Biyani's Think Tank

Concept based notes

C++

(MCA)

Asst. Prof.
Nitasha Jain
Biyani Institute of Science and Management, Jaipur

Published by :
Think Tanks
Biyani Group of Colleges

Sector-3, Vidhyadhar Nagar,
Jaipur-302 023 (Rajasthan)
Ph : 0141-2338371, 2338591-95 • Fax : 0141-2338007
E-mail : acad@biyanicolleges.org
Website :www.gurukpo.com; www.biyanicolleges.org

Edition: 2011
New Edition: 2012

Leaser Type Setted by:
Biyani College Printing Department

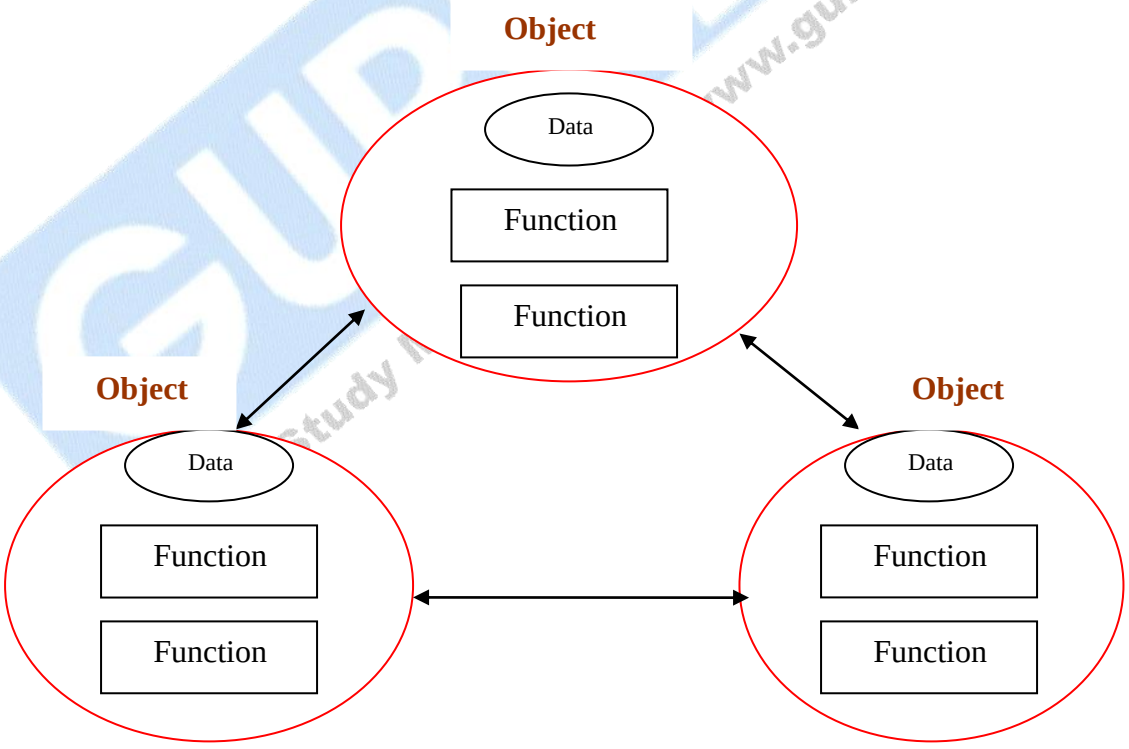
Index

Object Oriented Paradigm

Q 1. What is Object–Oriented Paradigm?

Ans: OOP is a method of implementation in which programs are organized as co-operative collections of objects, each of which represents an instance of some class and whose classes are all members of a hierarchy of classes united through the property called Inheritance.

Three important concepts to be noted in the above definition are: Object, Classes, and Inheritance.OOP uses objects and not algorithms as its fundamental building blocks. Each object is an instance of some class. Classes allow the mechanism of data abstraction for creating new data types .Inheritance allows building of a new classes from the existing classes' .Hence, if any of these elements are missing in a program, then it is not object-oriented one; it resembles the program with abstract data types.



