UNIT 1 OBJECT ORIENTED PROGRAMMING FUNDAMENTALS

DIFFERENCE BETWEEN C & C++

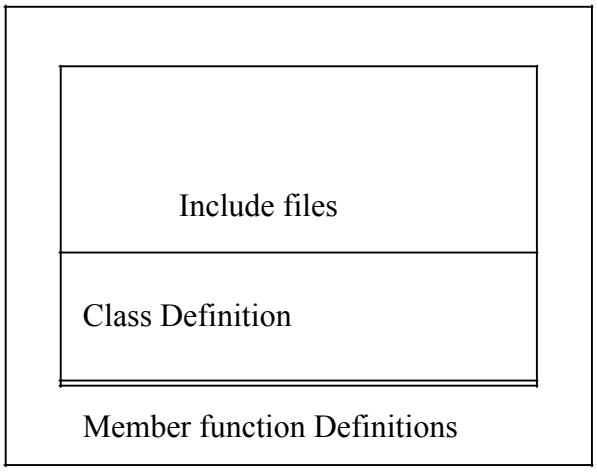
Procedure Oriented Programming	Object Oriented Programming.
The problem is viewed as a sequence of things to be done.	Emphasis on data rather than procedure.
Larger programs are divided into smaller programs known as functions.	Larger programs are divided into objects.
The data is defined as global and accessible to all the functions of a program without any restrictions which reduces data security and integrity.	It ties data more closely to the function that operates on it and protects it from accidental modification from outside.
Employs top-down approach	Employs bottom-up approach

INTRODUCTION TO C++

Basics of C++ Programming

C++ was developed by BJARNE STROUSSTRUP at AT&T BELL Laboratories in Murry Hill, USA in early 1980‘s. STROUSSTRUP combines the features of _C‘ language and _SIMULA67‘ to create more powerful language that support OOPS concepts, and that language was named as —C with CLASSES‖. In late 1983, the name got changed to C++. The idea of C++ comes from _C‘ language increment operator (++) means more additions. C++, but the object oriented features (Classes, Inheritance, Polymorphism, Overloading) makes the C++ truly as Object Oriented Programming language.

STRUCTURE OF C++ PROGRAM



Include files provides instructions to the compiler to link functions from the system library.

Eg: #include <iostream.h>

#include – Preprocessor Directive

iostream.h – Header File

A class is a way to bind and its associated functions together. It is a user defined datatype. It must be declared at class declaration part.

Member function definition describes how the class functions are implemented. This must be the next part of the C++ program.

Finally main program part, which begins program execution.

```
main( )
{

}
```

Program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical and of the program.

Input / Output statements

Input Stream

Syntax:

cin >> var1 >> var2 >>;

cin – Keyword, it is an object, predefined in C++ to correspond to the standard input stream.

>> - is the extraction or get from operator

Extraction operation (>>) takes the value from the stream object on its left and places it in the variable on its right.

Eg:

```
cin>>x;
cin>>a>>b>>c;
```

Output Stream:

Syntax:

cout<<var1<<var2;

cout - object of standard output stream

<< - is called the insertion or put to operator

It directs the contents of the variable on its right to the object on its left. Output stream can be used to display messages on output screen.

Eg:

cout<<a<<b; cout<<‖value of x is‖<<x;

cout<<‖Value of x is‖<<x<<‖less than‖<<y;

TOKENS

The smallest individual units in a program are known as tokens. C++ has the following tokens

- Keywords
- Identifiers
- Constants
- Strings
- Operators

Keywords

It has a predefined meaning and cannot be changed by the user

Keywords cannot be used as names for the program variables. Keywords supported by C++ are:

asm	double	new	switch	
auto	else		operator	template
break	enum		private	this
case	extern		protected	throw
catch	float		public	try
char	for	register	typedef	
class	friend	return	union	
const	goto	short	unsigned	

Identifiers

Identifiers refer to the names of variables, functions, arrays, classes, etc. created by the programer.

Rules for naming these identifiers:

Only alphabetic characters, digits and underscores are permitted.

The name cannot start with a digit.

Uppercase and lowercase letters are distinct.

A declared keyword cannot be used as a variable name.

(i) Variables: It is an entity whose value can be changed during program execution and is known to the program by a name.

A variable can hold only one value at a time during program execution.A variable is the storage location in memory that is stored by its value. A variable is identified or denoted by a variable name.

