibility of members that follow it until the next access specifier or until the end of the class definition. You can use any number of access specifiers in any order. If you later define a class member within its class definition, its access specification must be the same as its declaration. The following example demonstrates this:

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

The compiler will not allow the definition of function **B** because this function has already been declared as private. A class member has the same access control regardless whether it has been defined within its class or outside its class. Access control applies to names. In particular, if you add access control to a typedef name, it affects only the typedef name. The following example demonstrates this:

```
class A
 {
   class B { };
  public:
   typedef B C;
};
```

```
int main()
 {
   A::C x;
// A::B y;
}
```

The compiler will allow the declaration **A::C x** because the typedef name **A::C** is public. The compiler would not allow the declaration **A::B y** because **A::B** is private.

Note: Accessibility and visibility are independent. Visibility is based on the scoping rules of C++. A class membe r can be visible and inaccessible at the same time.

2.2.4 Constructing Class Objects

Class objects can be global or local to a function. As you can with variables of built-in types such as int and double, you can refer to class objects using pointers and references. Think of classes as new data types, and use them accordingly, just as you do variables of other types. The nature of an object—that is, whether it's global, local, addressed by a pointer, and so on—affects when that object's class constructors and destructor are called. When an object is created, C++ automatically calls that object's default constructor, or it calls another constructor that you specify in the object's declaration. When an object is destroyed, C++ calls that object's destructor if its class declares one.

Note: You will be able to understand this section fully only after reading the next chapter on Constructors and Destructors.

Global Class Objects

A global object is declared outside of all the functions. A global object's class constructor is called before function main begins to run. This action ensures that all global class objects are initialized before the program formally starts. Using the TTime class from previous chapter, the global declaration

```
TTime today;
```

creates a global object today of the class TTime. The scope of this object is the entire program, not any particular function. All the functions in the program can access this object.

Local Class Objects

Automatic class objects declared locally to a function are created when the function is called and destroyed when the function ends. Like variables of common types, automatic class object data members are stored on the stack. In the function

```
void anyFunction(void)
{
  TTime now;  // Initialized on each entry to function
  ...
}
```

the default TTime constructor is called to initialize the class object now to the current date and time each time a statement calls the function. When the function ends, C++ calls TTime's destructor for the now object, thus giving the object the chance to clean up after itself before it is permanently destroyed.

Pointers to Objects

A pointer may address a dynamic class object, which is typically allocated heap memory by new. You can declare a pointer pToday to a class object of type TTime:

```
TTime *pToday;
```

This might be a global declaration or it might be local to a function. Like all pointers, pToday must be initialized before use, usually by using new:

```
pToday = new TTime;
```

Alternatively, you can perform both steps in one easy statement:

```
TTime *pToday = new TTime;
```

Probably, this is the most common method for declaring and initializing a pointer to a class object. In this statement, operator new allocates memory for an

object of type TTime. The address of that object's first byte is assigned to pToday. In addition, C++ calls the object's default constructor.

To use pointers to class objects, dereference them as you do pointers to objects of other types. For example, this statement displays the date and time for the TTime class object addressed by pToday:

```
pToday–>Display();
```

Classes resemble structures, so when using pointers to class objects, you'll most often use the –> operator to refer to an addressed object's member. However, you can also use the * dereference operator with class object pointers. For example, the statement

```
(*pToday). Display();
```

Will also call the Display() function of the object assigned to pToday but the former statement is easier to write, clearer to read, and performs the identical service. C++ calls a dynamic class object's destructor when that object is deleted. The statement

```
delete pToday;
```

deletes the object addressed by pToday, returning that object's memory to the heap's available pool. Just before the object is destroyed, C++ calls the class destructor, which can delete any memory that the object happens to own.

Note: Do not confuse the life of a pointer with the life of an addressed object. A global pointer exists for the duration of the program, but it may address a multitude of class objects created by new and destroyed by delete. Similarly, an automatic pointer declared locally to a function is created when the function runs and destroyed when the function ends. However, the objects addressed by pointers have global scope and must be deleted explicitly. In a function f, if you call new to create a dynamic object, that object remains in memory even after the function ends. You must explicitly delete any such object if you don't want it to remain in memory after the local pointer is destroyed.

Reference Objects

You may declare references to class objects. Here's a sample. First declare a global TTime class object such as

```
TTime today;  // Global today object
```

Later in the program, you can declare a reference to today like this:

```
TTime &rToday = today;  // Reference to today
```

The reference rToday is an alias for today, and may be used in today's stead. For example, each of these two statements displays today's date and time:

```
rToday.Display();
today.Display();
```

Because references refer to an existing object, the class constructor is not called when the reference is initialized, nor is the destructor called when the reference is destroyed. These actions occur only when the object itself is created and destroyed. References are more commonly declared as function parameters and return values. In practice, they are rarely used as described in this section, but it's helpful to know the syntax as explained here.

Parameter Objects (or Passing Objects to Functions)

You may pass class objects, pointers to class objects, and class object references as arguments to functions. Classes are data types, and you can declare function parameters of them as you do other parameters. Here's a sample:

```
void anyFunction(TTime t)
{
  t.Display();
}
```

Function anyFunction has a single parameter t of type TTime. The function calls t's Display member function to display the parameter's date and time. A program can define a class object of type TTime:

```
TTime today;
```

Then it can pass that object by value to anyFunction:

```
anyFunction(today);
```

Passing large class objects by value to functions causes those objects to occupy an undesirable amount of stack space. In such cases, a pointer parameter is more efficient:

```
void anyFunction(TTime *tp)
{
  tp–>Display();
}
```

To call this version of anyFunction, a statement passes the address of a TTime class object rather than the object itself:

```
anyFunction(&today);
```

This is one of the most common methods for passing objects to functions. However, parameters may also be declared as references. The results are similar to what you can achieve with pointer parameters, but inside the function, no pointer dereferences are required. Here's yet one more version of anyFunction, this time using a reference parameter to an object of class TTime:

```
void anyFunction(TTime &tr)  // tr is a reference parameter
{
  tr.Display();  // Call Display for object that tr references
}
```

Elsewhere in the program, you can pass a TTime class object directly to anyFunction's reference parameter:

```
anyFunction(today);
```

One disadvantage of reference parameters is that statements such as this appear to pass an argument by value. The fact that a reference to today is passed to anyFunction is not clear from the text.

2.2.5 Functions Returning Objects

Functions may return a class object directly, as a pointer, or as a reference. Least common is a function that returns a TTime class object directly:

```
TTime newTime(void)
{
  TTime t;
  return t;
}
```

Local object t (initialized automatically by a constructor call when the function begins) is returned directly as the function result. A statement could call newTime like this:

```
TTime anotherTime = newTime();
```

which copies newTime's function result to anotherTime. This technique offers few advantages, and it's just as easy to create a new TTime object without calling a function:

```
TTime anotherTime;  // Same as above
```

Furthermore, object function results are passed on the stack, which can slow the program and waste precious stack space for large objects or deeply nested function calls. More commonly, functions return pointers and references to class objects. For instance, here's a function that allocates a new TTime class object in memory and returns the object's address:

```
TTime *newTimeP(void)
{
  TTime *p = new TTime;  // Allocate and initialize object
  return p;  // Return address of object as function result
}
```

Inside the function, a TTime pointer p is assigned the address of a TTime class object allocated by new. The pointer exists only inside the function, but the object addressed by the pointer has global scope; therefore, the function may return its address. A statement can call newTime to obtain a new object and assign the object's address to a TTime pointer such as tp:

```
TTime *tp = newTimeP();
```

Finally, functions may return references to class objects. You might use a reference function to refer to an existing object such as a global TTime today:

```
TTime &newTimeR(void)
{
  return today;
}
```

Elsewhere, you can declare a TTime reference and assign newTimeR's result to it:

```
TTime &tr = newTimeR();
```

Reference tr is now an alias for today, and can be used in a statement:

```
tr.Display();
```

The reference function also can be used directly in a statement. For example, the program can call newTimeR to display today's date and time:

```
newTimeR().Display();
```

This technique is especially useful when a function such as newTimeR performs a search operation, perhaps returning a reference to one of several TTime class objects based on some specified criteria.

Each time you pass an object into a function by value, a copy of the object is made. Each time you return an object from a function by value, another copy is made. Doing so takes time and memory. For small objects, such as the built-in integer values, this is a trivial cost. However, with larger, user-created objects, the cost is greater. The size of a user-created object on the stack is the sum of each of its member variables. These, in turn, can each be user-created objects, and passing such a massive structure by copying it onto the stack can be very expensive in performance and memory consumption. There is another cost as well. With the classes you create, each of these temporary copies is created when the compiler calls a special constructor: the copy constructor. You will learn how copy constructors work and how you can make your own in the chapter containing Constructors, but for now it is enough to know that the copy

constructor is called each time a temporary copy of the object is put on the stack. When the temporary object is destroyed, which happens when the function returns, the object's destructor is called. If an object is returned by the function by value, a copy of that object must be made and destroyed as well. With large objects, these constructor and destructor calls can be expensive in speed and use of memory.

To illustrate this idea, the following program creates a stripped-down user-created object: Car. A real object would be larger and more expensive, but this is sufficient to show how often the copy constructor and destructor are called. The program creates the Car object and then calls two functions. The first function receives the Car by value and then returns it by value. The second one receives a pointer to the object, rather than the object itself, and returns a pointer to the object.

```
1:  //Example 1
2:  // Passing pointers to objects
3:  #include <iostream>
4:  using namespace std;
5:
6:  class Car
7:  {
8:  public:
9:          Car ();               // constructor
10:         Car(Car&);     // copy constructor
11:         ~Car();               // destructor
12:  };
13:
14:  Car::Car()
15:  {
```

```
16:         cout << "Car Constructor...\n";
17:  }
18:
19:  Car::Car(Car&)
20:  {
21:         cout << "Car Copy Constructor...\n";
22:  }
23:
24:  Car::~Car()
25:  {
26:         cout << "Car Destructor...\n";
27:  }
28:
29:  Car FunctionOne (Car theCar);
30:  Car* FunctionTwo (Car *theCar);
31:
32:  int main()
33:  {
34:         cout << "Making a Car...\n";
35:         Car Maruti;
36:         cout << "Calling FunctionOne...\n";
37:         FunctionOne(Maruti);
38:         cout << "Calling FunctionTwo...\n";
39:         FunctionTwo(&Maruti);
```

```
40:   return 0;
41: }
42:
43: // FunctionOne, passes by value
44: Car FunctionOne(Car theCar)
45: {
46:          cout << "Function One. Returning...\n";
47:          return theCar;
48: }
49:
50: // functionTwo, passes by reference
51: Car* FunctionTwo (Car  *theCar)
52: {
53:          cout << "Function Two. Returning...\n";
54:          return theCar;
55: }
```

Output:

```
1: Making a Car...
2: Car Constructor...
3: Calling FunctionOne...
4: Car Copy Constructor...
5: Function One. Returning...
6: Car Copy Constructor...
```

7: Car Destructor...

8: Car Destructor...

9: Calling FunctionTwo...

10: Function Two. Returning...

11: Car Destructor...

Explanation: A very simplified Car class is declared on lines 6-12. The constructor, copy constructor, and destructor all print an informative message so that you can tell when they've been called. On line 34, main() prints out a message, and that is seen on output line 1. On line 35, a Car object is instantiated. This causes the constructor to be called, and the output from the constructor is seen on output line 2. On line 36, main() reports that it is calling FunctionOne, which creates output line 3. Because FunctionOne() is called passing the Car object by value, a copy of the Car object is made on the stack as an object local to the called function. This causes the copy constructor to be called, which creates output line 4. Program execution jumps to line 46 in the called function, which prints an informative message, output line 5. The function then returns, and returns the Car object by value. This creates yet another copy of the object, calling the copy constructor and producing line 6. The return value from FunctionOne() is not assigned to any object, and so the temporary object created for the return is thrown away, calling the destructor, which produces output line 7. Since FunctionOne() has ended, its local copy goes out of scope and is destroyed, calling the destructor and producing line 8. Program execution returns to main(), and FunctionTwo() is called, but the parameter is passed by reference. No copy is produced, so there's no output. FunctionTwo() prints the message that appears as output line 10 and then returns the Car object, again by reference, and so again produces no calls to the constructor or destructor. Finally, the program ends and Maruti goes out of scope, causing one final call to the destructor and printing output line 11. **The net effect of this is that the call to FunctionOne(), because it passed the Car by value, produced two calls to the copy constructor and two to the destructor, while the call to FunctionTwo() produced none.**

2.2.6 Arrays of Objects

Any object, whether built-in or user-defined, can be stored in an array. When you declare the array, you tell the compiler the type of object to store and the number of objects for which to allocate room. The compiler knows how much room is needed for each object based on the class declaration.

Accessing member data in an array of objects is a two-step process. You identify the member of the array by using the index operator ([ ]), and then you add the member operator (.) to access the particular member variable. The following example demonstrates how you would create an array of five CARs.

```
// Example 2
// An array of objects
#include <iostream>
using namespace std;
class CAT
{
   private:
     int itsAge;
     int itsWeight;
   public:
     CAT() { itsAge = 1; itsWeight=5; }
     int GetAge() const { return itsAge; }
     int GetWeight() const { return itsWeight; }
     void SetAge(int age) { itsAge = age; }
};
 int main()
   {
```

```
    CAT C[5];
    int i;
    for (i = 0; i < 5; i++)
        C[i].SetAge(2*i +1);
    for (i = 0; i < 5; i++)
    {
        cout << "CAT #" << i+1<< ": ";
        cout << C[i].GetAge() << endl;
    }
  return 0;
}
```

Output:

CAT #1: 1

CAT #2: 3

CAT #3: 5

CAT #4: 7

CAT #5: 9

Explanation: The CAT class must have a default constructor so that CAT objects can be created in an array. Remember that if you create any other constructor, the compiler-supplied default constructor is not created; you must create your own. The first for loop sets the age of each of the five CATs in the array. The second for loop accesses each member of the array and calls GetAge(). Each individual CAT's GetAge() method is called by accessing the member in the array, C[i], followed by the dot operator (.), and the member function.

2.2.7 This Pointer

When a call to non-static member function is made, the this pointer is created and set to the address of the object for which it is called, and then passed as first implicit argument. The this pointer can be treated like any other pointer to an object and thus can be used to access the data in the object it points to. As a specific case, if the name of the arguments of the member function is same as that of data members, then the arguments take precedence over data members being local to the member function. In this case, to refer to data members you have to use this pointer.

```
// Program to demonstrate use of this pointer
#include<iostream>
using namespace std;
class UseThis
 {
   int a,b;
   public:
   void setData(int a, int b)
    {
     a=a;
     b=b;
    }
  void showData()
   {
    cout<<"\n a = "<<a;
    cout<<"\n b = "<<b;
   }
```

```
};
void main()
 {
   UseThis u1;
   u1.showData();
   u1.setData(21,11);
   u1.showData();
 }
```

In the above program, in both the calls to showData(), the output will be same because in setData() member function, the values of the arguments get assigned to themselves, being local and the data members are left unchanged. In order to assign the values of local variables – a,b to the object variables - a,b we should make use of this pointer in setData() member function. The setData() should be modified as:

```
void setData(int a, int b)
   {
    this.a=a;
    this.b=b;
   }
```

Now in the above setData() member function, simple a,b refer to local variables a and b whereas this.a and this.b refer to object variables a and b. Therefore the statement:

```
this.a = a;
```

assigns value of local variable a to object variable a that is in the above example it will assign value of 21 to variable a of object u1.

2.2.8 Summary

Member access determines if a class member is accessible in an expression or declaration. To control the access of a class member, you use one of the access specifiers *public, private,* or *protected* as a label in a class member list. Class objects can be global or local to a function. A global object is declared outside of all the functions. A global class object's class constructor is called before function main begins to run. The scope of this object is the entire program, not any particular function. Automatic class objects declared locally to a function are created when the function is called and destroyed when the function ends. We can assign an object created by new operator to an object pointer. You may declare references to class objects. The reference is simply an alias. We may pass class objects, pointers to class objects, and class object references as arguments to functions. Functions may return a class object directly, as a pointer, or as a reference. Any object, whether built-in or user-defined, can be stored in an array. When you declare the array, you tell the compiler the type of object to store and the number of objects for which to allocate room. The compiler knows how much room is needed for each object based on the class declaration. The this pointer can be treated like any other pointer to an object and thus can be used to access the data in the object it points to.