Q 2. What are the elements of Object-Oriented Programming? Explain its key components such as object and classes with example?

Ans: Object-Oriented Programming is centered around new concepts such as objects, classes, polymorphism, inheritance, etc. It is a well-suited paradigm for the following:

- Modeling the real-world problem as close as possible to the user's perspective.
- Interacting easily with computational environment using familiar metaphors.
- Constructing reusable software components and easily extendable libraries.
- Easily modifying and extending implementations of components without having to record every thing from scratch.

A Language's quality is judged by twelve important criteria.
They are a well defined syntactic and semantic structure, reliability, fast translation, efficient object code, orthogonality, machine independence, provability, generality, and consistency with commonly used notations, subsets, uniformity and extensibility. The various constructs of OOP language are designed to achieve these with ease.

Key components of Object-Oriented Programming are:

Class: - of A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.
Object:- An *object* is an instantiation a class. In terms of variables, a class would be the type, and an object would be the variable.

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;

Q 3. What are the fundamental features of the OOPs in C++?

Ans: Object Oriented Programming (OOP) is a programming paradigm that uses "objects" and their interactions to design applications and computer programs. It is based on several techniques, including encapsulation, modularity, polymorphism, and inheritance.

Inheritance :- It allows the extension and reuse of existing code without having to rewrite the code from scratch. Inheritance involves the creation of new classes(derived cases) from the existing ones (base classes),thus enabling the creation of a hierarchy of classes that simulate the class and subclass concept of the real world. The new derived class inherits the members of the base class and also adds its own.

Encapsulation :- It is a mechanism that associates the code and the data it manipulates into a single unit and keeps then safe from external interference and misuse. In c++ it is supported by a construct called class. An instance of a class is known as an object, which represents a real world.

Data Abstraction :- The technique of creating new data types that are well suited to an application to be programmed is known as data abstraction. it provides the ability to create user-defined data types, for modeling a real world object, having the properties of built-in data types and a ser of permitted operators. The class is a construct in c++ for creating user-defined data types called abstract data types.

Polymorphism :- It allows a single name/operator to be associated with different operations depending on the type of data passed to it .In c++, it is achieved by function overloading ,operator overloading and dynamic binding(virtual function).

Message _Passing:- It is the process of invoking an operation on an object .In response to a message; the corresponding method is executed in the object.

Q 4. How the comments are declared in C++ language?

Ans: C++ has borrowed the new commenting style from Basic Computer Language, the predecessor of the C language. In C language, comments is /are enclosed between /* and */ character pairs. It can be either used for single line comment or multiple line comment.

Single line comment in C language :

```
/*hello world*/
```

Multiple line comment in C language:

```
/* multiple line comments are
    Designed like that*/
```

Apart from the above style of commenting, C++ supports a new style of commenting .It starts with two forward slashes i.e. // (without separation by spaces) and ends with the end-of-line character.
The syntax for the style of C++ comment is

```
[Any c++ statement]  // I am a C++ comment
```

For example: int acc; //account number in integer format

Q 5. What is the compilation process of C++?

Ans: The program coded by the programmer is called by the source code. This source code is supplied to the compiler for converting it into the machine code. C++ programs make use of libraries. A library contains the object code of standard function. The object code of all function used in the program have to be combine with the program written by the programmer. In addition, some start- up code is required to produce an executable version of the program .this process of combining all the required object codes and the start-up code is called linking and the final product is called the executable code.

Q 6. Explain the stream based I/O?

Ans: C++ supports a rich set of functions for performing input and output operations. The syntax of using these IO functions is totally consistent ,irrespective of the device with which IO operation are performed C++ new features for handling IO operation are called stream. Streams are abstractions that refer to data flow. Streams in C++ are classified into

1. Output streams
2. Input streams

Output Streams :The output streams allow to perform write operation on output devices such as screen ,disk etc.output on the standard stream is performed using the "cout" object.C++ uses the bit –wise left-shift operator for performing console output operation. The syntax for standard output stream operation is as follow:

```
cout<<variable;
```

The word cout is followed by the symbol << called the insertion or put-to operator, and then with the items that are to be output. Variables can be of any basic data types.