The variable name is a sequence of one or more letters, digits or underscore.

Rules for defining variable name:

A variable name can have one or more letters or digits or underscore for example character White space, punctuation symbols or other characters are not permitted to denote variable name.A variable name must begin with a letter.

Variable names cannot be keywords or any reserved words of the C++ programming language. C++ is a case-sensitive language.

Variable names written in capital letters differ from variable names with the same name but written in small letters.

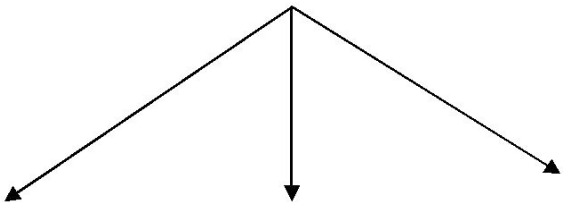
Declaration of Variables

Syntax :datatype variablename;

Datatype

It is the type of data, that is going to be processed within the program

C++ datatype



User Defined type	Built-in type	Derived type
Eg:	Eg:	Eg:
structure	Void	Array
union	int	Function
Enumerated	float	Pointers
	double	
	char	

constant

A quantity that does not change is known as constants. integer constants are represented as decimal notation.

Decimal notation is represented with a number.

Octal notation is represented with the number preceded by a zero(0) character.

A hexadecimal number is preceded with the characters 0x.

Types of constants:

- Integer constants - Eg: 123, 25 – without decimal point
- Character constants - Eg: _A‘, _B‘, _*‘, _1‘
- Real constants - Eg: 12.3, 2.5 - with decimal point

Strings

A sequence of characters is called string. String constants are enclosed in double quotes as follows

―Hello‖

OPERATORS

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.

Types of Operators

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment & decrement Operators
6. Conditional Operators
7. Bitwise Operators
8. Special Operators
9. Manipulators
10. Memory allocate / delete Operators

An expression is a combination of variables, constants and operators written according to the syntax of the language.

Arithmetic Operators

- C++ has both unary & binary arithmetic operators.

- Unary operators are those, which operate on a single operand.
- Whereas, binary operators on two operands +, -, *, /, %

Examples for Unary Operators:

1. int x = 10;

y = -x;

int x = 5; sum = -x;

Examples for Unary Operators:

int x = 16, y=5;

x+y = 21; (result of the arithmetic expression) x-y = 11;

x*y=80;

/ - Division Operator

Eg:

x = 10, y = 3;

x/y=3; (The result is truncated, the decimal part is discarded.)

% - Modulo Division

The result is the remainder of the integer division only applicable for integer values. x=11, y = 2

x%y = 1

Relational Operators

- A relational operator is used to make comparison between two expressions.
- All relational operators are binary and require two operands.

<, <=, >, >=, ++, !=

Relational Expression

Expression1 (relational operator) Expression2

Expression1 & 2 – may be either constants or variables or arithmetic expression. Eg:

a < b (Compares its left hand side operand with its right hand side operand) 10 = = 15(Equals)

a != b(Not equals)

An relational expression is always return either zero or 1, after evaluation.

Eg: (a+b) <= (c+d)

arithmetic expression

Logical Operators

&& - Logical AND

- Logical OR

Increment & Decrement Operators

- Increment ++, this operator adds 1 to the operand
- Decrement --, this operator subtracts 1 from the operand
- Both are unary operators

Eg:

m = 5;

y = ++m; (adds 1 to m value)

x = --m; (Subtracts 1 from the value of m)

Types

- Pre Increment / Decrement OP:
- Post Increment / Decrement OP:

If the operator precedes the operand, it is called pre increment or pre decrement.

Eg: ++i, --i;

If the operator follows the operand, it is called post increment or post decrement.

Eg: i++, i--;

In the pre Increment / Decrement the operand will be altered in value before it is utilized for its purpose within the program.

Eg: x = 10;

Y = ++x;

1st x value is getting incremented with 1.

Then the incremented value is assigned to y.

In the post Increment / Decrement the value of the operand will be altered after it is utilized.

Eg: y = 11;

x = y++;

1st x value is getting assigned to x & then the value of y is getting increased.

Conditional Operator? :

General Form is

Conditional exp ? exp 1 : exp 2;

Conditional exp - either relational or logical expression is used.