2.2.9 Short Answer Type Questions

1. Explain Global and Local objects.
2. What is an object reference?
3. Is it possible to create an array of objects? Explain.
4. What is an object pointer?

2.2.10 Long Answer Type Questions

1. Explain in detail the different types of objects that can be created in C++.
2. Can you pass objects as arguments to functions and can the functions return objects? Explain with example.
3. Explain this pointer with the help of an example.

2.2.11 Suggested Readings

4. Object Oriented Programming in C++ Robert Lafore
5. The Complete Reference C++ Herbert Schildt
6. Object Oriented Programming with C++ E Balagurusamy

LESSON NO. 2.3 AUTHOR : KANWAL PREET SINGH

STATIC MEMBERS AND FRIENDS

2.3.1 Introduction

2.3.2 Objective

2.3.3 Static members

2.3.4 const Members

2.3.5 Friend Function

2.3.6 friend Classes

2.3.7 Summary

2.3.8 Short Answer Type Questions

2.3.9 Long Answer Type Questions

2.3.10 Suggested Readings

2.3.1 Introduction

Until now, we have dealt with object variables. Each object has its own copy of object variables. In this lesson we will introduce the concept of static variables – the variables which are shared by all the objects of the class and which are initialized before any object is created. These variables can be accessed without

using the object name. The chapter also introduces const members, that is, const variables, methods and objects. In the previous lesson we have done that the private members of the class can be accessed by the member functions of that class only but in this chapter, we will learn how outside functions and classes can access the private members of the class by becoming its friend.

2.3.2 Objective

After reading this lesson you will be able to understand the concept of:

- Static variables
- Static member functions
- const members
- friend functions
- friend classes

2.3.3 Static members

Class members can be declared using the storage class specifier **static** in the class member list. Only one copy of the static member is shared by all objects of a class in a program. When you declare an object of a class having a static member, the static member is not part of the class object.

A typical use of static members is for recording data common to all objects of a class. For example, you can use a static data member as a counter to store the number of objects of a particular class type that are created. Each time a new object is created, this static data member can be incremented to keep track of the total number of objects. This is shown on the following example:

```
// Example
// static class data
#include <iostream>
using namespace std;
class Stat_Demo
  {
  private:
```

```
    static int count;   //only one data item for all objects
                        //note: *declaration* only!
  public:
    Stat_Demo()              //increments count when object created
      { count++; }
    int getcount()     //returns count
    { return count; }
  };
int Stat_Demo::count = 0;       //*definition* of count
int main()
  {
  Stat_Demo f1, f2, f3;        //create three objects
  cout << "count is " << f1.getcount() << endl;  //each object
  cout << "count is " << f2.getcount() << endl;  //sees the
  cout << "count is " << f3.getcount() << endl;  //same value
  return 0;
  }
```

Explanation: The class **Stat_Demo** in this example has one data item, **count,** which is type **static int.** The constructor for this class causes **count** to be incremented. In main() we define three objects of class **Stat_Demo.** Since the constructor is called three times, **count** is incremented three times. Another member function, **getcount(),** returns the value in count. We call this function from all three objects, and as we expected, each prints the same value. Here's the output:

count is 3

count is 3

count is 3

If we had used an ordinary automatic variable—as opposed to a static variable—for count, each constructor would have incremented its own private copy of count once, and the output would have been

count is 1

count is 1

count is 1

The following Figure-1 shows how static variables compare with automatic variables.

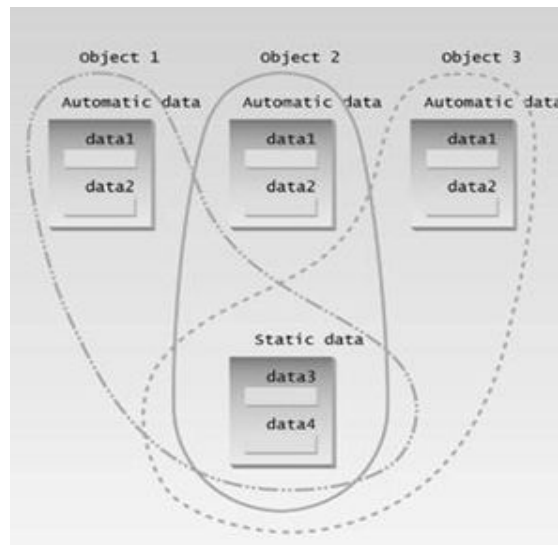


Figure-1

Static data members

The declaration of a static data member in the member list of a class is not a definition. You must define the static member outside of the class declaration, in namespace scope. For example:

class X

{

```
 public:
    static int i;
};
int X::i = 0; // definition outside class declaration
```

Once you define a static data member, it exists even though no objects of the static data member's class exist. In the above example, no objects of class X exist even though the static data member X::i has been defined.

Static data members of a class in namespace scope have external linkage. The initializer for a static data member is in the scope of the class declaring the member. A static data member can be of any type except for **void** or **void** qualified with **const** or **volatile**. You cannot declare a static data member as **mutable.** You can only have one definition of a static member in a program. Unnamed classes, classes contained within unnamed classes, and local classes cannot have static data members. Static data members and their initializers can access other static private and protected members of their class. The following example shows how you can initialize static members using other static members, even though these members are private:

```
class C
 {
    static int i;
    static int j;
    static int k;
    static int l;
    static int r;
    static int s;
    static int f() { return 0; }
    int a;
```

```
  public:
    C() { a = 0; }
 };
C c;
int C::i = C::f();    // initialize with static member function
int C::j = C::i;      // initialize with another static data member
int C::k = c.f();     // initialize with member function from an object
int C::l = c.j;       // initialize with data member from an object
int C::s = c.a;       // initialize with nonstatic data member
int C::r = 1;         // initialize with a constant value
```

If a static data member is of **const** integral or **const** enumeration type, you may specify a constant initializer in the static data member's declaration. This constant initializer must be an integral constant expression. Note that the constant initializer is not a definition. You still need to define the static member in an enclosing namespace. The following example demonstrates this:

```
#include <iostream>
using namespace std;
class X
 {
  static const int a = 76;
 };
 const int X::a;
 int main()
 {
  cout << X::a << endl;
```

```
}
```

The tokens **= 76** at the end of the declaration of static data member **a** is a constant initializer.

Static member functions

You cannot have static and nonstatic member functions with the same names and the same number and type of arguments. Like static data members, you may access a static member function **f()** of a class **A** without using an object of class **A.** A static member function does not have a **this** pointer. The following example demonstrates this:

```
#include <iostream>
using namespace std;
class X
 {
  private:
    int i;
    static int si;
  public:
    void set_i(int arg) { i = arg; }
    static void set_si(int arg) { si = arg; }
    void print_i()
     {
       cout << "Value of i = " << i << endl;
       cout << "Again, value of i = " << this->i << endl;
     }
    static void print_si()
```

```
    {
      cout << "Value of si = " << si << endl;
//    cout << "Again, value of si = " << this->si << endl;
    }
  };
  int X::si = 77;       // Initialize static data member
 int main()
 {
   X xobj;
   xobj.set_i(11);
   xobj.print_i();
   // static data members and functions belong to the class and
   // can be accessed without using an object of class X
   X::print_si();
   X::set_si(22);
   X::print_si();
 }
```

Output:

Value of i = 11

Again, value of i = 11

Value of si = 77

Value of si = 22

Explanation: The compiler does not allow the member access operation **this->si** in function **A::print_si()** because this member function has been declared as static, and therefore does not have a **this** pointer.

A static member function cannot be declared with the keywords **virtual, const, volatile or const volatile.** A static member function can access only the names of static members, enumerators, and nested types of the class in which it is declared. Suppose a static member function **f()** is a member of class X. The static member function **f()** cannot access the nonstatic members X or the nonstatic members of a base class of X.

2.3.4 const Members

With the **const** prefix you can declare constants with a specific type in the same way as you would do with a variable:

```
const int pathwidth = 100;
const char tabulator = '\t';
```

Here, **pathwidth** and **tabulator** are two typed constants. They are treated just like regular variables except that their values cannot be modified after their definition.

const Member Functions

A **const** member function guarantees that it will never modify any of its class's member data. The following program shows how this works.

```
//demonstrates const member functions
class aClass
  {
    private:
      int alpha;
    public:
      void nonFunc()        //non–const member function
```

```
      { alpha = 81; }        //OK
      void conFunc() const  //const member function
      { alpha = 81; }        //ERROR: can't modify a member
   };
```

The non–const function **nonFunc()** can modify member data alpha, but the constant function **conFunc()** can't. If it tries to, a compiler error results. A function is made into a constant function by placing the keyword **const** after the declarator but before the function body. If there is a separate function declaration, **const** must be used in both declaration and definition. Member functions that do nothing but acquire data from an object are obvious candidates for being made **const**, because they don't need to modify any data.

Making a function **const** helps the compiler flag errors, and tells anyone looking at the listing that you intended the function not to modify anything in its object. It also makes possible the creation and use of const objects, which we'll discuss soon.

To avoid raising too many subjects at once we have, up to now, avoided using **const** member functions in the example programs. However, there are many places where **const** member functions should be used. For example, in the **Distance** class, shown in the following program, the showdist() member function could be made const because it doesn't (or certainly shouldn't!) modify any of the data in the object for which it is called. It should simply display the data.

```
// const member functions and const arguments to member functions
#include <iostream>
using namespace std;
class Distance
  {
   private:
      int feet;
```

```
    float inches;
  public:
    Distance() { feet=0, inches=0.0;}                    //constructor
(no args)
    Distance(int ft, float in) { feet=ft, inches=in;}     //constructor
(two args)
     void getdist()              //get length from user
    {
    cout << "\nEnter feet: ";
    cin >> feet;
    cout << "Enter inches: ";
    cin >> inches;
    }
    void showdist() const          //display distance
    { cout << feet << "\'–" << inches << '\"'; }
 };
int main()
  {
   Distance dist1, dist2;         //define two lengths
   Distance dist2(11, 6.25);      //define, initialize dist2

   dist1.getdist();               //get dist1 from user
                  //display all lengths
   cout << "\ndist1 = ";  dist1.showdist();
```

```
   cout << "\ndist2 = ";  dist2.showdist();
   cout << endl;
   return 0;
  }
```

Note: If an argument is passed to an ordinary function by reference, and you don't want the function to modify it, then the argument should be made **const** in the function declaration (and definition). This is true of member functions as well.

const Objects

In several example programs we've seen that we can apply **const** to variables of basic types like **int** to keep them from being modified. In a similar way we can apply const to objects of classes. When an object is declared as **const**, you can't modify it. It follows that you can use only **const** member functions with it, because they're the only ones that guarantee not to modify it. The following program shows an example.

```
// constant Distance objects
#include  <iostream>
using namespace std;
class Distance
 {
  private:
     int feet;
     float inches;
  public:
     Distance() { feet=0, inches=0.0;}                          //constructor
(no args)
```

```
    Distance(int ft, float in) { feet=ft, inches=in;}        //constructor
(two args)
     void getdist()                  //get length from user
    {
    cout << "\nEnter feet: ";
    cin >> feet;
    cout << "Enter inches: ";
    cin >> inches;
    }
    void showdist() const           //display distance
    { cout << feet << "\'-" << inches << '\"'; }
 };
int main()
  {
   const Distance Stat_Demotball(300, 0);
   // Stat_Demotball.getdist();            //ERROR: getdist() not const
   cout << "Stat_Demotball = ";
   Stat_Demotball.showdist();           //OK
   cout << endl;
   return 0;
  }
```

A Stat_Demotball field (for American–style Stat_Demotball) is exactly 300 feet long. If we were to use the length of a Stat_Demotball field in a program, it would make sense to make it **const**. The CONSTOBJ program makes Stat_Demotball a const variable. Now only const functions, such as showdist(), can be called for this

object. Non–const functions, such as getdist(), which gives the object a new value obtained from the user, are illegal. In this way the compiler enforces the const value of Stat_Demotball.

When you're designing classes it's a good idea to make const any function that does not modify any of the data in its object. This allows the user of the class to create const objects. These objects can use any const function, but cannot use any non–const function. Remember, using const helps the compiler to help you.

2.3.5 Friend Function

Imagine that you want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments, and operate on their private data. In this situation we need a **friend** function. Here's a simple example that shows how **friend** functions can act as a bridge between two classes:

```
// friend functions
#include <iostream>
using namespace std;
class beta;            //needed for frifunc declaration
class alpha
  {
    private:
     int data;
    public:
     alpha() { data = 3; }          //no-arg constructor
     friend int frifunc(alpha, beta);  //friend function
  };
class beta
  {
```

```
    private:
     int data;
    public:
     beta() { data = 7;  }              //no-arg constructor
     friend int frifunc(alpha, beta);  //friend function
  };
int frifunc(alpha a, beta b)            //function definition
  {
    return( a.data + b.data );
  }
int main()
  {
    alpha aa;
    beta bb;
   cout << frifunc(aa, bb) << endl;     //call the function
   return 0;
  }
```

In this program, the two classes are **alpha** and **beta.** The constructors in these classes initialize their single data items to fixed values (3 in alpha and 7 in beta). We want the function **frifunc()** to have access to both these private data members, so we make it a **friend** function. It's declared with the **friend** keyword in both classes:

friend int frifunc(alpha, beta);

This declaration can be placed anywhere in the class, it doesn't matter if it goes in the **public** or the **private** section. An object of each class is passed as an argument to the function **frifunc(),** and it accesses the private data member of

both classes through these arguments. The function doesn't do much: It adds the data items and returns the sum. The main() program calls this function and prints the result.

Remember that a class can't be referred to until it has been declared. Class **beta** is referred to in the declaration of the function **frifunc()** in class **alpha,** so **beta** must be declared before **alpha**. Hence the declaration

class beta;

at the beginning of the program.

We should note that friend functions are controversial. During the development of C++, arguments raged over the desirability of including this feature. On the one hand, it adds flexibility to the language while on the other, it is not in keeping with **data hiding,** the philosophy that only member functions can access a class's private data.

A friend function must be declared as such within the class whose data it will access. Thus a programmer who does not have access to the source code for the class cannot make a function into a friend. In this respect the integrity of the class is still protected. Even so, friend functions are conceptually messy. For this reason friend functions should be used sparingly. If you find yourself using many friends, you may need to rethink the design of the program.

2.3.6 friend Classes

If we want all the member functions of a class to be made friends with some other class at the same time, then we make the entire class a friend. The following program shows an example of friend class:

```
// friend classes
#include <iostream>
using namespace std;
class alpha
  {
   private:
```

```
    int data1;
   public:
    alpha() { data1=81} { }   //constructor
    friend class beta;        //beta is a friend class
  };
class beta
  {                              //all member functions can
   public:                       //access private alpha data
    void fun1(alpha a)  { cout << "\ndata1=" << a.data1; }
    void fun2(alpha a)  { cout << "\ndata1=" << a.data1; }
  };
int main()
  {
   alpha a;
   beta b;
   b.fun1(a);
   b.fun2(a);
   cout << endl;
   return 0;
}
```

In class alpha the entire class beta is proclaimed a friend. Now all the member functions of beta can access the private data of alpha (in this program the single data item data1). Note that in the friend declaration we specify that beta is a class using the class keyword:

friend class beta;

We could also have declared beta to be a class before the alpha class specifier, as in previous examples.

class beta;

and then, within alpha, referred to beta without the class keyword:

friend beta;

2.3.7 Summary

Class members can be declared using the storage class specifier **static** in the class member list. Only one copy of the static member is shared by all objects of a class in a program. When you declare an object of a class having a static member, the static member is not part of the class object. The this pointer can be treated like any other pointer to an object and thus can be used to access the data in the object it points to. A const member function guarantees that it will never modify any of its class's member data. When an object is declared as const, you can't modify it. When you declare a function as a friend to a particular class, then that function can access the private members of that class. Similarly if you want that all the members functions of the class should be able to access the private members of some particular class, then the first class should be declared as a friend to the second one.

2.3.8 Short Answer Type Questions

1. What are static data members?
2. Can an outside function access private members of a class? How?

2.3.9 Long Answer Type Questions

4. Explain with example how static variables are different from normal object variables.
5. Explain const variables, functions and objects.