Example of cout

```
cout << "hello world";
```

it will display hello world on the screen.

Input Streams: The input streams allow to performed read operation with input devices such as keyboard, disk etc.Input from the standard stream is performed using the cin object .C++ uses the bit-wise right-shift operator for performing console input operation. The syntax for standard input streams operation is as follow

```
cin>>variable;
```

Example of cin:

```
cout<<"enter the age";
cin>>age;
```

The two important points to be noted about the streams operations:

1. Streams do not require explicit data type specification in IO statement.
2. Streams do not require explicit address operator prior to the variable in the input statement.

Q 7. Expalin the scope resolution operator"::" ?

Ans: C++ supports a mechanism to access a global variable from a function in which a local variable is defined with the same name as a global variable. It is achieved using the scope resolution operator.
The syntax for accessing a global variable using the scope resolution operator:

```
:: GlobalVariableName    //Syntax of global variable access
```

The global variable to be accessed must be proceeded by the scope resolution operator. It directs the compiler to access a global variable, instead of one define as a local variable.

"The scope resolution operator permits a program to reference identifiers in the global scope that has been hidden by another identifier with the same name in the local scope."

Q 8. Explain the constructs supported by C++ for runtime memory management?

Ans: Whenever an array is defined a specified amount of memory is set aside at compile time, which may not be utilized fully or may not be sufficient .If a situation in which the amount of memory required is unknown at compile time ,the memory allocation can be performed during execution .Such a technique of allocating memory during runtime on demand is Known as Dynamic Memory Allocation.

C++ provides the following two special operators to perform memory management dynamically

1. "new" Operator for dynamic memory allocation
2. "delete" Operator for dynamic memory deallocation

The memory management functions such as malloc (), calloc (), and free () in C, have been improved and evolved in c++ as the new and delete operators to accomplish dynamic memory allocation and deallocation respectively.

Q 9. Explain the new and delete operator with their syntax and example?

Ans: New Operator: The new operator offers dynamic storage allocation similar to the standard library function malloc ().It is particularly designed keeping OOP in mind and throws and exception if memory allocation fails.

The general format of the new operator is shown below:

Data type * new Data type[size in integer]; //Memory allocation in C++

The C++ statement:

Ptrvar=new Data type[integer size];

The operator new allocates a specified amount of memory during run time and returns a pointer to that memory location .It computes the size of the memory to be allocated by

Size of (data type)*integer size * Where data type can be a standard data type or a user defined data type. Integer size can be an integer expression which specifies the number of elements in the array .The new operator returns null if memory allocation is unsuccessful.

For example

```
int *a;
a=new int [100];
```

Delete Operator: The new operator counter part, delete, ensures the safe and efficient use of memory. This operator is used to return the memory allocated by the new operator back to the memory pool .Memory thus released, will be reused by the other parts of the program. All though the memory allocate is return automatically to the system when the programs terminates ,it is safer to use this operator explicitly with in the pointer this is absolutely necessary in situation where local variable pointing to the memory get destroy when the function terminates ,leaving memory in accessible to the rest of the program.

The syntax of the delete operator is below:
Delete ptrvar;
For example
Delete a;

Q 10. Write an interactive program of addition of two vectors using C++'s memory management operator?

Ans:

```
#include<iostream.h>
#include<conio.h>
void addvector (int *a, int *b, int *c, int size)
{
for (int i=0;i<size;i++)
{
c[i] =a[i] +b[i];
}
}
void readvector (int *vector, int size)
 {
      for(int i=0; i<size; i++)
cin>>vector[i];
 }
void showvector (int *vector, int size)
 {
      for (int i=0; i<size; i++)
      {
   cout<<vector[i] <<" ";
      }
 }
void main()
  {
int vec_size;
int *x,*y,*z;
cout<<"enter size of vector ";
cin>>vec_size;
//Allocates memory for all the three vectors
x=new int[vex_size];
```

```
y= new int[vex_size];
z=new int[vex_size];
cout<<"enter elements of vector x:";
Readvector(x, vec_size);
cout<<"enter elements of vector y:";
Readvector(y, vec_size);
Addvector(x, y, z, vec_size);  //z=x+y
cout<<"Display result is::";
Showvector (z, vec_size);
//free memory allocated to all the three vectors
delete x;
delete y;
delete z;
getch();
 }
```

Data types, Operators and Expressions

Q 1. What are the specific Keywords available in C++?