Exp1 & exp 2 : are may be either a variable or any statement.

Eg:

(a>b)?a:b;

Conditional expression is evaluated first.

If the result is _1' is true, then expression1 is evaluated.

If the result is zero, then expression2 is evaluated.

Special Operators

sizeof

comma(,)

size of operators returns the size the variable occupied from system memory.

Eg:

var = sizeof(int)

cout<<var; Ans: 2

x = size of (float);

cout << x; Ans: 4

int y;

Manipulators

Manipulators are operators used to format the data display. The commonly used manipulators are endl, setw.

endl manipulators

It is used in output statement causes a line feed to be inserted, it has the same effect as using the newline character —\n‖ in _C' language.

```
#include <iostream.h>
main()
{
int a=10, b=20;
cout << —C++ language‖ << endl; cout << —A value: — << a << endl; cout << —B value:‖ << b << endl;
}
```

O/P:

C++ language

A value: 10

B value: 20

Setw Manipulator

The setw manipulator is used or specify the field width for printing, the content of the variable.

Syntax: setw(width);

where width specifies the field width.

int a = 10;

cout << — A value‖ << setw(5) << a << endl;

Output:

A value 10

REFERENCE VARIABLE

A reference variable provides an alias (alternative name) for a previously defined variable.

For example, if we make the variable sum a reference to the variable total, then sum & total can be used interchangeably to represent that variable.

A reference variable is created as follows:

datatype & ref_name = var_name;

Eg:

Float total = 100; float & sum = total; cout << sum << total;

Both the variables refer to the same data object in the memory i.e. total, sum 100

CLASS SCOPE AND ACCESSING CLASS MEMBERS

A **class definition** starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations

General Syntax of a class:

General structure for defining a class is:

```
class classname
{
  acess specifier:
  data member;
  member functions;
acess specifier:
data member;
member functions;
};
```

Generally, in class, all members (data) would be declared as private and the member functions would be declared as public. Private is the default access level for specifiers. If no access specifiers are identified for members of a class, the members are defaulted to private access.

example

```
class Box
{
  public:
    double length;   // Length of a box
    double breadth;  // Breadth of a box
    double height;   // Height of a box
};
```

The keyword public determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as private orprotected

Creation of Objects:

Once the class is created, one or more objects can be created from the class as objects are instance of the class.

Objects are also declared as:

class name followed by object name;

exforsys e1;

This declares e1 to be an object of class exforsys.

For example a complete class and object declaration is given below:

```
class exforsys
{
  private:
  int x,y;
  public:
  void sum()
  {
    ……
    ……
  }
};
```

```
main()
{
 exforsys e1;
 ……
 ……
}
```

The object can also be declared immediately after the class definition. In other words the object name can also be placed immediately before the closing flower brace symbol } of the class declaration.

For example

```
class exforsys
{
 private:
 int x,y;
 public:
 void sum()
 {
  ……
  ……
 }
}e1 ;
```

The above code also declares an object e1 of class exforsys.

It is important to understand that in object-oriented programming language, when a class is created no memory is allocated. It is only when an object is created is memory then allocated.

Accessing the Data Members:

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear:

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

class Box
{
  public:
```

```
    double length;   // Length of a box
    double breadth;  // Breadth of a box
    double height;   // Height of a box
};

int main( )
{
  Box Box1;        // Declare Box1 of type Box
  Box Box2;        // Declare Box2 of type Box
  double volume = 0.0;     // Store the volume of a box here
   // box 1 specification
  Box1.height = 5.0;
  Box1.length = 6.0;
  Box1.breadth = 7.0;
  // box 2 specification
  Box2.height = 10.0;
  Box2.length = 12.0;
  Box2.breadth = 13.0;
  // volume of box 1
  volume = Box1.height * Box1.length * Box1.breadth;
  cout << "Volume of Box1 : " << volume <<endl;
  // volume of box 2
  volume = Box2.height * Box2.length * Box2.breadth;
  cout << "Volume of Box2 : " << volume <<endl;
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

Access Specifiers

Access specifiers are used to identify access rights for the data and member functions of the class.

There are three main types of access specifiers in C++ programming language: private public protected

A private member within a class denotes that only members of the same class have accessibility. The private member is inaccessible from outside the class.

Public members are accessible from outside the class.