2.3.10 Suggested Readings

2. Object Oriented Programming in C++ Robert Lafore
3. The Complete Reference C++ Herbert Schildt
4. Object Oriented Programming with C++ E Balagurusamy

LESSON NO. 2.4

AUTHOR : KANWAL PREET SINGH

CONSTRUCTORS AND DESTRUCTORS

2.4.1 Introduction

2.4.2 Objective

2.4.3 Constructors

2.4.4 Default Constructor

2.4.5 Parameterized Constructors

2.4.6 Overloading Constructors

2.4.7 Copying an Object

2.4.8 Using Copy Constructor

2.4.9 Destructors

2.4.10 Summary

2.4.11 Short Answer Type Questions

2.4.12 Long Answer Type Questions

2.4.13 Suggested Readings

2.4.1 Introduction

C++ provides special functions for initializing objects when they are created. These functions are called Constructors. In this lesson, we will discuss how to define constructors, properties of constructors and different types of constructors. The constructors have same name as that of the class and do not have any return type. Like other functions, we can define more than one constructor in the same class and can distinguish between them by number and type of arguments. Destructors are opposite of constructors. They are executed when the object is being deleted.

2.4.2 Objective

After reading the following lesson you must be able to understand:

- Constructors – Definition and Properties
- Different types of Constructors
- Constructor Overloading
- Copy Constructor
- Destructors – Definition and Properties

2.4.3 Constructors

A constructor is a member function with the same name as its class. It initializes the object of the class whenever that object is created. For example:

```
class X
{
  public:
  X();      // constructor for class X
};
```

In the above example X() is a constructor of class X. It is invoked itself whenever an object of class X is created.

The various properties of constructors are:

1. Constructors are used initialize objects of their class type.

2. The name of the constructor functions is same as the name of the class in which they are defined.
3. They should be declared in the public section of the class.
4. There is no need for calling a constructor explicitly. A constructor is invoked itself whenever an object of that particular class is declared.
5. You cannot declare a constructor as virtual or static.
6. You cannot declare a constructor as const, volatile, or const volatile.
7. Unlike some of the other methods, the constructor does not return a value, not even void.
8. They cannot be inherited, though the derived class can call the base class constructor.

The following program modifies the Rectangle program created in Chapter 1 of this section to include a constructor that initializes the values of variables x and y to 0.

```
#include <iostream>
using namespace std;
class Rectangle
{
    int x, y;
  public:
    Rectangle(){ x=0; y=0; }    // Constructor function
    void set_values (int,int);
    int area () {return (x*y);}
};
void  Rectangle::set_values (int a, int b)
{
  x = a;
  y = b;
```

```
}
int main ()
{
  Rectangle rect1, rect2;
  rect1.set_values (3,4);
  rect2.set_values (5,6);
  cout << "rect1 area: " << rect1.area() << endl;
  cout << "rect2 area: " << rect2.area() << endl;
  return 0;
}
```

In the above program, whenever you create an object of class Rectangle, the constructor function Rectangle() will be invoked itself and will initialize the values of variables x and y of that object to 0.

2.4.4 Default Constructor

The **default constructor** is the constructor used to create an object when you don't provide explicit initialization values. That is, it's the constructor used for declarations like this:

```
Rectangle rect1; // uses the default constructor
```

If you fail to provide any constructors, C++ automatically supplies a default constructor. It's a default version of a default constructor, and it does nothing. For the **Rectangle class,** it would look like this:

```
Rectangle::Rectangle() { }
```

The net result is that the **rect1** object is created with its members uninitialized, just as

```
int x;
```

creates **x** without providing it a value. The fact that the default constructor has no arguments reflects the fact that no values appear in the declaration.

But after you define any constructor for a class, the responsibility for providing a default constructor for that class passes from the compiler to you. A curious fact about the default constructor is that the compiler provides one only if you don't define any constructors. After you define any constructor for a class, the responsibility for providing a default constructor for that class passes from the compiler to you. If you provide a non-default constructor, such as **Rectangle(int h, int l)** and don't provide your own version of a default constructor, then a declaration like

```
Rectangle rect1; // not possible with current constructor
```

becomes an error. The reason for this behavior is that you might want to make it impossible to create uninitialized objects. On the other hand, you might want to be able to create objects without explicit initialization. If so, you must define your own default constructor. This is a constructor that takes no arguments. You can define a default constructor two ways. One is to provide default values for all the arguments to the existing constructor :

```
Rectangle(){x=0;y=0;}
```

The second is to use function overloading to define a second constructor, one that has no arguments:

```
Rectangle();
```

Actually, you usually should initialize objects in order to ensure that all members begin with known, reasonable values. Thus, the default constructor typically provides implicit initialization for all member values.

2.4.5 Parameterized Constructors

In the above example, we use a constructor to initialize the object variables. But for all the objects, the variables are initialized to same default value – 0. To assign actual values to the object variables, we have to call the function – set_values(). For example, in the above program, the code

```
Rectangle rect1, rect2;
```

simply creates two objects and assigns default values – 0 to the variables of these objects. To assign actual values, we had to call set_values() function as:

```
rect1.set_values (3,4);
rect2.set_values (5,6);
```

Now what would happen if in the previous example we called the member function area() before having called function set_values()? We would have got the result - 0, since the members x and y have been assigned a default value of 0. In order to avoid that, a class can include a parameterized constructor, in which we can assign the values to object variables during the object creation itself. **The constructors that can take arguments are called parameterized constructors.** When a constructor has been parameterized, the object declaration statement such as:

```
Rectangle rect1;
```

will not work. We must pass the initial values as arguments to the constructor function when an object is declared(or we should provide an explicit definition for the default constructor). This can be done in two ways:

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

The following declaration illustrates the first method:

```
Rectangle rect1 = Rectangle(3,4);
```

This statement creates a Rectangle object rect1 and passes values 3 and 4 to it. The second method is implemented as follows:

```
Rectangle rect1(3,4);
```

This method, also called the shorthand method is used often as it is shorter, looks better and is easy to implement. We can change the Rectangle program to include paramterized constructors as follows:

```
#include <iostream>
using namespace std;
class Rectangle
```

```
{
   int x, y;
  public:
   Rectangle(int a, int b)  // Parameterized Constructor function
    {
      x=a;
      y=b;
    }
   int area () {return (x*y);}
};
int main ()
{
  Rectangle rect1(3,4);
  Rectangle rect2(5,6);
  cout << "rect1 area: " << rect1.area() << endl;
  cout << "rect2 area: " << rect2.area() << endl;
  return 0;
}
```

As you can see, the result of this example is identical to the previous one. But now we have removed the member function set_values(), and have included instead a parameterized constructor that performs a similar action: it initializes the values of x and y with the parameters that are passed to it. Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

```
  Rectangle rect1(3,4);
  Rectangle rect2(5,6);
```

2.4.6 Overloading Constructors

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration:

```
// overloading class constructors
#include <iostream>
using namespace std;
class Rectangle {
    int x,y;
  public:
    Rectangle ();
    Rectangle (int,int);
    int area (void) {return (x*y);}
};
Rectangle::Rectangle () {
  x = 5;
  y = 5;
}
Rectangle::Rectangle (int a, int b) {
  x = a;
  y = b;
}
int main () {
```

```
  Rectangle rect1 (3,4);
  Rectangle rect2;
  cout << "rect1 area: " << rect1.area() << endl;
  cout << "rect2 area: " << rect2.area() << endl;
  return 0;
}
```

In this case, rect2 was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both x and y with a value of 5 while rect1 was declared with two arguments, so it has been initialized with constructor having two parameters, which initializes x to value of a that is 3 and y to value of b that is 4.

Important: Notice how if we declare a new object and we want to use its default constructor (the one without parameters), we do not include parentheses ():

```
Rectangle rect2;   // right
Rectangle rect2(); // wrong!
```

2.4.7 Copying an Object

After creating an object and assigning appropriate values to its members, you can perform any regular operation on it. Although this gets a little particular with objects, which will be expanded when learning about operator overloading, you can assign an object to another object. We have already learned:

How to assign	Example
A value to a variable	int a = 250;
The value of one variable to another	NbrOfBoys = NbrOfGirls;
A value to an object's member	Video.Category = "Drama"

Assigning a variable to another is equivalent to making a copy of that variable. As you assign a variable to another, you can assign one object to another. Both objects must be recognizably equivalent to the compiler. Imagine you want build the same Rectangle twice. All you have to do is to assign one Rectangle to another, which is take care of in the following main() function:

```
#include <iostream>
#include<conio>
using namespace std;
class Rectangle
{
 int x, y;
 public:
  Rectangle(){};                          // Default Constructor
  Rectangle(int a, int b);                // Parameterized
Constructor function
  void show();
  int area () {return (x*y);}
};
 Rectangle::Rectangle(int a, int b)                  // Parameterized
Constructor function
   {
      x=a;
      y=b;
   }
Rectangle::void show()
   {
```

```
    cout<<"\n x = "<<x;
    cout<<"\n y = "<<y;
    cout<<"\n Area = "<<area();
   }
int main ()
{
  clrscr();
  Rectangle rect1(3,4);
  Rectangle rect2;
  rect2 = rect1;
  cout << "\n rect1 :\n" ;
  rect1.show();
  cout<<endl;
  cout << "\n rect2 :\n" ;
  rect2.show();
  return 0;
}
```

Here is an example of running the program:

```
rect1 :
 x = 3
 y = 4
 Area = 12
 rect2 :
 x = 3
```

y = 4

Area = 12

Notice that both orders display the same thing.

2.4.8 Using Copy Constructor

Besides the default constructor, the compiler creates another function method called the copy constructor. This is another special method that is used for operations such as copying an object into another. When you have two instances of an object such as:

Rectangle rect1, rect2;

you can assign one object to another like this:

rect1 = rect2;

This operation indeed assigns a copy of the rect1 to the rect2 object. Behind the scenes, this transaction is handled by the copy constructor. Like the default constructor, the compiler automatically creates a copy constructor when an object is instantiated. Like the default constructor, you can explicitly create a copy constructor; it has a different syntax although it also holds the same name as the object. The syntax of the copy constructor.

ObjectName(ObjectName& Name);

The copy constructor takes one argument, which is the same as the object itself. When a copy is made, it holds and carries the building constructor of the object. This object is specified as the argument. As a copy whose value still resides with the object, this argument should be passed as a reference. As a copy, this argument should not be modified. It is only used to pass a copy of the object to the other objects that need it. Therefore, the argument should not be modified. As a result, it should be declared as a constant. The syntax of the copy constructor becomes:

ObjectName(const ObjectName& Name);

To copy one object to another, first create a copy constructor. In the implementation, assign a member variable of the copy constructor to the equivalent member of the object:

```
#include <iostream>
#include<conio>
using namespace std;
class Rectangle
{
  int x, y;
  public:
    Rectangle(){};                          // Default Constructor
    Rectangle(int a, int b);                // Parameterized
Constructor function
    Rectangle(const Rectangle &r);          // Copy Constructor
function
    void show();
    int area () {return (x*y);}
};
Rectangle::Rectangle(int a, int b)          // Parameterized
Constructor function
     {
        x=a;
        y=b;
     }
Rectangle::Rectangle(const Rectangle &r)    // Copy
Constructor function
    {
          x=r.x;
```

```
        y=r.y;
    }
void Rectangle:show()
    {
     cout<<"\n x = "<<x;
     cout<<"\n y = "<<y;
     cout<<"\n Area = "<<area();
    }
```

Here is an example that uses a copy constructor to copy one order into another:

```
int main ()
{
  clrscr();
  Rectangle rect1(3,4);                    // Paramterized Constructor
called
  Rectangle rect2(rect1);                 //  Copy Constructor called
  Rectangle rect3 = rect1;                //  Copy Constructor called again

  Rectangle rect4;
  rect4 = rect1;     // Copy Constructor not called
  cout << "\n rect1 :\n" ;
  rect1.show();
  cout<<endl;
  cout << "\n rect2 :\n" ;
  rect2.show();
```

```
  cout<<endl;
  cout << "\n rect3 :\n" ;
  rect3.show();
  cout<<endl;
  cout << "\n rect4 :\n" ;
  rect4.show();
return 0;
}
```

This would produce:

```
 rect1 :
 x = 3
 y = 4
 Area = 12
 rect2 :
 x = 3
 y = 4
 Area = 12
 rect3 :
 x = 3
 y = 4
 Area = 12
 rect4 :
 x = 3
 y = 4
```

Area = 12

2.4.9 Destructors

Destructors are also special member functions used in C++ programming language. Destructors have the opposite function of a constructor. The main use of destructors is to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically called when an object is destroyed. Like constructors, destructors also take the same name as that of the class name.

General Syntax of Destructors

~ classname();

The above is the general syntax of a destructor. In the above, the symbol tilde ~ represents a destructor which precedes the name of the class. For example:

```
class X {
public:
  // Constructor for class X
  X();
  // Destructor for class X
  ~X();
};
```

A destructor takes no arguments and has no return type. Its address cannot be taken. Destructors cannot be declared const, volatile, const volatile or static. A destructor can be declared virtual or pure virtual. If no user-defined destructor exists for a class and one is needed, the compiler implicitly declares a destructor. This implicitly declared destructor is an inline public member of its class. The compiler will implicitly define an implicitly declared destructor when the compiler uses the destructor to destroy an object of the destructor's class type. Suppose a class A has an implicitly declared destructor. The following is equivalent to the function the compiler would implicitly define for class A:

```
A::~A() { }
```

The compiler first implicitly defines the implicitly declared destructors of the base classes and nonstatic data members of a class A before defining the implicitly declared destructor of A.

The destructors have the following properties:

1. Destructors take the same name as the class name.
2. Like the constructor, the destructor must also be defined in the public. The destructor must be a public member.
3. The Destructor does not take any argument which means that destructors cannot be overloaded.
4. No return type is specified for destructors.
5. Destructors cannot be declared const, volatile, const volatile or static.
6. A destructor can be declared virtual or pure virtual.
7. If no user-defined destructor exists for a class and one is needed, the compiler implicitly declares a destructor.

The following example demonstrates the use of Constructor and Destructor in a program:

```
#include <iostream>
using namespace std;
class myclass

{
   int a;
  public:
   myclass();                // constructor
   ~myclass();               // destructor
   void show();
};
myclass::myclass()
{
  cout << "In constructor\n";
  a = 10;
```

```
}
myclass::~myclass()
{
  cout << "Destructing...\n";
}
void myclass::show()
{
  cout << a << endl;
}
int main()
{
  myclass ob;
  ob.show();
  return 0;
}
```

Output is:

In constructor

10

Destructing

The above program contains both a constructor**(myclass())** and the destructor**(~myclass)**. The constructor function is invoked itself when we create the object of the class as:

myclass ob;

When the constructor function is executed it prints the line:

In constructor

and assigns value of 10 to a. After that we call the **show()** function, which displays the value of variable – a. Now when the main function is exited the object **ob** becomes out of scope and hence must be destroyed. When the object is destroyed, the destructor function is executed which displays the line:

Destructing

Note: Dynamic Constructors will be discussed in the chapter on Dynamic Memory Allocation and Virtual Destructors in the chapter on Inheritance.

2.4.10 Summary

C++ provides special functions for initializing objects when they are created. These functions are called Constructors. The constructor functions have same name as that of the class in which they are defined and do not have any return type. The default constructor is the constructor used to create an object when you don't provide explicit initialization values. But after you define any constructor for a class, the responsibility for providing a default constructor for that class passes from the compiler to you. The constructors that can take arguments are called parameterized constructors. Besides the default constructor, the compiler creates another function method called the copy constructor. This is another special method that is used for operations such as copying an object into another. Like other functions, we can define more than one constructor in the same class and can distinguish between them by number and type of arguments. Destructors are opposite of constructors. They are executed when the object is being deleted.

2.4.11 Short Answer Type Questions

1. What is a Constructor? What are its properties?
2. What is a Destructor? What are its properties?
3. Discuss Default Constructor?

2.4.12 Long Answer Type Questions

1. Explain different types of Constructors in C++.
2. Explain copy constructor in detail with the help of an example.
3. Write a program which contains multiple constructors and a destructor.

2.4.13 Suggested Readings

1. Object Oriented Programming in C++ Robert Lafore
2. The Complete Reference C++ Herbert Schildt
3. Object Oriented Programming with C++ E Balagurusamy

]   .