Ans: There are several keywords specific top C++ which are listed in table .These keywords primarily deal with classes, templates and exception handling.

asm	new	template
catch	operator	this
class	private	throw
delete	protected	try
friend	public	virtual
inline		

Q 2. What is type conversion?

Ans: A mixed mode expression is one in which the operands are not of the same type the operands are converted before evaluation to maintain compatibility between data types. It can be carried out by the complier automatically or by the programmer explicitly.

There are two types of conversion

1. Implicit type conversion:-The complier performs type conversion of data items when an expression consists of data items of different types. This is called implicit or automatic type conversion. The rule follow by the complier for implicit type conversion is shown below

Operands 1	Operands 2	Result
char	int	int
int	long	long
int	float	float
int	double	double
int	unsigned	unsigned
long	double	double
double	float	double

Automatic type conversion rule table

Consider the following statement to illustrate automatic type conversion

```
Float f=10.0;
Int i=0;
i=f/3;
```

in this expression the constant three will be converted to a float and then the floating point division will take place, resulting in 3.3333.this type of conversion, where the variable of a lower data type is converted to a higher data type is called promotion.
The type conversion where the variables of a higher type are converted to a lower type is called demotion.
The implicit conversion thus occurring is also called silent conversions since the programmer is not aware of these conversions. The flexibility of the C++ language to allow mixed type conversions implicit, save a lot of efforts on the part of the programmer, but at times it can give rise to bugs in the program.

Q 3. Define Enumerated Data types with example.

Ans: An enumerated data type is a user defined type, with values ranging over a finite set of identifiers called enumeration constants. For example:
enum color (red, blue, green);
This defines color to be of a new data type which can assume the value red, blue, or green .each of these is an enumeration constant .In this statement color can be used as a new type. A variable of type color can have any one of the three values: red, blue or green.
for example: Color c;
Enum example

```
#include <iostream.h>
void main()
{
enum Days{Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday;
Days TheDay;
int j;
cout<<"Please enter the day of the week (0 to 6)";
cin>>j;
TheDay = Days (j);
if(TheDay == Sunday || TheDay == Saturday)
cout<<"Hurray it is the weekend"<<endl;
else
cout<<"Curses still at work"<<endl;
getch();
}
```

Q 4. What is sizeof operator ?

Ans: The operator sizeof returns the number of bytes required to represent a data type or variable .It has the following forms:
sizeof (datatype)
sizeof(variable)
the data type can be standard or user defined data type.the following statements illustrate the usage of sizeof operator:

```
#include<iostream.h>
void main()
{
int i,j;
float c;
cout<<"the size of int is::"<<sizeof(int)<<endl;
cout<<"the size of float is::"<<sizeof(float)<<endl;
cout<<"the sze of integer variable is::"<<sizeof(i)<<endl;
cout<<"the size of float variable is::"<<sizeof(c)<<endl;
getch();
}
```

Q 5. What is conditional operator (ternary operator)?

Ans: The ternary operator "?:" earns its name because it's the only operator to take three operands. It is a conditional operator that provides a shorter syntax for the if..then..else statement. The first operand is a boolean expression; if the expression is true then the value of the second operand is returned otherwise the value of the third operand is returned:
boolean expression ? value1: value2
The following if..then..else statement:

```
boolean isHappy = true;
 String mood = "";
if (isHappy == true)
 {
 mood = "I'm Happy!";
}
else
{
 mood = "I'm Sad!";
 }
```

can be reduced to one line using the ternary operator:

```
boolean isHappy = true;
String mood = (isHappy == true)?"I'm Happy!":"I'm Sad!";
```

Generally the code is easier to read when the if..then..else statement is written in full but sometimes the ternary operator can be a handy syntax shortcut.