A protected access specifier is a stage between private and public access. If member functions defined in a class are protected, they cannot be accessed from outside the class but can be accessed from the derived class.

When defining access specifiers, the programmer must use the keywords: private, public or protected when needed, followed by a colon and then define the data and member

Scope resolution operator

Scope resolution operator is used to uncover the hidden variables. It also allows access to global version of variables.

Eg:

```
#include<iostream. h>
int m=10; // global variable m
void main ( )
{
int m=20; // local variable m
cout<<"m="<<m<<"\n";
cout<<": : m="<<: : m<<"\n";
}
```

output:

20

10 (: : m access global m)

Scope resolution operator is used to define the function outside the class.

Syntax:

Return type <class name> : : <function name>

Eg:

Void x : : getdata()

MEMBER FUNCTION AND CLASS

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Member functions can be defined within the class definition or outside the class

MEMBER FUNCTION INSIDE THE CLASS

A member function of a class can also be defined inside the class. However, when a member function is defined inside the class, the class name and the scope resolution operator are not specified in the function header. Moreover, the member functions defined inside a class definition are by default inline functions.

Defining a member function within the class definition declares the function inline, even if you do not use the inline specifier. So either you can define Volume() function as below:

```
class Box
{
  public:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box
      double getVolume(void)
    {
      return length * breadth * height;
    }
};
```

MEMBER FUNCTION OUTSIDE THE CLASS

Member function outside the class can defined using scope resolution operator.defining a member function outside a class requires the function declaration (function prototype) to be provided inside the class definition. The member function is declared inside the class like a normal function. This declaration informs the compiler that the function is a member of the class

and that it has been defined outside the class. After a member function is declared inside the class, it must be defined (outside the class) in the program.

The definition of member function outside the class differs from normal function definition, as the function name in the function header is preceded by the class name and the scope resolution operator (: :). The scope resolution operator informs the compiler what class the member belongs to. The syntax for defining a member function outside the class is

Return_type class_name :: function_name (parameter_list)

{

// body of the member function

}

If you like you can define same function outside the class using scope resolution operator, :: as follows:

```
double Box::getVolume(void)
{
   return length * breadth * height;
}
```

Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (.) on a object where it will manipulate data related to that object only as follows:

A member function will be called using a dot operator (.) on a object where it will manipulate data related to that object only as follows:

```
Box myBox;          // Create an object
myBox.getVolume();  // Call member function for the object
```

Let us put above concepts to set and get the value of different class members in a class:

```
#include <iostream>
using namespace std;
class Box
{
  public:
     double length;        // Length of a box
     double breadth;       // Breadth of a box
     double height;        // Height of a box
```

```
    // Member functions declaration
    double getVolume(void);
    void setLength( double len );
    void setBreadth( double bre );
    void setHeight( double hei );
};
// Member functions definitions
double Box::getVolume(void)
{
   return length * breadth * height;
}
void Box::setLength( double len )
{
   length = len;
}
void Box::setBreadth( double bre )
{
   breadth = bre;
}
void Box::setHeight( double hei )
{
   height = hei;
}
// Main function for the program
int main( )
{
  Box Box1;              // Declare Box1 of type Box
  Box Box2;              // Declare Box2 of type Box
  double volume = 0.0;     // Store the volume of a box here
   // box 1 specification
  Box1.setLength(6.0);
  Box1.setBreadth(7.0);
  Box1.setHeight(5.0);
```

```
  // box 2 specification
  Box2.setLength(12.0);
  Box2.setBreadth(13.0);
  Box2.setHeight(10.0);
  // volume of box 1
  volume = Box1.getVolume();
  cout << "Volume of Box1 : " << volume <<endl;
  // volume of box 2
  volume = Box2.getVolume();
  cout << "Volume of Box2 : " << volume <<endl;
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Volume of Box1 : 210

Volume of Box2 : 1560

BASIC CONCEPTS OF OOPS

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

- Class definitions – Basic building blocks OOP and a single entity which has data and operations on data together
- Objects – The instances of a class which are used in real functionality – its variables and operations
- Abstraction – Specifying what to do but not how to do ; a flexible feature for having a overall view of an object's functionality.

- Encapsulation – Binding data and operations of data together in a single unit – A class adhere this feature
- Inheritance and class hierarchy – Reusability and extension of existing classes
- Polymorphism – Multiple definitions for a single name - functions with same name with different functionality; saves time in investing many function names Operator and Function overloading
- Generic classes – Class definitions for unspecified data. They are known as container classes. They are flexible and reusable.
- Class libraries – Built-in language specific classes
- Message passing – Objects communicates through invoking methods and sending data to them. This feature of sending and receiving information among objects through function parameters is known as Message Passing.

Features of OOPS.

- Emphasis is on data rather than on procedure.
- Programs are divided into objects.
- Data is hidden and cannot be accessed by external functions.
- Follows bottom -up approach in program design.

Applications of OOPS

- Real-time systems.
- Simulation and modeling.
- Object-oriented databases.
- AI and expert systems.

VARIOUS ACCESS SPECIFIERS

- Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type.

- The access restriction to the class members is specified by the labelled **public, private,** and **protected** sections within the class body.

- The keywords public, private, and protected are called access specifiers.

- A class can have multiple public, protected, or private labeled sections.

- Each section remains in effect until either another section label or the closing right brace of the class body is seen.

- The default access for members and classes is private.