B.A. PART-III | **PAPER : BAP-301**

SEMESTER-V | **OBJECT ORIENTED PROGRAMMING USING C++**

LESSON NO. 2.5 | **AUTHOR : KANWAL PREET SINGH**

DYNAMIC MEMORY ALLOCATION

2.5.1 Introduction

2.5.2 Objective

2.5.3 Dynamic Memory

2.5.4 Allocating Memory

2.5.5 Freeing-Up Allocated Memory

2.5.6 Allocating Dynamic Memory to Arrays

2.5.7 Allocating Dynamic Memory to Objects

2.5.8 Dynamic Constructors

2.5.9 Memory corruption

2.5.10 Nested Classes

2.5.11 Container Class

2.5.12 Summary

2.5.13 Short Answer Type Questions

2.5.14 Long Answer Type Questions

2.5.15 Suggested Readings

2.5.1 Introduction

When you have just finished your program, you would like to see it at work. Usually you press a combination of keys to compile and run it. But what actually happens when you press those keys? The compilation process translates your code from C++ to machine language, which is the only language computers understand. Then your application is given a certain amount of memory to use, which is divided into three segments as follows:

- **Code Segment**, in which all the application code is stored.
- **Data Segment**, which holds the global data.
- **Stack Segment**, used as a container for local variables and other temporary information.

In addition to these, the operating system also provides an extra amount of memory called **Heap**.

Now let us recall what a C++ variable consisted of:

- **name**, an identifier to recognize the variable.
- **type**, the set of values it may take.
- **scope**, the area of code in which it is visible.
- **address**, the address at which the variable is allocated.
- **size**, the amount of memory used to store its contents.

Consider the following sequence of code :

```
void fun() {
  int integer_X;
  ...
}
```

Let us try to figure out what are the attributes of the variable declared inside this function.

- name : integer_X

- type : int
- scope : local, only visible within the fun function
- address : a certain address in the stack segment, where the variable is allocated
- size : 16 bits (depending on the compiler)

What is a pointer then? We all heard about it, it is the variable declared with a star in front of it, but let us try and put some order in what we know by enumerating its exact attributes :

- name : an arbitrary identifier
- type : integer address
- scope : local or global (depending on the situation)
- address : the address in the data segment (if it is global) or stack segment (if it is local)
- size : 32 bits (16 bits for segment + 16 bits for offset of the address it points to)

2.5.2 Objective

After reading this chapter you will be able to understand:

- How to allocate and deallocate dynamic memory to a variable.
- How to allocate and deallocate dynamic memory to an array.
- How to allocate and deallocate dynamic memory to an object.
- Dynamic Constructors.
- Nesting of classes.
- Containership.

2.5.3 Dynamic Memory

Until now, in all our programs, we have only had as much memory available as we declared for our variables, having the size of all of them to be determined in the source code, before the execution of the program. But, what if we need a variable

amount of memory that can only be determined during runtime? For example, in the case that we need some user input to determine the necessary amount of memory space. The answer is **dynamic memory,** for which C++ integrates the operators new and delete.

A **dynamic variable** is a variable that is stored in the heap, outside the data and stack segments, that may change size during runtime or may even disappear. In fact if we do not allocate a certain amount of memory to store its contents, it will never exist.

2.5.4 Allocating Memory

Dynamic memory is allocated by the **new** keyword. Memory for one variable is allocated as below:

DataType ptr=new DataType (initializer);

Here,

- **ptr** is a valid pointer of type DataType.
- **DataType** is any valid C++ data type.
- **initializer** (optional) if given, the newly allocated variable is initialized to that value.

If the call to the new operator is successful, it returns a pointer to the space that is allocated. Otherwise, if the space could not be found or if some kind of error is detected it returns the address zero. For example consider the next declaration:

double *dp;

This tells the compiler we have a pointer **dp** that may hold the starting address of a dynamic variable of type double. However, the dynamic variable does not exist yet. To create it we must use the new operator like this:

dp = new double;

Remember one thing - dynamic variables are never initialized by the compiler. Therefore it is good practice to first assign them a value, or your calculations may not come out right. There are two ways of doing this:

dp = new double;

***dp = a;**

or

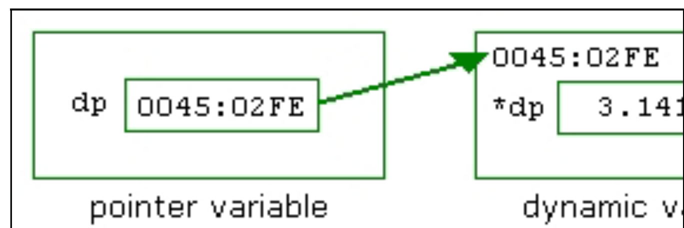
dp = new double(a);

where **a** is the initial value. The second version is recommended as it is more suggestive and also more compact. Consider this line of code :

double *dp = new double(3.1415);

The Figure-1 below shows the possible connection between dp and the dynamic variable associated with it, after the previous line is executed.

Figure-1



2.5.5 Freeing-Up Allocated Memory

Unlike static variables, C++ will not free-up the memory allocated through dynamic allocation. It is your duty to free them up. **delete** keyword is used to free-up the allocated memory.

delete ptr;

Now consider you have decided to erase the dynamic variable from the heap associated with pointer dp. Use the delete operator to achieve this:

delete dp;

Notice that while we have freed 8 bytes of memory - which is sizeof(double) - the dp pointer was not erased. It still exists in the data or stack segment of the application and we may use it to initialize another dynamic variable. The following program shows how to create and delete a dynamic variable using operators – new and delete respectively.

//Example Program showing how to create and delete a Dynamic Variable

```
#include<iostream.h>
void main(void)
{
 int *ptr;
 ptr=new int(10);
 cout << "At " << ptr << " is the value " << *ptr << "\n";
 delete ptr;
}
```

Output is:

At 0x3d2448 is the value 10

The above program will allocate memory for an int(eger) having initial value 10, pointed by the **ptr** pointer. In the output, the value of ptr may vary according to the memory allocated to that pointer.

2.5.6 Allocating Dynamic Memory to Arrays

C++ also provides the possibility to allocate an entire array in the heap, keeping a pointer to its first element. Again we use the new operator but mention the size of the array in square brackets. Memory space for arrays is allocated as shown below:

DataType ptr=new DataType [x];

Here,

- **ptr** and DataType have the same meaning as above.
- **x** is the number of elements and C is a constant.

For Example:

int *table;

```
table = new int[100];
```

This will allocate 100 * sizeof(int) = 100 * 2 = 200 bytes of memory and assign the starting address of the memory block to the table pointer. Arrays allocated in the heap are similar to those allocated in the data or stack segments, so we may access an arbitrary element using the indexing operator []. For example the next loop initializes each element of the array to zero :

```
for (int i = 0; i < 100; ++i)
        table[i] = 0;
```

C++ does not provide any method of initializing heap arrays as you would an ordinary dynamic variable, so we have to do it ourselves using a loop similar to the one above. The following line will generate errors at compilation :

```
table = new int[100](0);
```

To erase a heap-allocated array we will use the delete operator, but this time add a pair of square brackets so that the compiler can differentiate it from an ordinary dynamic variable. Syntax is:

```
delete []ptr;
```

For example, the above created table array can be deleted as:

```
delete [] table;
```

The following program shows how to create and delete a dynamic array using operators – new and delete respectively.

```
//Example Program showing how to create and delete a Dynamic Array
#include <iostream.h>
using namespace std;
int main()
{
   int *p, i;
```

```
    p = new int [10]; // allocate 10 integer array
    for(i=0; i<10; i++ )
    p[i] = i;
    for(i=0; i<10; i++)
    cout << p[i] << " ";
   delete [] p; // release the array
   return 0;
}
```

Output is:

0 1 2 3 4 5 6 7 8 9

All we have learned so far are means of replacing the old malloc() and free() functions in C with their new and delete C++ analogues. The main reason is that they are better alternatives to dynamic memory allocation and also have a more compact syntax. However C pointer arithmetics and other pointer operations are also available in C++.

Pointers to classes

It is perfectly valid to create pointers that point to classes. We simply have to consider that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer. For example:

```
Rectangle * prect;
```

is a pointer to an object of class **Rectangle.** In order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (->) of indirection.

2.5.7 Allocating Dynamic Memory to Objects

Till now, we initialized the objects at declaration time. However, like other variables, the objects can also be initialized at program execution time(run time). This is called dynamic initialization of objects. A dynamic object can be created by execution of the new operator. The syntax for creating a dynamic object using the new operator is:

new ClassName;

The above line returns the address of a newly created object. The returned address can be stored in a variable of type pointer to object as:

ptr_name = new ClassName;

While creating a dynamic object, if a class has the default constructor, it is invoked as part of object creation activity. If there is a parameterized constructor, then we should provide the required arguments along with the ClassName to the new operator.

The dynamic object created by using new can be destroyed by using delete operator. The syntax is:

delete ptr_name;

The above statement destroys the object pointed to by ptr_name. It also invokes destructor of the class if it exists as part of the object destruction activity. The following program shows how to create, access and destroy a dynamic object.

//Example Program showing how to create and delete a Dynamic Object

```
#include <iostream>
using namespace std;
class Rectangle
 {
   int width;
   int height;
  public:
   Rectangle(int w, int h)
  {
   width = w;
   height = h;
   cout << "\n
Constructing " << width << " by " << height << " rectangle.\n";
  }
```

```
  ~Rectangle() {
    cout << "\n
Destructing " << width << " by " << height << " rectangle.\n";
  }
  int area()
  {
   return width * height;
  }
};
 int main()
{
 Rectangle *p;
 p = new Rectangle(10, 8);
 cout << "\n Area is " << p->area();
 delete p;
 return 0;
}
```

The output is:

Constructing 10 by 8 rectangle.

Area is 80

Destructing 10 by 8 rectangle.

2.5.8 Dynamic Constructors

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of new operator. The following program shows the use of new in constructor that is used to construct strings in objects.

// using new to get memory for strings

```
#include <iostream>
#include <cstring>      //for strcpy(), etc
using namespace std;

class String            //user-defined string type
 {
 private:
   char* str;                  //pointer to string
 public:
   String(char* s)              //constructor, one arg
   {
   int length = strlen(s);      //length of string argument
   str = new char[length+1];    //get memory
   strcpy(str, s);              //copy argument to it
   }
   ~String()                    //destructor
   {
   delete[] str;                //release memory
   }
   void display()               //display the String
   {
   cout << str << endl;
   }
 };
```

```
int main()
  {                          //uses 1-arg constructor
  String s1 = "Work hard to get good marks in C++.";
  String s2 = "Otherwise, you may fail.";
  cout << "s1=";             //display string
  s1.display();
  cout << "s2=";             //display string
  s2.display();
  return 0;
  }
```

The **String** class has only one data item: a pointer to **char**, called **str.** This pointer will point to the string held by the **String** object. There is no array within the object to hold the string. The string is stored elsewhere, only the pointer to it is a member of **String.** The constructor in this example takes a normal char* string as its argument. It obtains space in memory for this string with **new, str** points to the newly obtained memory. The constructor then uses **strcpy()** to copy the string into this new space. In the above example we create two objects. In earlier examples memory allocated to each object was equal, but in this case different memory will be allocated to different objects according to the size of the string passed to the constructor. If we allocate memory when we create an object, it's reasonable to deallocate the memory when the object is no longer needed. The destructor gives back to the system the memory obtained when the object was created. Objects (like other variables) are typically destroyed when the function in which they were defined terminates. This destructor ensures that memory obtained by the **String** object will be returned to the system, and not left in indeterminate state, when the object is destroyed.

2.5.9 Memory corruption

Memory is said to be corrupted if we try one of the following :

- Free a dynamic variable that has already been freed

```
char *str = new char[100];
delete [] str;
delete [] str;
```

- Free a dynamic variable that has not been yet allocated

```
char *str;
delete str;
```

- Assign a certain value to a dynamic variable that was never allocated

```
char *str;
strcpy(str, "error");
```

The previous three examples will most probably cause the application to crash. However the next two "bugs" are harder to detect:

- Assign a certain value to a dynamic variable, after it has been freed (this may also affect other data stored in the heap)

```
char *str = new char[100];
delete [] str;
char *test = new char[100];
strcpy(str, "surprise !");
cout << test;
delete [] test;
```

- Access an element with an index out of range

```
char *str = new char[30];
str[40] = 'C';
```

Last but not least, remember it is good practice to test whether allocation was or was not successful before proceeding with the code. The reason for this is that the operating system may run out of heap at some point, or the memory may get too fragmented to allow another allocation.

```
char *str = new char[512];
if (str == NULL) {
   // unable to allocate block of memory !
}
```

2.5.10 Nested Classes

In C++, you can place a class declaration inside another class. The class declared within another is called a **nested class,** and it helps avoid name clutter by giving the new type class scope. The name of a nested class is local to its enclosing class. Unless you use explicit pointers, references, or object names, declarations in a nested class, it can only use visible constructs, including type names, static members, and enumerators from the enclosing class and global variables. Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes. Member functions of the enclosing class have no special access to members of a nested class. Member functions of the class containing the declaration can create and use objects of the nested class. The outside world can use the nested class only if the declaration is in the public section and if you use the scope resolution operator. The following example demonstrates this:

```
class A
 {
   int x;
   class B { };
   class C
   {
     // The compiler cannot allow the following
     // declaration because A::B is private:
     //   B b;
      int y;
```

```
    void f(A* p, int i)
     {
       // The compiler cannot allow the following
       // statement because A::x is private:
       //   p->x = i;
      }
  };
  void g(C* p)
   {
    // The compiler cannot allow the following
    // statement because C::y is private:
    //   int z = p->y;
   }
};
int main() { }
```

The compiler would not allow the declaration of object b because class **A::B** is private. The compiler would not allow the statement **p->x = i** because **A::x** is private. The compiler would not allow the statement **int z = p->y** because **C::y** is private. You can define member functions and static data members of a nested class in namespace scope.

Nesting classes is not the same as containment. Containment, means having a class object as a member of another class. Nesting a class, on the other hand, does not create a class member. Instead, it defines a type that is known just locally to the class containing the nested class declaration. The usual reasons for nesting a class are to assist the implementation of another class and to avoid name conflicts.

Scope of Nested Class

If the nested class is declared in a private section of a second class, it is known only to that second class. If the nested class is declared in a protected section of a second class, it is visible to that class but invisible to the outside world. However, in this case, a derived class would know about the nested class and could directly create objects of that type. If a nested class is declared in a public section of a second class, it is available to the second class, to classes derived from the second class, and, because it's public, to the outside world. However, because the nested class has class scope, it has to be used with a class qualifier in the outside world. For example, suppose you have this declaration:

```
class Team
{
public:
    class Coach { ... };
    ...
};
```

Now suppose you have an unemployed coach, one who belongs to no team. To create a **Coach** object outside of the **Team** class, you can do this:

```
Team::Coach forhire; // create a Coach object outside the Team class
```

The following Program illustrates how to create a nested class and how to access its objects inside the class and outside the class in which it is created:

```
#include <iostream>
#include <stdlib.h>
using namespace std;
class OuterClass
{
public:
   void outerFunction();
     class InnerClass
```

```
    {
    public:
        void innerFunction1();
        void innerFunction2();
    };
};
void OuterClass::outerFunction()
{
 cout << "This is a function in the base class\n";
 InnerClass myclass;     // Accessing InnerClass object inside
OuterClass
 myclass.innerFunction1();
}
void OuterClass::InnerClass::innerFunction1()
{
  cout << "This is a function in the InnerClass class accessed from
inside OuterClass \n";
}
void OuterClass::InnerClass::innerFunction2()
{
  cout << "This is a function in the InnerClass class accessed from
outside OuterClass \n";
}
int main()
{
 OuterClass myclass;
 OuterClass::InnerClass inner;
 myclass.outerFunction();
 inner.innerFunction2();
 return 0;
}
```

The output is:

This is a function in the base class

This is a function in the InnerClass class accessed from inside OuterClass

This is a function in the InnerClass class accessed from outside OuterClass

2.5.11 Container Class

Containment, means having a class object as a member of another class. In inheritance, if a class B is derived from a class A, we can say that "B is a kind of A." This is because B has all the characteristics of A, and in addition some of its own. There's another type of relationship, called a **"has a"** relationship, or **containership.** We say that a cat **has a** tail, meaning that each cat includes an instance of a tail. In Object-Oriented Programming the "has a" relationship occurs when one object is contained in another. Here's a case where an object of class A is contained in a class B:

```
class A
  {
  };
class B
  {
  A objA;  // define objA as an object of class A
  };
```

All the objects of B will contain object objA which is an instance of class A. So, we say that B Contains A and therefore B is called the container class. Creation of an object that contains another object is different from the creation of an independent object. An independent object is created by its constructor when it is declared with arguments. On the other hand, a nested object is created in two stages: First, the member objects are created using their respective constructors

and then the ordinary members are created. This means, constructor of all the member objects should be called before its own constructor body is executed.