Q 6. What are data types available in C++?

Ans: Boolean type

bool - type, capable of holding one of the two values: true or false.

Character types

signed char - type for signed character representation.

unsigned char - type for unsigned character representation.

char - type for character representation which can be most efficiently processed on the target system (equivalent to either signed char or unsigned char).

Integer types

int - basic integer type.

Modifiers

Modifies the integer type. Can be mixed in any order. Only one of each group can be present in type definition.

Signedness

signed- target type will have signed representation (this is the default if omitted)

unsigned- target type will have signed representation

Size

short - target type will be optimized for space and will have width of at least 16 bits.

long - target type will have width of at least 32 bits.

long long - target type will have width of at least 64 bits (C++0x)

Floating point types

float - single precision floating point type. Usually IEEE-754 32 bit floating point type

double - double precision floating point type. Usually IEEE-754 64 bit floating point type

long double - extended precision floating point type.

Range of values

Note: C++ standard does not define neither sizes nor value ranges of arithmetic types.

This is implementation-defined

Type	Size in bits	Value range
character	8 (signed)	**-128** to **127**
	8 (unsigned)	**0** to **255**
integral	16 (signed)	**-32768** to **32767**
	16 (unsigned)	**0** to **65535**
	32 (signed)	**-2,147,483,648** to **2,147,483,647**
	32 (unsigned)	**0** to **4,294,967,295**
	64 (signed)	**-9,223,372,036,854,775,808** to **9,223,372,036,854,775,807**
	64 (unsigned)	**0** to **18,446,744,073,709,551,615**
floating point	32 (IEEE-754)	**± 3.4** $\cdot 10^{\pm 38}$ (~7 digits)
	64 (IEEE-754)	**± 1.7** $\cdot 10^{\pm 308}$ (~15 digits)

Control flow

Q 1. Define if statement with example.

Ans: The if statement enables you to test for a condition (such as whether two variables are equal) and branch to different parts of your code, depending on the result.

The simplest form of an if ,if else and nested if else statements is this:

```
//----- if statement with a true clause
  if (expression) {
     statements // do these if  expression is true
  }

//----- if statement with true and false clause
  if (expression) {
     statements // do these if expression is true
  } else {
     statements // do these if expression is false
  }

//----- if statements with many parallel tests
  if (expression1) {
     statements // do these if expression1 is true
  } else if (expression2) {
     statements // do these if expression2 is true
  } else if (expression3) {
     statements // do these if expression3 is true
  . . .
  } else {
     statements // do these no expression was true
  }
```

For example:

```
//greater out of 2 numbers.
#include <iostream.h>
#include<conio.h>
void main()
```

```
{
 // define two integers
 int x = 3;
 int y = 4;

 //print out a message telling which is bigger
 if (x > y)
{
  cout << "x is bigger than y" << endl;
 }
 else {
  cout << "x is smaller than y" << endl;
 }
 getch();
}
```

Q 2. Define switch statements with example.

Ans: if and if/else statements can become quite confusing when nested too deeply, and C++ offers an alternative. Unlike if, which evaluates one value, switch statements allow you to branch on any of a number of different values. The general form of the switch statement is:

```
switch (expression)
{
case valueOne: statement;
               break;
case valueTwo: statement;
               break;
....
case valueN:   statement;
               break;
default:       statement;
}
```

expression is any legal C++ expression, and the statements are any legal C++ statements or block of statements. switch evaluates expression and compares the result to each of the case values. Note, however, that the evaluation is only for equality; relational operators may not be used here, nor can Boolean operations.

If one of the case values matches the expression, execution jumps to those statements and continues to the end of the switch block, unless a break statement is encountered. If nothing matches, execution branches to the optional default statement. If there is no default and there is no matching value, execution falls through the switch statement and the statement ends.

NOTE: It is almost always a good idea to have a default case in switch statements. If you have no other need for the default, use it to test for the supposedly impossible case, and print out an error message; this can be a tremendous aid in debugging.