```
class Base {
   public:
  // public members go here
   protected:
 // protected members go here
   private:
  // private members go here
 };
```

The public members:

A public member is accessible from anywhere outside the class but within a program.

```
#include <iostream>
class Line
{
  public:
     double length;
     void setLength( double len );
     double getLength( void );
};
 double Line::getLength(void)
{
   return length ;
}
void Line::setLength( double len )
{
   length = len;
```

```
}
int main( )
{
  Line line;
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;
  line.length = 10.0; // OK: because length is public
  cout << "Length of line : " << line.length <<endl;
  return 0;
}
```

Output

Length of line: 6

Length of line: 10

The private members:

- A **private** member variable or function cannot be accessed, or even viewed from outside the class.
- Only the class and friend functions can access private members.
- By default all the members of a class would be private

```
#include <iostream>

class Box
{
  double width;
  public:
    double length;
    void setWidth( double wid );
    double getWidth( void );
};
double Box::getWidth(void)
{
```

```
    return width ;
}
 void Box::setWidth( double wid )
{
    width = wid;
}
int main( )
{
   Box box;
   box.length = 10.0; // OK: because length is public
   cout << "Length of box : " << box.length <<endl;
  // box.width = 10.0; // Error: because width is private
   box.setWidth(10.0);
   cout << "Width of box : " << box.getWidth() <<endl;
    return 0;
}
```

Output:

Length of box: 10

Width of box: 10

The protected members:

A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

```
#include <iostream>
class Box
{
  protected:
     double width;
};
class SmallBox:Box
{
```

```
  public:
    void setSmallWidth( double wid );
    double getSmallWidth( void );
};
double SmallBox::getSmallWidth(void)
{
   return width ;
}
void SmallBox::setSmallWidth( double wid )
{
   width = wid;
}
int main( )
{
  SmallBox box;
  box.setSmallWidth(5.0);
  cout << "Width of box : "<< box.getSmallWidth() << endl;
   return 0;
}
```

Output:

Width of box: 5

STATIC MEMBER AND MEMBER FUNCTION

Static variable are normally used to maintain values common to the entire class.

Feature:

• It is initialized to zero when the first object is created. No other initialization is permitted

• only one copy of that member is created for the entire class and is shared by all the objects

• It is only visible within the class, but its life time is the entire class type and scope of each static member variable must be defined outside the class

• It is stored separately rather than objects

Eg: static int count//count is initialized to zero when an object is created.

int classname::count;//definition of static data member

```
#include<iostream.h>
#include<conio.h>
 class stat
{
   int code;
   static int count;
   public:
   stat()
   {
    code=++count;
   }
   void showcode()
   {
    cout<<"\n\tObject number is :"<<code;
   }
   static void showcount()
   {
          cout<<"\n\tCount Objects :"<<count;
   }
};

int stat::count;
void main()
{
  clrscr();
  stat obj1,obj2;

  obj1.showcount();
  obj1.showcode();
```

```
    obj2.showcount();
    obj2.showcode();
    getch();
}
```

Output:

Count Objects: 2

Object Number is: 1

Count Objects: 2

Object Number is: 2

INLINE FUNCTION

An inline function is a function that is expanded in line when it is invoked. The compiler replaces the function call with corresponding function code. The inline functions are defined as follows:

```
inline function-header
    {
     Function body;
    }
```

Example:

```
inline double cube(double a)
      {
        Return(a*a*a);
      }
```

Some situations where inline expansion may not work are:

- For functions returning values, if a loop , a switch, or a goto exists.
- For functions not returning values, if a return statement exists.
- If functions contain static variables.
- If inline functions are recursive.

-Example program to illustrate inline functions :

```
#include <iostream.h>
inline float mul(float x, float y)
  {    return(x*y);    }
```

```
inline double div(double p, double q)
    {   return(p/q) ;       }
int main( )
{
        float a=1.2 ;
        float b=2.3;
        cout<< mul(a,b)<<"\n";
        cout<< div(a,b)<<"\n";
         return 0;
}
```

One of the most useful facilities available in C++ is the facility to defined default argument values for functions. In the function prototype declaration, the default values are given. Whenever a call is made to a function without specifying an argument, the program will automatically assign values to the parameters from the default function prototype declaration. Default arguments facilitate easy development and maintenance of program.

DIFFERENT TYPES OF CONSTRUCTORS

PARAMETERIZED CONSTRUCTORS

The constructor that can take arguments are called parameterized constructor. The arguments can be separated by commas and they can be specified within braces similar to the argument list in function.

When a constructor has been parameterized, the object declaration without parameter may not work. In case of parameterized constructor, we must provide the appropriate arguments to the constructor when an object is declared. This can be done in two ways:

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

The implicit call method is sometimes known as shorthand method as it is shorter and is easy to implement.

Program : creating parameterized constructor

```
#include<iostream.h>
#include<conio.h>
class student
{
private:

int roll,age, marks;
public:
student(int r, int m, int a);                    //parameterized constructor
void display( )
{
cout<<"\nRoll number :" <<roll <<endl;
cout<<"Total marks : <<marks<<endl;
cout<<"Age:"<<age<<endl;
}
};                                               //end of class declaration
student : : student(int r, int m, int a)         //constructor definition
{
roll = r;
marks =m;
age=a;
}
int main( )
{
Student manoj(5,430,16);                         //object creation
Cout<<"\n Data of student 1:"<<endl;
manoj.display();
Student ram(6,380,15);                           //object creation
Cout<<"\n Data of student 1:"<<endl;
ram.display();
getch();
return 0;
```

```
}
```

Data of student 1:

Roll number : 5

Total Marks : 430

Age : 16

Data of student 2:

Roll number : 6

Total Marks : 380

Age : 15

COPY CONSTRUCTOR

Copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

Syntax:

```
classname (const classname &obj)
 {
// body of constructor
}
```

Program : Copy Constructor

```
#include<iostream.h>
#include<conio.h>
class student
{
private:
int roll, marks, age;
public:
student(int r,int m,int a)                    //parameterized constructor
{
```

```
rol l = r;
marks = m;
age = a;
}
student(student &s)                    //copy constructor
{
roll = s.roll;

marks = s.marks;
age=s.age;
}
void display( )
{
cout<<"Roll number :" <<roll <<endl;
cout<<"Total marks :"<<marks<<endl;
cout<<"Age:"<<age<<endl;
}
};
int main( )
{
clrscr();
student t(3,350,17);                   // or student k(t);
student k = t;                         // invokes copy constructor
cout<<"\nData of student t:"<<endl;
t.display();
cout<<"\n\nData of student k:"<<endl;
k.display();
getch();
return 0;
}
```

The outout of the above program will be like this:

Data of student t :

Roll number : 3

Total marks : 350

Age : 17

Data of student k :

Roll number : 3

Total marks : 350

Age : 17

DEFAULT CONSTRUCTOR

n C++, the standard describes the default constructor for a class as a constructor that can be called with no arguments (this includes a constructor whose parameters all have default arguments).[1] For example:

```
class MyClass
{
public:
MyClass();  // constructor declared

private:
int x;
};

MyClass :: MyClass()  // constructor defined
{
x = 100;
}

int main()
{
MyClass m;  // at runtime, object m is created, and the default constructor is called
}
```

DYNAMIC CONSTRUCTORS

The constructor 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. 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 the new operator.

Program: illustrate the working of dynamic constructors:

```
#include<iostream.h>
 class  Sample
    {
          char  *name;
          int  length;
      public:
           Sample( )
             {
                 length = 0;
                 name = new char[ length + 1];
              }
          Sample ( char  *s )
             {
                 length = strlen(s);
                 name = new char[ length + 1 ];
                 strcpy( name , s );  }
          void  display( )
            {
                 cout<<name<<endl;
            }  };
        int  main( )
          {
                 char  *first = " C++ ";
                 Sample  S1(first), S2("ABC"), S3("XYZ");
                 S1.display( );
                 S2.display( );
                 S3.display( );
                 return 0;   }
```

RESULT :

C++

ABC

XYZ

POINTERS

- C++ pointer can also have address for an member functions. They are known as pointer to member function.
- Pointer to function is known as call back functions.
- The size of the object is determined by the size of all non-static data members.

Pointer to member use either

- star-dot combination
- arrow notation

Syntax: Class_name *Ptr_var;

Ptr_var=new student;

```
class student
{
public:
int rollno;
string name;
void print()
{
cout<<”rollno”<<rollno;
cout<<”name”<<name;
}};
void main() //Star-dot notation
{
student *stu-ptr;
stu-ptr=new student;
(*stu-ptr).rollno=1;
```

```
(*stu-ptr).name=”xxx”;
(*stu-ptr).print();
}
```

CONSTANT DATA MEMBER

These are data variables in class which are made const. They are not initialized during declaration. Their initialization occur in the constructor.

Example:

```
class Test
{
 const int i;                    // Constant Data Member
 public:
 Test (int x)
 {
   i=x;
 }
};
int main()
{
 Test t(10);
 Test s(20);
}
```

In this program, **i** is a const data member, in every object its independent copy is present, hence it is initialized with each object using constructor. Once initialized, it cannot be changed.

Constant class Member function

A const member function never modifies data members in an object.

Syntax :

return_type function_name() const;

Example for const Object and const Member function

```
class X
{
 int i;
 public:
 X(int x)                    // Constructor
 {
 i=x;
 }
 int f() const               // Constant function
 {
 i++;
}
 int g()
{
 i++;
}
};
int main()
{
X obj1(10);                  // Non const Object
const X obj2(20);            // Const Object
obj1.f();                    // No error
obj2.f();                    // No error
cout << obj1.i << obj2.i ;
obj1.g();                    // No error
obj2.g();                    // Compile time error
}
```

Output : 10 20

Here, we can see, that const member function never changes data members of class, and it can be used with both const and non-const object. But a const object can't be used with a member function which tries to change its data members.

REFERENCES

PROGRAM:

```
#include<iostream>
using namespace std;
void swap(int &x,int &y)
{
int t;
t=x;
x=y;
y=t;
}
int  main()
{
int a,b;
cout<<" enter two integers<a,b>";
cin>>a>>b;
swap(a,b);
cout<<"value of a & b on swap(a,b) in main():"<<a<<" "<<b;
return 0;
}
```

OUTPUT

```
 enter two integers<a,b>25 9
value of a & b on swap(a,b) in main():9 25
```

STORAGE CLASSES

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program.

These specifiers precede the type that they modify.

There are following storage classes, which can be used in a C++ Program

auto

register

static

extern

mutable

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
  int mount;
  auto int month;
}
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

The register Storage Class

- The **register** storage class is used to define local variables that should be stored in a register instead of RAM.
- This means that the variable has a maximum size equal to the register size and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  miles;
}
```

- The register should only be used for variables that require quick access such as counters.
- It should also be noted that defining 'register' does not mean that the variable will be stored in a register.
- It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

- The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope.
- Therefore, making local variables static allows them to maintain their values between function calls.
- The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.
- In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

```
#include <iostream.h>
void func(void);
static int count = 10; /* Global variable */
main()
{
   while(count--)
   {
     func();
   }
   return 0;
}
// Function definition
void func( void )
{
   static int i = 5; // local static variable
   i++;
   cout << "i is " << i ;
   cout << " and count is " << count <<endl;
}
```

Output:

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
```

i is 10 and count is 5

i is 11 and count is 4

i is 12 and count is 3

i is 13 and count is 2

i is 14 and count is 1

i is 15 and count is 0

The extern Storage Class

- The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files.
- When 'extern' is used the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