2.5.12 Summary

A **dynamic variable** is a variable that is stored in the heap, outside the data and stack segments, that may change size during runtime. Dynamic memory is allocated by the **new** keyword. If the call to the new operator is successful, it returns a pointer to the space that is allocated. Otherwise, if the space could not be found or if some kind of error is detected it returns the address zero. Unlike static variables, C++ will not free-up the memory allocated through dynamic allocation. It is your duty to free them up. **delete** keyword is used to free-up the allocated memory. C++ also provides the possibility to allocate an entire array in the heap, keeping a pointer to its first element. Again we use the new operator but mention the size of the array in square brackets. like other variables, the objects can also be initialized at program execution time(run time). This is called dynamic initialization of objects. A dynamic object can be created by execution of the new operator. The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in saving of memory. In C++, you can place a class declaration inside another class. The class declared within another is called a **nested class.** Containment, means having a class object as a member of another class.

2.5.13 Short Answer Type Questions

1. Name the three segments in which program data is stored.
2. What is Dynamic memory?
3. What are Dynamic Constructors?
4. Can we have pointer to classes? How are they declared?

2.5.14 Long Answer Type Questions

1. Explain in detail with examples how can you dynamically create and destroy variables and arrays in C++.
2. Explain the difference between nesting of classes and containership.

2.5.15 Suggested Readings

1. Object Oriented Programming in C++ Robert Lafore
2. The Complete Reference C++ Herbert Schildt
3. Object Oriented Programming with C++ E Balagurusamy

B.A. PART-III **PAPER : BAP-301**

SEMESTER-V **OBJECT ORIENTED PROGRAMMING USING C++**

LESSON NO. 2.6 **AUTHOR : KANWAL PREET SINGH**

SCOPES

2.6.1 Introduction

2.6.2 Objective

2.6.3 Scope

2.6.4 Linkage

2.6.5 Summary

2.6.6 Short Answer Type Questions

2.6.7 Long Answer Type Questions

2.6.8 Suggested Readings

2.6.1 Introduction

C++ names can be used only in certain regions of a program. This area is called the "scope" of the name. In this lesson, we will study different kinds of scope in C++. Mainly scope can be of four types – local, global, class and namespace. All four kinds of scope are discussed later in the chapter. Also program variables have a storage class in addition to a data type. The different type of storage classes used in C++ are - auto, static, register, extern, and mutable.

2.6.2 Objective

After reading this lesson you will be able to understand

- Scope and Visibility
- Linkage
- Storage Classes
- Different types of scope

2.6.3 Scope

C++ names can be used only in certain regions of a program. This area is called the "scope" of the name. Scope determines the "lifetime" of a name that does not denote an object of static extent. Broadly speaking, scope is the general context used to differentiate the meanings of entity names. The rules for scope combined with those for name resolution enable the compiler to determine whether a reference to an identifier is legal at a given point in a file.

The scope of a declaration and the visibility of an identifier are related but distinct concepts. Scope is the mechanism by which it is possible to limit the visibility of declarations in a program. The visibility of an identifier is the region of program text from which the object associated with the identifier can be legally accessed. Scope can exceed visibility, but visibility cannot exceed scope. Scope exceeds visibility when a duplicate identifier is used in an inner declarative region, thereby hiding the object declared in the outer declarative region. The original identifier cannot be used to access the first object until the scope of the duplicate identifier (the lifetime of the second object) has ended.

Thus, the scope of an identifier is interrelated with the storage duration of the identified object, which is the length of time that an object remains in an identified region of storage. The lifetime of the object is influenced by its storage duration, which in turn is affected by the scope of the object identifier.

- **Block/local scope**

A name has local scope or block scope if it is declared in a block. A name with local scope can be used in that block and in blocks enclosed within that block, but the name must be declared before it is used. When the block is exited, the names declared in the block are no longer available.

Parameter names for a function have the scope of the outermost block of that function. Also, if the function is declared and not defined, these parameter names have function prototype scope.

When one block is nested inside another, the variables from the outer block are usually visible in the nested block. However, if the declaration of a variable in a nested block has the same name as a variable that is declared in an enclosing block, the declaration in the nested block hides the variable that was declared in the enclosing block. The original declaration is restored when program control returns to the outer block. This is called block visibility.

Name resolution in a local scope begins in the immediate scope in which the name is used and continues outward with each enclosing scope. The order in which scopes are searched during name resolution causes the phenomenon of information hiding. A declaration in an enclosing scope is hidden by a declaration of the same identifier in a nested scope.

- **Function prototype scope**

In a function declaration (also called a function prototype) or in any function declaratory except the declarator of a function definition--parameter names have function prototype scope. Function prototype scope terminates at the end of the nearest enclosing function declarator.

- **File/global scope**

Global scope or global namespace scope is the outermost namespace scope of a program, in which objects, functions, types and templates can be defined. A name has global namespace scope if the identifier's declaration appears outside of all blocks, namespaces, and classes.

A name with global namespace scope and internal linkage is visible from the point where it is declared to the end of the translation unit.

A name with global (namespace) scope is also accessible for the initialization of global variables. If that name is declared **extern,** it is also visible at link time in all object files being linked.

A user-defined namespace can be nested within the global scope using namespace definitions, and each user-defined namespace is a different scope, distinct from the global scope.

- **Class scope**

A name declared within a member function hides a declaration of the same name whose scope extends to or past the end of the member function's class.

When the scope of a declaration extends to or past the end of a class definition, the regions defined by the member definitions of that class are included in the scope of the class. Members defined lexically outside of the class are also in this scope. In addition, the scope of the declaration includes any portion of the declarator following the identifier in the member definitions.

The name of a class member has class scope and can only be used in the following cases:

- In a member function of that class
- In a member function of a class derived from that class
- After the . (dot) operator applied to an instance of that class
- After the . (dot) operator applied to an instance of a class derived from that class, as long as the derived class does not hide the name
- After the -> (arrow) operator applied to a pointer to an instance of that class
- After the -> (arrow) operator applied to a pointer to an instance of a class derived from that class, as long as the derived class does not hide the name
- After the :: (scope resolution) operator applied to the name of a class
- After the :: (scope resolution) operator applied to a class derived from that class

- **Namespace Scope**

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name. The format of namespaces is:

```
namespace identifier
{
entities
}
```

Where **identifier** is any valid identifier and **entities** is the set of classes, objects and functions that are included within the namespace. The creation of a namespace starts with the **namespace** keyword followed by a name that would identify the section of code. The name follows the rules we have been applying to C++ names. A namespace has a body: this is where the entities that are part of the namespace would be declared or defined. The body of the namespace starts with an opening curly bracket "{" and ends with a closing curly bracket "}". For example:

```
namespace Mine
{
   int a;
}
```

The entities included in the body of a namespace are referred to as its members.

Note: Namespace definitions cannot appear inside functions; that is, they must appear outside functions at global scope.

- **Accessing a namespace: The Scope Access Operator**

To access a member of a namespace, there are various techniques you can use. The scope access operator **"::"** is used to access the members of a namespace. To do this, type the name of the namespace, followed by the scope access operator **"::",** followed by the member you are want to access. Only the members of a particular namespace are available when using its name. For example, to access the member **"a"** of the Mine namespace above, you can write:

```
Mine::a;
```

Once you have access to a namespace member, you can initialize it or display its value using **cout**. Here is an example:

```
#include <iostream>
using namespace std;
namespace Mine
{
	int a;
}
int main()
{
	Mine::a = 140;

	cout << "Value of a = " << Mine::a << endl;
	return 0;
}
```

The output is:

Value of a = 140

When creating a namespace, you can add as many members as you see fit. When necessary, you can use the scope access operator "**::**" to call anyone of them as needed. You can also request the values of the members of a namespace from the user. Remember to use the scope access operator "**::**" whenever you need to access the member of a namespace. The member variable of a namespace can also be mixed with a variable that is locally declared in a function. All you have to do is make sure that you qualify the member of the namespace so the compiler would be able to locate it. Here is an example:

```
#include <iostream>
using namespace std;
namespace InterestAndDiscount
```

```
{
        double principal;
        double rate;
        int period;
}
int main()
{
        double interest;
        double maturityValue;
        cout << "Interest and Discount\n";
        cout << "Principal: Rs.";
        cin >> InterestAndDiscount::principal;
        cout << "Rate (example 8.75): ";
        cin >> InterestAndDiscount::rate;
        cout << "Number of Years: ";
        cin >> InterestAndDiscount::period;
        interest = InterestAndDiscount::principal *
                        (InterestAndDiscount::rate/100) *
                        InterestAndDiscount::period;
        maturityValue = InterestAndDiscount::principal + interest;
        cout << "\nLoan Processing";
        cout << "\nPrincipal: Rs." << InterestAndDiscount::principal;
        cout << "\nRate:        " << InterestAndDiscount::rate << "%";
```

```
        cout << "\nPeriod:    " << InterestAndDiscount::period << "
years";
        cout << "\nInterest: Rs." << interest;
        cout << "\nMaturity Value: Rs." << maturityValue << "\n\n";
        return 0;
}
```

- **The using Keyword**

The scope access operator **"::"**provides a safe mechanism to access the members of a namespace. If the namespace is very long and the application needs constant access, this might be a little cumbersome. Another technique used to access the members of a namespace involves using two keywords: **using** and **namespace**.

To call a namespace, on the section of the program where you need to access the members, type:

using namespace NamespaceName;

Both the using and the namespace keywords are required by the compiler. The **NamespaceName** represents the name of the namespace whose member(s) you want to access. Using this technique, the above program can be written:

```
#include <iostream>
using namespace std;
namespace InterestAndDiscount
{
        double principal;
        double rate;
        int periods;
}
int main()
```

```
{
        using namespace InterestAndDiscount;
        double interest;
        double maturityValue;
        cout << "Interest and Discount\n";
        cout << "Principal: Rs.";
        cin >> principal;
        cout << "Rate (example 8.75): ";
        cin >> rate;
        cout << "Number of Years: ";
        cin >> periods;
        interest = principal * (rate/100) * periods;
        maturityValue = principal + interest;
        cout << "\nLoan Processing";
        cout << "\nPrincipal: Rs." << principal;
        cout << "\nRate:        " << rate << "%";
        cout << "\nPeriods:    " << periods << " years";
        cout << "\nInterest: Rs." << interest;
        cout << "\nMaturity Value: Rs." << maturityValue << "\n\n";

        return 0;
}
```

Here is an example of a result:

Interest and Discount

Principal: Rs.2500

Rate (example 8.75): 12.15

Number of Years: 4

Loan Processing

Principal: Rs.2500

Rate: 12.15%

Periods: 4 years

Interest: Rs.1215

Maturity Value: Rs.3715

In a variable intensive program (we are still inside of one function only) where a local variable holds the same name as the member of a namespace that is being accessed with the using namespace routine, you will be sensitive to the calling of the same name variable. When manipulating such a name of a variable that is present locally in the function as well as in the namespace that is being accessed, the compiler will require more precision from you. You will need to specify what name is being called.

- **Namespace Extensions**

Namespaces are extensible; that is, you can append subsequent declarations to previously defined namespaces. Namespace extensions may also appear in files separate from the original namespace definition. The following two definitions of namespace Blue are equivalent to the definition of namespace BigBlue.

```
namespace Blue
{      // original namespace definition
  int j;
  void print(int);
}
```

```
namespace Blue
{      // namespace extension
  char ch;
  char buffer[20];
}
. . .
namespace BigBlue
{     // equivalent to the above
  int j;
  void print(int);
  char ch;
  char buffer[20];
}
```

When you create a namespace, avoid placing include files inside the namespace definition.

2.6.4 Linkage

Linkage refers to the use or availability of a name across multiple translation units or within a single translation unit. The term translation unit refers to a source code file plus all the header and other source files that are included after preprocessing with the #include directive, minus any source lines skipped because of conditional preprocessing directives. Linkage allows the correct association of each instance of an identifier with one particular object or function.

Scope and linkage are distinguishable in that scope is for the benefit of the compiler, whereas linkage is for the benefit of the linker. During the translation of a source file to object code, the compiler keeps track of the identifiers that have external linkage and eventually stores them in a table within the object file. The linker is thereby able to determine which names have external linkage, but is

unaware of those with internal or no linkage. The linkage of an identifier depends on how it was declared. There are three types of linkages:

- **Internal linkage** : identifiers can only be seen within a translation unit(file).
- **External linkage** : identifiers can be seen (and referred to) in other translation units(files).
- **No linkage**: identifiers can only be seen in the scope in which they are defined.

2.6.5 Summary

C++ names can be used only in certain regions of a program. This area is called the "scope" of the name. Scope determines the "lifetime" of a name that does not denote an object of static extent. Linkage refers to the use or availability of a name across multiple translation units or within a single translation unit. Program variables have a storage class in addition to a data type. The different type of storage classes used in C++ are - auto, static, register, extern, and mutable. A name has local scope or block scope if it is declared in a block. A name with local scope can be used in that block and in blocks enclosed within that block. Global scope or global namespace scope is the outermost namespace scope of a program, in which objects, functions, types and templates can be defined. A name has global namespace scope if the identifier's declaration appears outside of all blocks, namespaces, and classes. The members of a class come under class scope and certain rules have to be followed for accessing members in class scope. Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.

2.6.6 Short Answer Type Questions

1. What do you mean by scope of a variable?
2. Is there any difference between scope and visibility? Explain.
3. What do you understand by linking? Explain different types of linking.
4. Can you access members of a namespace outside it? How?

2.6.7 Long Answer Type Questions

1. Explain different storage classes used in C++.
2. Explain different kinds of scope in C++.

2.6.8 Suggested Readings

1. Object Oriented Programming in C++ Robert Lafore
2. The Complete Reference C++ Herbert Schildt
3. Object Oriented Programming with C++ E Balagurusamy

LESSON NO. 2.7

AUTHOR : KANWAL PREET SINGH

INHERITANCE

2.7.1 Introduction

2.7.2 Objective

2.7.3 Derived Class

2.7.4 Protected Access

2.7.5 Types of Derivation

2.7.6 Overriding Member Functions (Function Redefining)

2.7.7 Types of Inheritance

2.7.8 Virtual base classes

2.7.9 Constructors in Derived class

2.7.10 Types of Base Classes

2.7.11 Summary

2.7.12 Short Answer Type Questions

2.7.13 Long Answer Type Questions

2.7.14 Suggested Readings

2.7.1 Introduction

A key feature of C++ classes is inheritance. Inheritance allows creating classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own. Inheritance is a mechanism of reusing and extending existing classes without modifying them, thus producing hierarchical relationships between them. Inheritance is almost like embedding an object into a class. Suppose that you declare an object x of class A in the class definition of B. As a result, class B will have access to all the public data members and member functions of class A. However, in class B, you have to access the data members and member functions of class A through object x. The following example demonstrates this:

```
#include <iostream>
using namespace std;
class A
{
   int data;
  public:
   void f(int arg) { data = arg; }
   int g() { return data; }
};
class B
{
  public:
   A x;
};
int main()
 {
```

```
  B obj;
  obj.x.f(20);
  cout << obj.x.g() << endl;
//  cout << obj.g() << endl;
}
```

In the main function, object obj accesses function A::f() through its data member B::x with the statement obj.x.f(20). Object obj accesses A::g() in a similar manner with the statement obj.x.g(). The compiler would not allow the statement obj.g() because g() is a member function of class A, not class B.

The inheritance mechanism lets you use a statement like obj.g() in the above example. In order for that statement to be legal, g() must be a member function of class B. Inheritance lets you include the names and definitions of another class's members as part of a new class. The class whose members you want to include in your new class is called a base class. Your new class is derived from the base class. The new class contains a subobject of the type of the base class. The following example is the same as the previous example except it uses the inheritance mechanism to give class B access to the members of class A:

```
#include <iostream>
using namespace std;
class A
{
  int data;
 public:
  void f(int arg) { data = arg; }
  int g() { return data; }
};
class B : public A { };
```

```
int main()
{
  B obj;
  obj.f(20);
  cout << obj.g() << endl;
}
```

Class A is a base class of class B. The names and definitions of the members of class A are included in the definition of class B; class B inherits the members of class A. Class B is derived from class A. Class B contains a subobject of type A. You can also add new data members and member functions to the derived class. You can modify the implementation of existing member functions or data by overriding base class member functions or data in the newly derived class.