It is important to note that if there is no break statement at the end of a case statement, execution will fall through to the next case statement. This is sometimes necessary, but usually is an error. If you decide to let execution fall through, be sure to put a comment, indicating that you didn't just forget the break.

This C++ program illustrates use of the switch statement.

Demonstrating the switch statement.

```
// Demonstrates switch statement
#include <iostream.h>
void main()
{
  unsigned short int number;
  cout << "Enter a number between 1 and 5: ";
  cin >> number;
  switch (number)
  {
                    case 0:
                     cout << "Too small, sorry!";
                              break;
            case 5:
                    cout << "Good job!\n";
                    break;
            case 4:
                    cout << "Nice Pick!\n";
                    break;
            case 3:
                    cout << "Excellent!\n";
                    break;
            case 2:
```

```
                    cout << "Masterful!\n";
                    break;
            case 1:
                    cout << "Incredible!\n";
                    break;
            default:
                    cout << "Too large!\n";
                    break;
        }
        cout << "\n\n";
        getch();
}
```

Output: Enter a number between 1 and 5: 3
Excellent!
Masterful!
Incredible!

Enter a number between 1 and 5: 8
Too large!

Q 5. Describes the while loop with example.

Ans: Iteration structures (loops)

The while loop

The **while** statement tests an expression. If the expression evaluates to true, it executes the body of the while. If it is false, execution continues with the statement after the while body. Each time after the body is executed, execution starts with the test again. This continues until the expression is false or some other statement (break or return) stops the loop.

Its format is:

```
Initilization;
while (condition)
{
        Statement;
}
```

For example, we are going to make a program to countdown using a while-loop:

```
  // custom countdown using while
```

```
#include <iostream>
#include<conio.h>
void main ()
{
  int n;
  cout << "Enter the starting number > ";
  cin >> n;
  while (n>0)
  {
                          cout << n << ", ";
                         --n;
  }
  cout << "Finish\n";
  getch();
}
```

Output:

Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, Finish!

When the program starts the user is prompted to insert a starting number for the countdown. Then the while loop begins, if the value entered by the user fulfills the condition n>0 (that n is greater than zero) the block that follows the condition will be executed and repeated while the condition (n>0) remains being true.
The whole process of the previous program can be interpreted according to the following script (beginning in main):
1. User assigns a value to n
2. The while condition is checked (n>0). At this point there are two possibilities:
    * condition is true: statement is executed (to step 3)
    * condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
cout << n << ", ";
--n;
(prints the value of n on the screen and decreases n by 1)
4.End of block. Return automatically to step 2
5.Continue the program right after the block: print Finish! and end program.
When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the

condition to become false at some point, otherwise the loop will continue looping forever. In this case we have included --n; that decreases the value of the variable that is being evaluated in the condition (n) by one - this will eventually make the condition (n>0) to become false after a certain number of loop iterations: to be more specific, when n becomes 0, that is where our while-loop and our countdown end.
Of course this is such a simple action for our computer that the whole countdown is performed instantly without any practical delay between numbers.

Q 6. Describes the do while loop with example.

Ans: **The do-while loop**

This is the least used of the loop statements, but sometimes you want a loop that executes one time before testing.

do {
statements
} **while** (*testExpression*);

Its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if condition is never fulfilled. For example, the following example program echoes any number you enter until you enter 0.

```
// number echoer
#include<conio.h>
#include <iostream>
void main ()
{
 unsigned long n;
 do {
  cout << "Enter number (0 to end): ";
  cin >> n;
  cout << "You entered: " << n << "\n";
 } while (n != 0);
 getch();
}
```

Output:

Enter number (0 to end): 12345
You entered: 12345

Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the value 0 in the previous example you can be prompted for more numbers forever.

Q 7. Describes the for loop with example.