```
#include <iostream>

int count ;
extern void write_extern();

main()
{
  count = 5;
  write_extern();
}
```

The mutable Storage Class

- The **mutable** specifier applies only to class objects.
- It allows a member of an object to override constness.
- That is, a mutable member can be modified by a const member function.

DYNAMIC MEMORY ALLOCATION

- Memory allocated "on the fly" during run time
- dynamically allocated space usually placed in a program segment known as the heap or the free store

- Exact amount of space or number of items does not have to be known by the compiler in advance.
- For dynamic memory allocation, pointers are crucial
- We can dynamically allocate storage space while the program is running, but we cannot create new variable names "on the fly"
- For this reason, dynamic allocation requires two steps:
- Creating the dynamic space.
- Storing its address in a pointer (so that the space can be accesed)
- To dynamically allocate memory in C++, we use the new operator.

De-allocation:

- Deallocation is the "clean-up" of space being used for variables or other data storage
- Compile time variables are automatically deallocated based on their known extent (this is the same as scope for "automatic" variables)
- It is the programmer's job to deallocate dynamically created space
- To de-allocate dynamic memory, we use the delete operator
- C++ integrates the operators new and delete for allocating dynamic memory.

New operator:

Allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called new operator.

There is following generic syntax to use new operator to allocate memory dynamically for any data-type.

new datatype;

Here, data-type could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types.

To allocate space dynamically, use the unary operator new, followed by the type being allocated.

```
new int;        // dynamically allocates an int
new double;     // dynamically allocates a double
```

If creating an array dynamically, use the same form, but put brackets with a size after the type:

```
new int[40];      // dynamically allocates an array of 40 ints
new double[size]; // dynamically allocates an array of size doubles
                  //  note that the size can be a variable
```

These statements above are not very useful by themselves, because the allocated

Used to free memory allocated with new operator

The **delete operator** should be called on a pointer to dynamically allocated memory when it is no longer needed

Can delete a single variable/object or an array

delete PointerName;

delete [] ArrayName;

After delete is called on a memory region, that region should no longer be accessed by the program

Convention is to set pointer to deleted memory to NULL

Any new must have a corresponding delete --- if not, the program has memory leak.

```
include <iostream>
#include <cstring>
using namespace std;
int main() {
    int n;
    cout << "Enter total number of students: ";
    cin >> n;
    float* ptr;
   ptr = new float[n];       // memory allocation for n number of floats
    cout << "Enter GPA of students." <<endl;
    for (int i = 0; i < n; ++i) {
        cout << "Student" << i+1 << ": ";
        cin >> *(ptr + i);
    }
    cout << "\nDisplaying GPA of students." << endl;
    for (int i = 0; i < n; ++i) {
        cout << "Student" << i+1 << " :" << *(ptr + i) << endl;
    }
   delete [] ptr;    // ptr memory is released
   return 0;
}
```

Enter total number of students: 4

Enter GPA of students.

Student1: 3.6

Student2: 3.1

Student3: 3.9

Student4: 2.9

Displaying GPA of students.

Student1 :3.6

Student2 :3.1

Student3 :3.9

Student4 :2.9

DEFAULT ARGUMENTS

Default arguments assign a default value to the parameter, which does not have matching argument in the function call.

Default values are specified when the function is declared.

The advantages of default arguments are,

- We can use default arguments to add new parameters to the existing function.
- Default arguments can be used to combine similar functions into one.

```
#include<iostream.h>
int volume(int length, int width = 1, int height = 1);
int main()
{
int a,b,c;
a=solume(4, 6, 2);
b=volume(4, 6);
c=volume(4);
return 0;
}
int volume(int length, int width=1, int height=1)
{
int v;
```

```
v=length*width*height;
return v;
}
```

ROLE OF 'this' POINTER

Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

```
#include <iostream>

using namespace std;

class Box

{

  public:

    // Constructor definition

    Box(double l=2.0, double b=2.0, double h=2.0)

    {

      cout <<"Constructor called." << endl;

      length = l;

      breadth = b;

      height = h;

    }

    double Volume()
```

```
    {
      return length * breadth * height;
    }
    int compare(Box box)
    {
      return this->Volume() > box.Volume();
    }
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

int main(void)
{
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2

  if(Box1.compare(Box2))
  {
    cout << "Box2 is smaller than Box1" <<endl;
```

```
    }
    else
    {
      cout << "Box2 is equal to or larger than Box1" <<endl;
    }
    return 0;
}
```