2.7.2 Objective

After reading this lesson you will be able to understand

- Inheritance
- Derived classes
- Types of derivation
- Protected members
- Method Overriding
- Types of inheritance
- Order of calling constructors in derived classes
- Types of base classes

2.7.3 Derived Class

If class **A** inherits from class **B**, then **B** is called **superclass** of **A**. **A** is called **subclass** of **B**. Objects of a subclass can be used where objects of the corresponding superclass are expected. This is due to the fact that objects of the subclass share the same behaviour as objects of the superclass. In the literature you may also find other terms for ``superclass'' and ``subclass''. Superclasses are

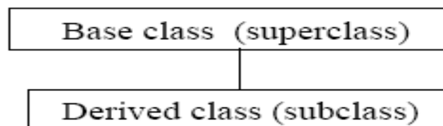
also called **parent** classes. Subclasses may also be called **child** classes or just **derived** classes.

When you declare a class, you can indicate what class it derives from by writing a colon after the class name, the type of derivation (public or otherwise), and the class from which it derives. The syntax is as follows:

class Child : [access-control] Parent

{...};

where access-control can be public, private or protected.



What a derived class inherits

- Every data member defined in the parent class (although such members may not always be accessible in the derived class)
- Every ordinary member function of the parent class (although such members may not always be accessible in the derived class)
- The same initial data layout as the base class

What a derived class doesn't inherit

- The base class's constructors and destructor
- The base class's assignment operator
- The base class's friends

What a derived class can add

- New data members
- New member functions
- New constructors and destructor

- New friends

Other Inheritance Rules

- If base class has a default constructor it is automatically called.
- Derived class inherits public & protected members access as:

  derivedclass.basefunction();

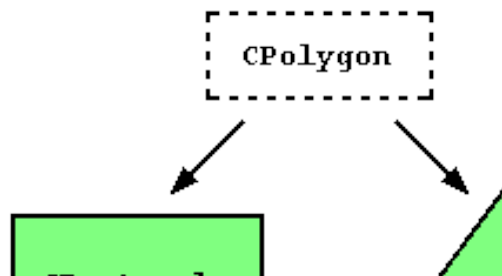
  derivedclass.basedata;

- Private members cannot be accessed in the derived class.
- If derived class member function of same name overrides the base class function, access base class function as:

  baseclass::basefunction();

Example:

We are going to suppose that we want to declare a series of classes that describe polygons like Rectangle, or like Triangle. They have certain common properties, such as both can be described by means of only two sides: height and base. This could be represented in the world of classes with a class Polygon from which we would derive the two other ones: Rectangle and Triangle.



The class Polygon would contain members that are common for both types of polygon. In our case: width and height. And Rectangle and Triangle would be its derived classes, with specific features that are different from one type of polygon to the other. Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive it to another class with another member called B, the derived class will contain both members A and B.

```
// derived classes
#include <iostream>
using namespace std;
class Polygon
{
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };
class Rectangle: public Polygon
{
  public:
    int area ()
      { return (width * height); }
  };
class Triangle: public Polygon
{
  public:
    int area ()
      { return (width * height / 2); }
  };
int main ()
```

```
{
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

Output is:

20

10

2.7.4 Protected Access

Private members of the base class are not accessible in the derived class (to preserve encapsulation). Sometimes, however, we would like to be able to define encapsulated data members which are not publicly accessible, but which are accessible to derived classes. In such a case, we make use of protected access specifier. Protected data members and functions are fully visible to derived classes, but are otherwise private.

The protected access specifier is similar to private. Its only difference occurs in fact with inheritance. When a class inherits from another one, the members of the derived class can access the protected members inherited from the base class, but not its private members. In the above example, since we wanted width and height to be accessible from members of the derived classes Rectangle and Triangle and not only by members of Polygon, we have used protected access instead of private.

2.7.5 Types of Derivation

When you declare a derived class, an access specifier can precede each base class in the base list of the derived class. This does not alter the access attributes of the

individual members of a base class as seen by the base class, but allows the derived class to restrict the access control of the members of a base class. You can derive classes using any of the three access specifiers:

1. **public**: In a public base class, public and protected members of the base class remain public and protected members of the derived class.
2. **private:** In a protected base class, public and protected members of the base class are protected members of the derived class.
3. **protected:** In a private base class, public and protected members of the base class become private members of the derived class.

In all cases, private members of the base class remain private. Private members of the base class cannot be used by the derived class unless friend declarations within the base class explicitly grant access to them. In the following example, class d is derived publicly from class b. Class b is declared a public base class by this declaration.

```
class b { };
class d : public b // public derivation
{ };
```

The following example illustrates the three modes of derivation:

- **class Parent { ... };**
- **class ChildPublic : public Parent { ... };**
- **class ChildProtected : protected Parent { ... };**
- **class ChildPrivate : private Parent { ... };**

Consider the following class Parent:

```
class Parent
{
  private:
    int priv;
  protected:
```

```
  int prot;
 public:
  int pub;
};
```

The above class contains a private member priv, a protected member prot and a public member pub. The following code illustrates whether these members are accessible in the Child class when the Child class is derived publicly, protectedly and privately respectively.

```
class Child : public Parent
{
 // Access to pub und prot
 // External access to pub
 // Subclasses: Access to pub und prot
};
class Child : protected Parent
{
 // Access to pub und prot
 // No external access to any variable
 // Subclasses: Access to pub und prot
};
class Child : private Parent
{
 // Access to pub und prot
 // No external access to any variable
 // Subclasses: No access to any variable
```

```
};
```

2.7.6 Overriding Member Functions (Function Redefining)

The derived class can have a member function with same name, same return type and same list of arguments as defined in the base class i.e When a derived class creates a function with the same return type and signature as a member function in the base class, but with a new implementation, it is said to be overriding that method.

```
#include<iostream>
using namespace std;
class Base
 {
   public:
    void display()
     {
       cout<<"\n Base class";
     }
  };
class Derived : public Base
 {
   public:
    void display()
     {
       cout<<"\n Derived class";
     }
  };
```

```
void main()
 {
   Base b;
   Derived d;
   b.display();
   d.display();
 }
```

Output is:

Base class

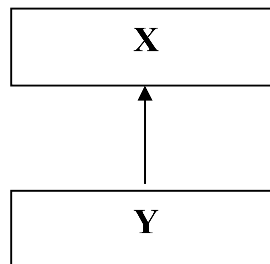
Derived class

In the above program the class Base defines a function – display(). The class Derived inherits Base. It also defines a function – display() , which has same name and type signature as display() function in the base class. So, we say that the display() function of derived class overrides the display() function of Base class. Here, when we call the display() function with the object of class Base, then the discplay() function of class Base will be called and when we call the display() function with the object of class Derived, then the discplay() function of class Derived will be called.

2.7.7 Types of Inheritance

Single Inheritance

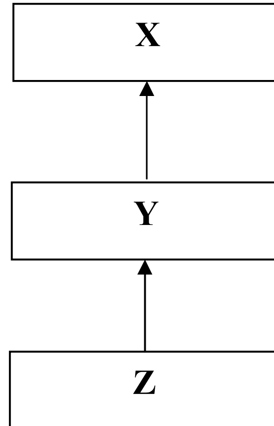
When only one class is derived from a single base class, such derivation of a class is known as single inheritance. Further the derived class is not used as a base class for another class derivation.



In the above figure X is the base class and Y is the derived class. This type of inheritance involves one base class and one derived class. Further no class is derived from Y.

Multilevel Inheritance

When a class is derived from a single base class and further can be used as a base class for deriving another class. This derivation can continue upto any level.



In the above figure X is the top base class and Y is the derived class of X. At the next level, class Z is derived from class Y. Therefore, class Y is not just a derived class of class X, but also base class for class Z. Further, Z can also be used as base class.

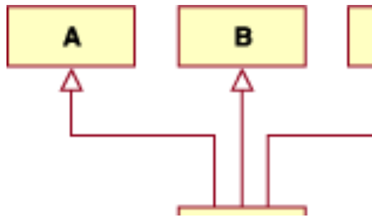
Multiple Inheritance

You can derive a class from any number of base classes. Deriving a class from more than one direct base class is called multiple inheritance. In the following example, classes A, B, and C are direct base classes for the derived class X:

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
```

class X : public A, private B, public C { **/* ... */** };

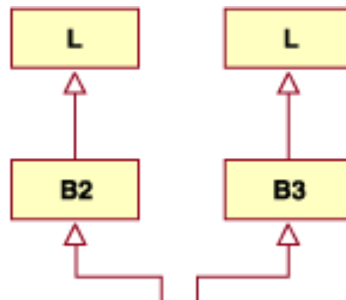
The following inheritance graph describes the inheritance relationships of the above example. An arrow points to the direct base class of the class at the tail of the arrow:



The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors. A direct base class cannot appear in the base list of a derived class more than once:

```
class B1 { /* ... */ };                    // direct base class
class D : public B1, private B1 { /* ... */ }; // error
```

However, a derived class can inherit an indirect base class more than once, as shown in the following example:




```
class L { /* ... */ };                  // indirect base class
class B2 : public L { /* ... */ };
class B3 : public L { /* ... */ };
```

class D : public B2, public B3 { /* ... */ }; // valid

In the above example, class D inherits the indirect base class L once through class B2 and once through class B3. However, this may lead to ambiguities because two subobjects of class L exist, and both are accessible through class D. You can avoid this ambiguity by referring to class L using a qualified class name. For example:

B2::L

or

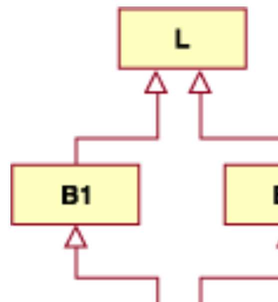
B3::L.

2.7.8 Virtual base classes

Suppose you have two derived classes B and C that have a common base class A, and you also have another class D that inherits from B and C. You can declare the base class A as virtual to ensure that B and C share the same subobject of A.

In the following example, an object of class D has two distinct subobjects of class L, one through class B1 and another through class B2. You can use the keyword virtual in front of the base class specifiers in the base lists of classes B1 and B2 to indicate that only one subobject of type L, shared by class B1 and class B2, exists.

For example:



class L { /* ... */ }; // indirect base class

class B1 : virtual public L { /* ... */ };

```
class B2 : virtual public L { /* ... */ };
class D : public B1, public B2 { /* ... */ }; // valid
```

Using the keyword virtual in this example ensures that an object of class D inherits only one subobject of class L.

In the above example, there are two derived classes B1 and B2 from the base class L. As shown in the above diagram, the D class is derived from both of the derived classes B1 and B2. In this scenario, if a user has a member function in the class D where the user wants to access the data or member functions of the class L it would result in error if it is performed like this:

```
class L
{
protected:
int x;
};
class B1:public L
{ };
class B2:public L
{ };
class D:public B1,public B2
{
public:
int example()
{
return x;
}
};
```

The above program results in a compile time error as the member function example() of class D tries to access member data x of class L. This results in an error because the derived classes B1 and B2 (derived from base class L) create copies of L called subobjects. This means that each of the subobjects have L member data and member functions and each have one copy of member data x. When the member function of the class D tries to access member data x, confusion arises as to which of the two copies it must access since it derived from both derived classes, resulting in a compile time error. When this occurs, **Virtual base class** is used. Both of the derived classes B1 and B2 are created as virtual base classes, meaning they should share a common subobject in their base class. For Example:

```
class L
{
protected:
int x;
;
class B1:virtual public L
{ };
class B2:virtual public L
{ };
class D:public B1,public B2
{
public:
int example()
{
return x;
}
```

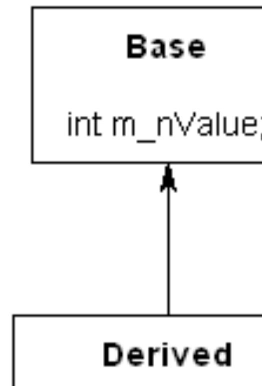
};

2.7.9 Constructors in Derived class

Consider the following example:

```
class Base
{
  public:
    int m_nValue;
    Base(int nValue)
       {  m_nValue = nValue;}
};
class Derived: public Base
 {
   public:
    double m_dValue;
    Derived(double dValue)
       { m_dValue = dValue;    }
};
```

In this example, Derived is derived from Base. Because Derived inherits functions and variables from Base, it is convenient to think of Derived as a two part class: one part Derived, and one part Base.



You've already seen plenty examples of what happens when we instantiate a normal (non-derived) class:

```
void main()
{
  Base cBase;
}
```

Base is a non-derived class because it does not inherit from anybody. C++ allocates memory for Base, then calls Base's default constructor to do the initialization.

Now let's take a look at what happens when we instantiate a derived class:

```
int main()
{
   Derived cDerived;
    return 0;
}
```

If you were to try this yourself, you wouldn't notice any difference from the previous example where we instantiate the non-derived class. But behind the scenes, things are slightly different. As mentioned, Derived is really two parts: a Base part, and a Derived part. When C++ constructs derived objects, it does so in

pieces, starting with the base portion of the class. Once that is complete, it then walks through the inheritance tree and constructs each derived portion of the class.

So what actually happens in this example is that the Base portion of Derived is constructed first. Once the Base portion is finished, the Derived portion is constructed. At this point, there are no more derived classes, so we are done. This process is actually easy to illustrate.

```
#include <iostream>
using namespace std;
class Base
{
   Base()
     {
             cout << "Base" << endl;
     }
};
class Derived: public Base
{
      Derived()
       {
               cout << "Derived" << endl;
       }
};
int main()
{
   cout << "Instantiating Base" << endl;
```

```
    Base cBase;
    cout << "Instantiating Derived" << endl;
    Derived cDerived;
    return 0;
}
```

This program produces the following result:

```
Instantiating Base
Base
Instantiating Derived
Base
Derived
```

As you can see, when we constructed Derived, the Base portion of Derived got constructed first. This makes sense: logically, a child cannot exist without a parent. It's also the safe way to do things: the child class often uses variables and functions from the parent, but the parent class knows nothing about the child. Instantiating the parent class first ensures those variables are already initialized by the time the derived class is created and ready to use them.

Order of construction for inheritance chains

It is sometimes the case that classes are derived from other classes, which are themselves derived from other classes. For example:

```
class A
{
public:
    A()
    {
        cout << "A" << endl;
```

```
    }
};
class B: public A
{
public:
    B()
    {
        cout << "B" << endl;
    }
};
class C: public B
{
public:
    C()
    {
        cout << "C" << endl;
    }
};
class D: public C
{
public:
    D()
    {
        cout << "D" << endl;
```

```
    }
};
```

Remember that C++ always constructs the "first" or "most base" class first. It then walks through the inheritance tree in order and constructs each successive derived class. Here's a short program that illustrates the order of creation all along the inheritance chain.

```
int main()
{
    cout << "Constructing A: " << endl;
    A cA;
    cout << "Constructing B: " << endl;
    B cB;

    cout << "Constructing C: " << endl;
    C cC;
    cout << "Constructing D: " << endl;
    D cD;
}
```

This code prints the following:

```
Constructing A:
A
Constructing B:
A
B
Constructing C:
```

A

B

C

Constructing D:

A

B

C

D

Destructors

When a derived class is destroyed, each destructor is called in the reverse order of construction. In the above example, when C class is destroyed, the C destructor is called first, then the B destructor, then the A destructor.

2.7.10 Types of Base Classes

A **direct base class** is a base class that appears directly as a base specifier in the declaration of its derived class.

An **indirect base class** is a base class that does not appear directly in the declaration of the derived class but is available to the derived class through one of its base classes. For a given class, all base classes that are not direct base classes are indirect base classes. The following example demonstrates direct and indirect base classes:

```
class A
{
 public:
  int x;
};
class B : public A
```

```
{
  public:
    int y;
};
class C : public B { };
```

Class B is a direct base class of C. Class A is a direct base class of B. Class A is an indirect base class of C. (Class C has x and y as its data members.)

Suppose you have two derived classes B and C that have a common base class A, and you also have another class D that inherits from B and C. You can declare the base class A as **virtual base class** to ensure that B and C share the same subobject of A.

2.7.11 Summary

Inheritance allows creating classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own. If class A inherits from class B, then B is called superclass of A. A is called subclass of B. Protected data members and functions are fully visible to derived classes, but are otherwise private. When you declare a derived class, an access specifier can precede each base class in the base list of the derived class. This does not alter the access attributes of the individual members of a base class as seen by the base class, but allows the derived class to restrict the access control of the members of a base class. The derived class can have a member function with same name, same return type and same list of arguments as defined in the base class. This is called function overriding. There can be different types of inheritance namely – single, multilevel and multiple inheritance. When an object of a derived class is created, it first calls the constructor of base class and then the derived class constructor is called. A base class can be a direct base class, an indirect base class or a virtual base class.

2.7.12 Short Answer Type Questions

1. What do you mean by a derived class?

2. What are protected data members?
3. What is function overriding?
4. What are the different kinds of base classes?

2.7.13 Long Answer Type Questions

1. What are the different types of derivation? Explain with examples.
2. Explain different kinds of inheritance.

2.7.14 Suggested Readings

1. Object Oriented Programming in C++ — Robert Lafore
2. The Complete Reference C++ — Herbert Schildt
3 Object Oriented Programming with C++ — E Balagurusamy

POINTERS

CONTENTS

Objectives

Introduction

2.8.1 Pointer Variables

2.8.2 Pointer Operators

2.8.3 Pointer Expressions

2.8.3.1 Pointer Assignments

2.8.3.2 Pointer Arithmetic

2.8.3.3 Pointer Comparisons

2.8.3.4 Pointers and Arrays

2.8.3.5 Multiple Indirection

2.8.4 Initializing Pointers

2.8.5 Pointers to Functions

2.8.6 Problems with Pointers

2.8.7 Summary

2.8.8 Self Understanding

2.8.9 Further Reading

Objective

After reading this lesson you should be able to:

Understand the general concept of pointer variables, operators and expressions

Understand the concept of multiple indirections

Understand the concept of pointers to functions

Understand how pinters and arrays are related

Introduction

A pointer is a variable that holds a memory address. This address is the location of another object (typically another variable) in memory. For example, if one variable contains the address of another variable, the first variable is said to point to the second. For a type T , T * is the type "pointer to T ." That is, a variable of type T * can hold the address of an object of type T . For example:

c h a r c = ´a ´;

c h a r * p = &c ; // p holds the address of c

The correct understanding and use of pointers is critical to successful C/C++ programming. There are three reasons for this: First, pointers provide the means by which functions can modify their calling arguments. Second, pointers support dynamic allocation. Third, pointers can improve the efficiency of certain routines. Pointers take on additional roles in C++. Pointers are one of the strongest but also one of the most dangerous features in C/C++. For example, uninitialized pointers (or pointers containing invalid values) can cause your system to crash. Perhaps worse, it is easy to use pointers incorrectly, causing bugs that are very difficult to find.

2.8.1 Pointer Variables

If a variable is going to hold a pointer, it must be declared as such. A pointer declaration consists of a base type, an *, and the variable name. The general form for declaring a pointer variable is

*type *name;*

where type is the base type of the pointer and may be any valid type. The name of the pointer variable is specified by name. The base type of the pointer defines what type of variables the pointer can point to. Technically, any type of pointer can point anywhere in memory. However, all pointer arithmetic is done relative to its base type, so it is important to declare the pointer correctly.

2.8.2 Pointer Operators

There are two special pointer operators: * and &.

- The & is a unary operator that returns the memory address of its operand.

For example,

m = &count;

places into m the memory address of the variable count. This address is the computer's internal location of the variable. It has nothing to do with the value of count. You can think of & as returning "the address of." Therefore, the preceding assignment statement means "m receives the address of count." To understand the above assignment better, assume that the variable count uses memory location 2000 to store its value. Also assume that count has a value of 100. Then, after the preceding assignment, m will have the value 2000.

- The second pointer operator, *, is the complement of &. It is a unary operator that returns the value located at the address that follows. For example, if m contains the memory address of the variable count,

q = *m;

places the value of count into q. Thus, q will have the value 100 because 100 is stored at location 2000, which is the memory address that was stored in m. You can think of * as "at address." In this case, the preceding statement means "q receives the value at address m."

Both & and * have a higher precedence than all other arithmetic operators except the unary minus, with which they are equal. You must make sure that your pointer variables always point to the correct type of data. For example, when you declare a pointer to be of type int, the compiler assumes that any address that it holds points to an integer variable—whether it actually does or not. Because C allows you to assign any address to a pointer variable, the following code fragment

compiles with no error messages (or only warnings, depending upon your compiler), but does not produce the desired result:

```
#include <stdio.h>
int main(void)
{
double x = 100.1, y;
int *p;
/* The next statement causes p (which is an
integer pointer) to point to a double. */
p = &x;
/* The next statement does not operate as
expected. */
y = *p;
printf("%f", y); /* won't output 100.1 */
return 0;
}
```

This will not assign the value of x to y. Because p is declared as an integer pointer, only 2 or 4 bytes of information will be transferred to y, not the 8 bytes that normally make up a double. In C++, it is illegal to convert one type of pointer into another without the use of an explicit type cast. For this reason, the preceding program will not even compile if you try to compile it as a C++ (rather than as a C) program. However, the type of error described can still occur in C++ in a more roundabout manner.

2.8.3 Pointer Expressions

In general, expressions involving pointers conform to the same rules as other expressions. This section examines a few special aspects of pointer expressions.

2.8.3.1 Pointer Assignments

As with any variable, you may use a pointer on the right-hand side of an assignment statement to assign its value to another pointer. For example,

```
#include <stdio.h>

int main(void)

{

int x;

int *p1, *p2;

p1 = &x;

p2 = p1;

printf(" %p", p2); /* print the address of x, not x's value! */

return 0;

}
```

Both p1 and p2 now point to x. The address of x is displayed by using the %p printf() format specifier, which causes printf() to display an address in the format used by the host computer.

2.8.3.2 Pointer Arithmetic

There are only two arithmetic operations that you may use on pointers: addition and subtraction. To understand what occurs in pointer arithmetic, let p1 be an integer pointer with a current value of 2000. Also, assume integers are 2 bytes long. After the expression

```
p1++;
```

p1 contains 2002, not 2001. The reason for this is that each time p1 is incremented, it will point to the next integer. The same is true of decrements. For example, assuming that p1 has the value 2000, the expression

```
p1--;
```

causes p1 to have the value 1998.

The following rules govern pointer arithmetic

- Each time a pointer is incremented, it points to the memory location of the next element of its base type.
- Each time it is decremented, it points to the location of the previous element. When applied to character pointers, this will appear as "normal" arithmetic because characters are always 1 byte long.
- All other pointers will increase or decrease by the length of the data type they point to. This approach ensures that a pointer is always pointing to an appropriate element of its base type. You are not limited to the increment and decrement operators. For example, you may add or subtract integers to or from pointers. The expression

  ```
  p1 = p1 + 12;
  ```

  Makes p1 point to the twelfth element of p1's type beyond the one it currently points to.
- Besides addition and subtraction of a pointer and an integer, only one other arithmetic operation is allowed: You may subtract one pointer from another in order to find the number of objects of their base type that separate the two.
- All other arithmetic operations are prohibited. Specifically, you may not multiply or divide pointers; you may not add two pointers; you may not aypply the bitwise operators to them; and you may not add or subtract type float or double to or from pointers.

2.8.3.3 Pointer Comparisons

You can compare two pointers in a relational expression. For instance, given two pointers p and q, the following statement is perfectly valid:

```
if(p<q) printf("p points to lower memory than q\n");
```

Generally, pointer comparisons are used when two or more pointers point to a common object, such as an array. Consider the following example, a pair of stack routines are developed that store and retrieve integer values. A stack is a list that uses first-in, last-out accessing. It is often compared to a stack of plates on a table—the first one set down is the last one to be used. Stacks are used frequently in compilers, interpreters, spreadsheets, and other system-related software. To

create a stack, you need two functions: push() and pop() . The push() function places values on the stack and pop() takes them off. These routines are shown here with a simple main() function to drive them. The program puts the values you enter into the stack. If you enter 0, a value is popped from the stack. To stop the program, enter -1.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 60
void push(int i);
int pop(void);
int *tos, *p1, stack[SIZE];

int main(void)
{
int value;
tos = stack; /* tos points to the top of stack */
p1 = stack; /* initialize p1 */
do {
printf("Enter value: ");
scanf("%d", &value);
if(value!=0) push(value);
else printf("value on top is %d\n", pop());
} while(value!=-1);
return 0;
}

void push(int i)
```

```
{
p1++;
if(p1==(tos+SIZE)) {
printf("Stack Overflow.\n");
exit(1);
}
*p1 = i;
}
int pop(void)
{
if(p1==tos) {
printf("Stack Underflow.\n");
exit(1);
}
p1--;
return *(p1+1);
}
```

You can see that memory for the stack is provided by the array stack. The pointer p1 is set to point to the first element in stack. The p1 variable accesses the stack. The variable tos holds the memory address of the top of the stack. It is used to prevent stack overflows and underflows. Once the stack has been initialized, push() and pop() may be used. Both the push() and pop() functions perform a relational test on the pointer p1 to detect limit errors. In push() , p1 is tested against the end of stack by adding SIZE (the size of the stack) to tos. This prevents an overflow. In pop() , p1 is checked against tos to be sure that a stack underflow has not occurred. In pop() , the parentheses are necessary in the return statement. Without them, the statement would look like this:

```
return *p1 +1;
```

which would return the value at location p1 plus one, not the value of the location p1+1.

2.8.3.4 Pointers and Arrays

There is a close relationship between pointers and arrays. Consider this program fragment:

```
char str[80], *p1;
p1 = str;
```

Here, p1 has been set to the address of the first array element in str. To access the fifth element in str, you could write

```
str[4]
```

or

```
*(p1+4)
```

Both statements will return the fifth element. Remember, arrays start at 0. To access the fifth element, you must use 4 to index str. You also add 4 to the pointer p1 to access the fifth element because p1 currently points to the first element of str.

Thus C/C++ provides two methods of accessing array elements: pointer arithmetic and array indexing. Although the standard array-indexing notation is sometimes easier to understand, pointer arithmetic can be faster. Since speed is often a consideration in programming, C/C++ programmers commonly use pointers to access array elements. These two versions of putstr()— one with array indexing and one with pointers— illustrate how you can use pointers in place of array indexing. The putstr() function writes a string to the standard output device one character at a time.

```
/* Index s as an array. */
void putstr(char *s)
{
register int t;
for(t=0; s[t]; ++t) putchar(s[t]);
```

```
}
/* Access s as a pointer. */
void putstr(char *s)
{
while(*s) putchar(*s++);
}
```

Most professional C/C++ programmers would find the second version easier to read and understand. In fact, the pointer version is the way routines of this sort are commonly written in C/C++.

Pointers may be arrayed like any other data type. The declaration for an int pointer array of size 10 is

```
int *x[10];
```

To assign the address of an integer variable called var to the third element of the pointer array, write

```
x[2] = &var;
```

To find the value of var, write

```
*x[2]
```

If you want to pass an array of pointers into a function, you can use the same method that you use to pass other arrays—simply call the function with the array name without any indexes. For example, a function that can receive array x looks like this:

```
void display_array(int *q[])
{
int t;
for(t=0; t<10; t++)
printf("%d ", *q[t]);
}
```

Remember, q is not a pointer to integers, but rather a pointer to an array of pointers to integers. Therefore you need to declare the parameter q as an array of integer pointers, as just shown. You cannot declare q simply as an integer pointer because that is not what it is. Pointer arrays are often used to hold pointers to strings. You can create a function that outputs an error message given its code number, as shown here:

```
void syntax_error(int num)
{
static char *err[] = {
"Cannot Open File\n",
"Read Error\n",
"Write Error\n",
"Media Failure\n"
};
printf("%s", err[num]);
}
```

The array err holds pointers to each string. As you can see, printf() inside syntax_error() is called with a character pointer that points to one of the various error messages indexed by the error number passed to the function. For example, if num is passed a 2, the message Write Error is displayed. As a point of interest, note that the command line argument argv is an array of character pointers.

2.8.3.5 Multiple Indirection

You can have a pointer point to another pointer that points to the target value. This situation is called multiple indirection, or pointers to pointers. The value of a normal pointer is the address of the object that contains the value desired. In the case of a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the object that contains the value desired. Multiple indirection can be carried on to whatever extent rquired, but more than a pointer to a pointer is rarely needed. In fact, excessive indirection is difficult to follow and prone to conceptual errors. Do not confuse multiple indirection with high-level

data structures, such as linked lists, that use pointers. These are two fundamentally different concepts. A variable that is a pointer to a pointer must be declared as such. You do this by placing an additional asterisk in front of the variable name. For example, the following declaration tells the compiler that newbalance is a pointer to a pointer of type float:

```
float **newbalance;
```

You should understand that newbalance is not a pointer to a floating-point number but rather a pointer to a float pointer.

To access the target value indirectly pointed to by a pointer to a pointer, you must apply the asterisk operator twice, as in this example:

```
#include <stdio.h>
int main(void)
{
int x, *p, **q;
x = 10;
p = &x;
q = &p;
printf("%d", **q); /* print the value of x */
return 0;
}
```

Here, p is declared as a pointer to an integer and q as a pointer to a pointer to an integer. The call to printf() prints the number 10 on the screen.

2.8.4 Initializing Pointers

After a local pointer is declared but before it has been assigned a value, it contains an unknown value. (Global pointers are automatically initialized to null.) Should you try to use the pointer before giving it a valid value, you will probably crash your program—and possibly your computer's operating system as well. There is an important convention that most C/C++ programmers follow when working with pointers: A pointer that does not currently point to a valid memory location is

given the value null (which is zero). By convention, any pointer that is null implies that it points to nothing and should not be used. The use of null is simply a convention that programmers follow. It is not a rule enforced by the C or C++ languages. You can use the null pointer to make many of your pointer routines easier to code and more efficient. For example, you could use a null pointer to mark the end of a pointer array. Another variation on the initialization theme is the following type of string declaration:

```
char *p = "hello world";
```

As you can see, the pointer p is not an array. The reason this sort of initialization works is because of the way the compiler operates. All C/C++ compilers create what is called a *string table*, which is used to store the string constants used by the program. Therefore, the preceding declaration statement places the address of hello world, as stored in the string table, into the pointer p. Throughout a program, p can be used like any other string. For example, the following program is perfectly valid:

```
#include <stdio.h>
#include <string.h>
char *p = "hello world";
int main(void)
{
register int t;
/* print the string forward and backwards */
printf(p);
for(t=strlen(p)-1; t>-1; t--) printf("%c", p[t]);
return 0;
}
```

In Standard C++, the type of a string literal is technically const char *. But C++ provides an automatic conversion to char *. Thus, the preceding program is still valid. However, this automatic conversion is a deprecated feature, which means that you should not rely upon it for new code. For new programs, you should

assume that string literals are constants and the declaration of p in the preceding program should be written like this.

```
const char *p = "hello world";
```

2.8.5 Pointers to Functions

A particularly confusing yet powerful feature of C++ is the *function pointer*. Even though a function is not a variable, it still has a physical location in memory that can be assigned to a pointer. This address is the entry point of the function and it is the address used when the function is called. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions. You obtain the address of a function by using the function's name without any parentheses or arguments. (This is similar to the way an array's address is obtained when only the array name, without indexes, is used.) To see how this is done, study the following program, paying close attention to the declarations:

```
#include <stdio.h>
#include <string.h>
void check(char *a, char *b,
int (*cmp)(const char *, const char *));
int main(void)
{
char s1[80], s2[80];
int (*p)(const char *, const char *);
p = strcmp;
gets(s1);
gets(s2);
check(s1, s2, p);
return 0;
}
```

```
void check(char *a, char *b,
int (*cmp)(const char *, const char *))
{
printf("Testing for equality.\n");
if(!(*cmp)(a, b)) printf("Equal");
else printf("Not Equal");
}
```

When the check() function is called, two character pointers and one function pointer are passed as parameters. Inside the function check() , the arguments are declared as character pointers and a function pointer. Notice how the function pointer is declared. You must use a similar form when declaring other function pointers, although the return type and parameters of the function may differ. The parentheses around the *cmp are necessary for the compiler to interpret this statement correctly. Inside check() , the expression

```
(*cmp)(a, b)
```

calls strcmp() , which is pointed to by cmp, with the arguments a and b. The parentheses around *cmp are necessary. This is one way to call a function through a pointer. A second, simpler syntax, as shown here, may also be used.

```
cmp(a, b);
```

The reason that you will frequently see the first style is that it tips off anyone reading your code that a function is being called through a pointer. (That is, that cmp is a function pointer, not the name of a function.) Other than that, the two expressions are equivalent. Note that you can call check() by using strcmp() directly, as shown here:

```
check(s1, s2, strcmp);
```

This eliminates the need for an additional pointer variable. You may wonder why anyone would write a program in this way. Obviously, nothing is gained and significant confusion is introduced in the previous example. However, at times it is advantageous to pass functions as parameters or to create an array of functions. For example, when a compiler or interpreter is written, the parser (the part that

evaluates expressions) often calls various support functions, such as those that compute mathematical operations (sine, cosine, tangent, etc.), perform I/O, or access system resources. Instead of having a large switch statement with all of these functions listed in it, an array of function pointers can be created. In this approach, the proper function is selected by its index. You can get the flavor of this type of usage by studying the expanded version of the previous example. In this program, check() can be made to check for either alphabetical equality or numeric equality by simply calling it with a different comparison function.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
void check(char *a, char *b,
int (*cmp)(const char *, const char *));
int numcmp(const char *a, const char *b);
int main(void)
{
   char s1[80], s2[80];
   gets(s1);
   gets(s2);
   if(isalpha(*s1))
   check(s1, s2, strcmp);
   else
   check(s1, s2, numcmp);
   return 0;
}
void check(char *a, char *b,
```

```
int (*cmp)(const char *, const char *))
{
  printf("Testing for equality.\n");
  if(!(*cmp)(a, b)) printf("Equal");
  else printf("Not Equal");
}
int numcmp(const char *a, const char *b)
{
   if(atoi(a)==atoi(b)) return 0;
   else return 1;
}
```

In this program, if you enter a letter, strcmp() is passed to check() . Otherwise, numcmp() is used. Since check() calls the function that it is passed, it can use different comparison functions in different cases.

2.8.6 Problems with Pointers

Pointers give you tremendous power and are necessary for many programs. At the same time, when a pointer accidentally contains a wrong value, it can be the most difficult bug to find. An erroneous pointer is difficult to find because the pointer itself is not the problem. The problem is that each time you perform an operation using the bad pointer, you are reading or writing to some unknown piece of memory. If you read from it, the worst that can happen is that you get garbage. However, if you write to it, you might be writing over other pieces of your code or data. This may not show up until later in the execution of your program, and may lead you to look for the bug in the wrong place. There may be little or no evidence to suggest that the pointer is the original cause of the problem. To help you avoid them, a few of the more common errors are discussed here.

- The classic example of a pointer error is the *uninitialized pointer*. Consider this program.

```
/* This program is wrong. */
```

```
int main(void)
{
    int x, *p;
    x = 10;
   *p = x;
   return 0;
}
```

This program assigns the value 10 to some unknown memory location. Here is why: Since the pointer p has never been given a value, it contains an unknown value when the assignment *p = x takes place. This causes the value of x to be written to some unknown memory location. This type of problem often goes unnoticed when your program is small because the odds are in favor of p containing a "safe" address—one that is not in your code, data area, or operating system. However, as your program grows, the probability increases of p pointing to something vital. Eventually, your program stops working. The solution is to always make sure that a pointer is pointing at something valid before it is used.

- A second common error is caused by a simple misunderstanding of how to use a pointer. Consider the following:

```
/* This program is wrong. */
#include <stdio.h>
int main(void)
{
int x, *p;
x = 10;
p = x;
printf("%d", *p);
return 0;
```

```
}
```

The call to printf() does not print the value of x, which is 10, on the screen. It prints some unknown value because the assignment p = x; is wrong. That statement assigns the value 10 to the pointer p. However, p is supposed to contain an address, not a value. To correct the program, write p = &x;

- Another error that sometimes occurs is caused by incorrect assumptions about the placement of variables in memory. You can never know where your data will be placed in memory, or if it will be placed there the same way again, or whether each compiler will treat it in the same way. For these reasons, making any comparisons between pointers that do not point to a common object may yield unexpected results.

  For example,

```
char s[80], y[80];
char *p1, *p2;
p1 = s;
p2 = y;
if(p1 < p2) . . .
```

  is generally an invalid concept. (In very unusual situations, you might use something like this to determine the relative position of the variables. But this would be rare.)

- A related error results when you assume that two adjacent arrays may be indexed as one by simply incrementing a pointer across the array boundaries. For example,

```
int first[10], second[10];
int *p, t;
p = first;
for(t=0; t<20; ++t) *p++ = t;
```

  This is not a good way to initialize the arrays first and second with the numbers 0 through 19. Even though it may work on some compilers under

certain circumstances, it assumes that both arrays will be placed back to back in memory with first first. This may not always be the case.

2.8.7 Summary

- A pointer is a variable that holds a memory address.
- The correct understanding and use of pointers is critical to successful C/C++ programming.
- Pointers can improve the efficiency of certain routines.
- There are two special pointer operators: * and &.
- There are only two arithmetic operations that you may use on pointers: addition and subtraction.
- You can compare two pointers in a relational expression.
- C/C++ provides two methods of accessing array elements: pointer arithmetic and array indexing.
- You can have a pointer point to another pointer that points to the target value. This situation is called multiple indirection
- Excessive indirection is difficult to follow and prone to conceptual errors
- By convention, any pointer that is null implies that it points to nothing and should not be used
- Even though a function is not a variable, it still has a physical location in memory that can be assigned to a pointer

2.8.8 Self Understanding

1. Why is the use of pointer critical for C++ Programming, explain?
2. How many methods of accessing array elements are there in C++, illustrate with the help of a program
3. Which arithmatic operations are allowed in pointers?
4. What are the main problems in the use of pointers?

2.8.9 Further Reading

1. Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.
2. Robert Lafore, “Object Oriented Programming in C++”, Galgotia Publications,1994.

3. E. Balaguruswamy, “Object Oriented Programming with C++”, Tata McGraw-Hill,2001. Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.

4. Deitel and Deitel, “ C++ How to program” ,Pearson Education, 2001.

5. Bjarne Straustrup, “ The C++ Programming Language”, Addison-Wesley Publication Co.,2001.

6. Bruce Eckel, “ Thinking In C++” Second Edition, Prentice Hall.

7. Yashavant P. Kanetkar,"Let Us C", Fifth Edition, BPB publications

B.A. PART-III	PAPER : BAP-301
SEMESTER-V	OBJECT ORIENTED PROGRAMING USING C++
LESSON NO. 2.9	AUTHOR : AMANDEEP KAUR

STORAGE CLASSES

2.9.0 Introduction

2.9.1 Storage Classes

- **2.9.1.1 auto**
- **2.9.1.2 static**
- **2.9.1.3 register**
- **2.9.1.4 extern**
- **2.9.1.5 mutable**

2.9.2 Summary

2.9.3 Further Reading

2.9.0 Introduction :

The area or scope of the variable depends on its storage class i.e. where and how it is declared. There are four scope variable i.e. function, file, block and function prototype. The storage class of a variable tells the compiler. Every variable in C++ possesses a characteristics called its storage class. The storage class define two characteristics of the variable :

- Life Time : Lifetime of a variable is the length of time it retains a particular value.

- Visibility : Visibility of a variable refers to those parts of a program that will be able to recognize it.

The correct use of storage class is important in large programs. The storage class precedes the variable declaration and tell the compiler how the variable should be stored.

2.9.1 Storage Classes

Program variables have a storage class in addition to a data type. Storage classes are important in C++ for several reasons. One, they tell the compiler how to create and release variables and where to place them in the run-time environment (stack, data area, or CPU registers). Two, storage class specifiers affect the initial values and the scope of variables. This section reviews the storage class specifiers **auto, static, register, extern, and mutable**.

2.9.1.1 auto

The default storage class specifier **auto** (short for automatic) allocates memory for variables from the run-time stack. The definition

```
auto int num;
```

for example, compiles in C++, although the keyword **auto** is optional and seldom used. Automatic definitions appear inside functions and blocks and have undefined initial values. Furthermore, their scope applies only to the block in which they are declared. The following function, for example, creates four automatic variables, all of different data types.

```
void subr() {
 int i = 5;          // auto integer
 float f = 34.56;       // auto float
 char buf[80];        // auto character array
 struct complex {
   float imag, real;
 } val;            // auto structure
```

```
 i++;            // increment i
 f++;            // increment f
 . . .
}
```

The compiler allocates stack memory for each auto variable every time a program calls **subr().** Likewise, the compiler releases this stack memory when the function returns. Even though **subr()** increments **i** and **f**, they continue to receive their initial values every time we call the function. This implies that automatic variables do not retain their values between function calls or after exiting a block and reentering it again.

Storage duration of automatic variables

Objects with the **auto** storage class specifier have automatic storage duration. Each time a block is entered, storage for **auto** objects defined in that block is made available. When the block is exited, the objects are no longer available for use. An object declared with no linkage specification and without the **static** storage class specifier has automatic storage duration. If an **auto** object is defined within a function that is recursively invoked, memory is allocated for the object at each invocation of the block.

Linkage of automatic variables

An **auto** variable has block scope and no linkage.

NOTE: Always initialize auto variables before you use them. Although many C++ compilers issue warnings if you don't, the following example is a common error.

```
int count(int a[], int max, int val)
{
  int cnt;              // auto count variable
  for (int i = 0; i < max; i++)    // loop over array
    if (a[i] == val)        // found value?
```

```
    cnt++;          // increment count
  return cnt;          // return count
}
```

The **count()** function loops over an array of integers and returns the number of elements that are equal to a certain value. The **cnt** variable, however, does not have an initial value (is it 0 or something else?). The behavior of this function is not predictable (it may work fine or return a random value).

2.9.1.2 static

The storage class specifier **static** allocates memory from the data area. The definition

```
static int num;
```

makes the variable **num** reside in the data area and retain its value throughout program execution. Variables that you declare **static** receive their initial values (0 if you don't provide one) before execution of **main()**. C++ has two types of static variables. A variable is internal static (the first type) if its definition appears inside a block with the keyword **static** and a variable is external static(the second type) if its definition appears outside any block with the keyword **static**.

Linkage of static variables

A declaration of an object that contains the **static** storage class specifier has file scope and thus gives the identifier internal linkage. Each instance of the particular identifier therefore represents the same object within one file only. For example, if a static variable **x** has been declared in function **f**, when the program exits the scope of **f**, **x** is not destroyed:

```
#include <stdio.h>
int f(void)
{
  static int x = 0;
  x++;
```

```
  return x;
}
int main()
 {
  int j;
  for (j = 0; j < 5; j++)
 {
    printf("Value of f(): %d\n", f());
  }
  return 0;
}
```

The following is the output of the above example:

```
Value of f(): 1
Value of f(): 2
Value of f(): 3
Value of f(): 4
Value of f(): 5
```

Because x is a static variable, it is not reinitialized to 0 on successive calls to f.

The second type of **static** variable (called external static) applies to separately compiled modules (files of function and variable definitions). External **static** definitions appear outside blocks with the keyword **static.** The external static are similar to internal static with the difference that external static can be accessed by all functions in the file while internal static are known only to the block in which they are declared. Here are several examples.

```
static char coal;     // file scope - only this module may access it
```

```
long fellow;       // program scope - any module may access it
void f()          // program scope - any module may call it
{
 . . .
}
static void g()      // file scope - only this module may call it
{
 . . .
}
```

The variable fellow and function f() have program scope, which means any module may call f() or access fellow (this makes them global). The character coal and function g(), on the other hand, have file scope (modules outside this file cannot call g() or access coal).

Note that a variable defined outside a function lives in the data area and the word static affects only its linkage. Inside a function, the word static does not affect a variable's scope, only its storage class.

NOTE: Use unnamed namespaces instead of external static to create variables with file scope. External static is deprecated in ANSI C++.

2.9.1.3 register

The storage class specifier **register** stores a variable's data in a hardware CPU register (if available) instead of memory. For example, the definition

```
register int num;
```

uses a register for the integer **num**. Only nonstatic, local variables may reside in registers, and C++ uses the same rules for register variable scope and initialization as it does with automatic variables. You cannot take the address of a register with **&**, use **static** with registers, or declare register variables outside of functions.

NOTE: Register variables are highly machine and compiler dependent. Many compilers, for instance, allocate registers for only pointers, int, and char data types. Furthermore, your compiler may choose to ignore all of your register declarations or give you a register even if you don't ask for one! Consult your compiler's documentation to see how to use register variables effectively.

Why use register variables? In time-critical code, register variables can improve a program's performance. Arithmetic and array subscripting operations inside loops usually execute faster with register variables than with auto or static variables. Loop variables, pointers, and function parameters are also suitable candidates for register variables. Registers are a limited resource, so you'll want to allocate them carefully. When the compiler runs out of hardware CPU registers, variables that you declare register become automatic.

The following code, for example, uses a register variable to loop through a large array if it's time to process data.

```
if (process)
{
 for (register int i = 0; i < huge; i++)
   . . . a[i] . . .
}
```

Inside the **for** loop, the program declares **i** as a **register** variable before it loops through the array. If a register is available, the loop executes faster than it would without one.

What should you do when there are not enough registers? If this situation arises, declare your registers outside of loops and declare the most important register variables first. This approach makes the least important variables become local variables if the compiler cannot provide registers.

2.9.1.4 extern

The storage class specifier **extern** allows modules to access global variables (program scope) and non-static functions defined in another module. The declaration

```
extern int num;
```

for example, makes the integer **num** (defined elsewhere) accessible in the module that contains this declaration. The compiler does not allocate memory with **extern** declarations, and you must supply a data type after **extern**.

Suppose, for example, **file mod1.C** calls a function whose definition appears in **file mod2.C**, which, in turn, accesses variables defined in **mod1.C**. Here's the code for **mod1.C**.

Listing 3.11 Defining external variables

```
// mod1.C
#include <iostream.h>
int nitems;
double servo[100];
int s;
double f();
int main()
{
 . . .
 servo[5] = f();
 . . .
}
```

Module **mod1.C** calls a function **f()** and defines an integer (**nitems**), an array of 100 doubles (**servo**), and a structure (**s**),. These variables will be available to mod2.C because we do not define them as **static**.

Here's the code for **mod2.C**.

//Declaring external variables // mod2.C

```
#include <iostream.h>
 extern int nitems;
extern double servo[];
double f()
{
 extern int s;
 . . .
}
```

Module mod2.C uses **extern** to declare **nitems, servo,** and **s** (defined in **mod1.C**). You may place **extern** statements anywhere a declaration is legal, as long as the **extern** declaration appears before you use the variable in a statement or expression. Note that the compiler ignores the size in an **extern** array declaration (such as **servo[100]**).

Likewise, the **extern** specifier in function prototypes is optional, since it is the default storage class. C++ allows

```
extern double f();
```

in **mod1.C**, although it's not necessary. Without the keyword **static**, all function declarations are external by default.

2.9.1.5 mutable

The **mutable** storage class specifier is used only on a class data member to make it modifiable even though the member is part of an object declared as **const.** You cannot use the mutable specifier with names declared as **static or const**, or reference members.

In the following example:

```
class A
{
```

```
  public:
    A() : x(4), y(5) { };
    mutable int x;
    int y;
};
int main()
{
  const A var2;
  var2.x = 345;
  // var2.y = 2345;
}
```

the compiler would not allow the assignment **var2.y = 2345** because **var2** has been declared as **const**. The compiler will allow the assignment **var2.x = 345** because **A::x** has been declared as **mutable.**

2.9.2 Summary

A program can consists of only main function or any number of function out of which one must be main function. All the functions that make up the complete program can be stored in one file. Storage class determines the lifetime and visibility of the variables. It also determines that where the variables will be stored.

2.9.3 Further Reading :

Ashok N.Kamthane, 'Programming With ANSI and Turbo C', Pearson Education

R.S. Salaria, 'Object Oriented Programming Using C++', 4th Edition, Khanna Book Publishing Co., Delhi

Type Setting :

Department of Distance Education, Punjabi University, Patiala