Ans: **The for loop**

Many loop have an initialization before the loop, and some "increment" before the next loop. The **for** loop is the standard way of combining these parts.

```
for (initialStmt; testExpr; incrementStmt) {
 statements
}
```

This is the same as:

```
initialStmt;
while (testExpr) {
statements
incrementStmt
 }
```

It works in the following way:
1.initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2.condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
3.statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
4.finally, whatever is specified in the increase field is executed and the loop gets back to step 2.
Here is an example of countdown using a for loop:

```
// countdown using a for loop
#include <iostream>
#includ<conio.h>
void main ()
```

```
{
    for (int n=10; n>0; n--)
  {
                        cout << n << ", ";
  }
  cout << "Finish!\n";
          getch();
}
```

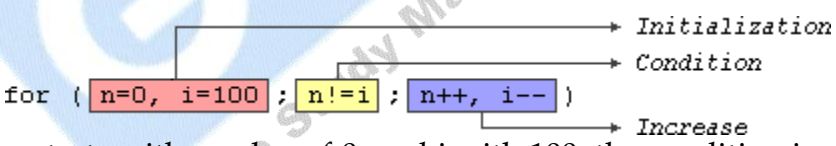
Output:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, Finish!

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: for (;n<10;) if we wanted to specify no initialization and no increase; or for (;n<10;n++) if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before). Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop, like in initialization, for example. The comma operator (,) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
  // whatever here...
}
```

This loop will execute for 50 times if neither n or i are modified within the loop:



n starts with a value of 0, and i with 100, the condition is n!=i (that n is not equal to i). Because n is increased by one and i decreased by one, the loop's condition will become false after the 50th loop, when both n and i will be equal to 50.

Q 8. Describe Jump statements.

Ans: There are three types of jump statements

1. break
2. goto
3.continue
4.exit

The break statement

Using break we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end (maybe because of an engine check failure?):

```
// break loop example
#include <iostream>
void main ()
{
    int n;
    for (n=10; n>0; n--)
    {
        cout << n << ", ";
        if (n==3)
        {
            cout << "countdown aborted!";
            break;
                }
    }
   getch();
}
```

Output: 10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:

```
// continue loop example
#include <iostream>
void main ()
{
```

```
  for (int n=10; n>0; n--)
  {
              if (n==5) continue;
               cout << n << ", ";
  }
  cout << "Finish!\n";
             getch();
}
```

Output:10, 9, 8, 7, 6, 4, 3, 2, 1, Finish!

The goto statement

goto allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.

The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).

Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using goto:

```
// goto loop example
#include <iostream>
void main ()
{
   int n=10;
   loop:
   cout << n << ", ";
   n--;
   if (n>0) goto loop;
   cout << "FIRE!\n";
            getch();
}
```

Output:
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

The exit function

exit is a function defined in the cstdlib library.

The purpose of exit is to terminate the current program with a specific exit code. Its prototype is:
void exit (int exitcode);
The exitcode is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened.

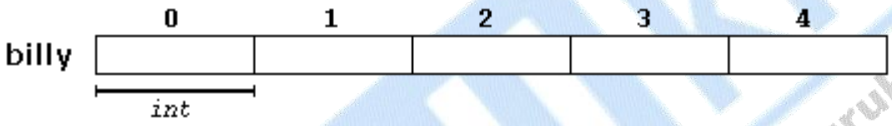
Arrays and Strings

Q1. What is array?

Ans: An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, we can store 5 values of type int in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, int for example, with a unique identifier.

For example, an array to contain 5 integer values of type int called billy could be represented like this:



where each blank panel represents an element of the array, that in this case are integer values of type int. These elements are numbered from 0 to 4 since in arrays the first index is always 0, independently of its length.

Like a regular variable, an array must be declared before it is used.

A typical declaration for an array in C++ is:

Datatype Arrayname [array_size];

where type is a valid type (like int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies how many of these elements the array has to contain.

Therefore, in order to declare an array called billy as the one shown in the above diagram it is as simple as:

int billy[5];

NOTE: The elements field within brackets [] which represents the number of elements the array is going to hold, must be a **constant** value, since arrays are blocks of non-dynamic memory whose size must be determined before execution.

Q 2. Define the types of array.

Ans. There are 3 types of Array

1. One dimensional:

By definition, an array is collection of data that is contiguous in memory. Like any variable, an array must be declared before it is used. There are also numerous types of arrays. The first type is the standard 1 dimensional array. A 1D array contains 1 long row of data. Here is how to declare a 1D array:
type name[**capacity**];

Where *type* is the data type of the array (i.e int, char, bool), name is an appropriate name for the array and capacity is the amount of space in the array. A rule is that the capacity cannot be negative or zero and is always an integer.

Notice the syntax of the array declaration. There are square brackets indicating the capacity. The brackets are used to show there is an array declared.

Data in an array is accessed with the [] operator. What goes inside the brackets is the index. By definition, each individual cell in an array is called an index. Indexes of an array ALWAYS BEGIN AT 0 (zero) and end at capacity-1.

2. Two dimensional:

A two dimensional array in C++ can be thought of as a table containing rows and columns. A 2 dimensional array is still contiguous in memory but now has a row and a column. It contains the other properties of an array in respect to the indices, capacity and data type.

Here is how to declare a 2D array:

type name[**rows**][**columns**];

3. Multi dimensional:

Q 3. How to access the array elements?

Ans: Accessing Array Elements

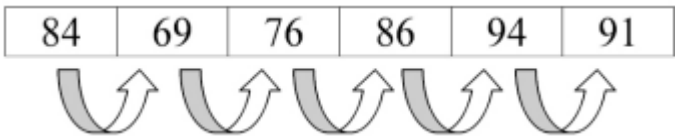
Once an array variable is defined,its elements can be accesed by using an index.the syntax for accessing array elements is shown :
Arrayname [index];
To access a particular element in the array, specify the array name followed by an integer constant or variable enclosed within square braces. the array index

,indicates the element of the array, which has to be accessed .for instance the expression
Age[4]
Accesses the 5th element of the array age.note that in an array of n element th first elemnt is indexed by zero and the last elemet of an array is indexed by n-1.the loop used to read the element of the array is:

```
for(int i=0,i<5;i++)
{
        cout<<"enter person "<<i+1<<"age";
cin>>age[i];
}
```

Q 4. Explain the bubble sort.

Ans. In the bubble sort, as elements are sorted they gradually "bubble" (or rise) to their proper location in the array, like bubbles rising in a glass of soda. The bubble sort repeatedly compares adjacent elements of an array. The first and second elements are compared and swapped if out of order. Then the second and third elements are compared and swapped if out of order. This sorting process continues until the last two elements of the array are compared and swapped if out of order



When this first pass through the array is complete, the bubble sort returns to elements one and two and starts the process all over again. So, when does it stop? The bubble sort knows that it is finished when it examines the entire array and no "swaps" are needed (thus the list is in proper order). The bubble sort keeps track of occurring swaps by the use of a flag.

The table below follows an array of numbers before, during, and after a bubble sort for *descending* order. A "pass" is defined as one full trip through the array comparing and if necessary, swapping, adjacent elements. Several passes have to be made through the array before it is finally sorted.

Array at beginning:	84	69	76	86	94	91
After Pass #1:	84	76	86	94	91	69
After Pass #2:	84	86	94	91	76	69
After Pass #3:	86	94	91	84	76	69
After Pass #4:	94	91	86	84	76	69
After Pass #5 (done):	94	91	86	84	76	69

The bubble sort is an easy algorithm to program, but it is slower than many other sorts. With a bubble sort, it is always necessary to make one final "pass" through the array to check to see that no swaps are made to ensure that the process is finished. In actuality, the process is finished before this last pass is made.

```
// Bubble Sort
#include<iostream.h>
#include<conio.h>
void main()
{
        int i, j, n, age[25], flag, temp;
cout<<"how many elements to sort <max-25>?";
cin>>n;
for(i=0;i<n;i++)
{
                cout<<"enter age["<<i<<"]:";
        cin>>age[i];
}
//sorting starts here
for(i=0;i<n-1;i++)
{
        flag=1;
        for(j=0;j<(n-1-i);j++)
        {
        if(age[j]>age[j+1])
        {
                flag=0;
                temp=age[j];
                age[j]=age[j+1];
```

```
age[j+1]=temp;
}
}
if(flag)
break;
}
//Sorting ends here
cout<<"sorted list…."<<endl;
for(i=0; i<n; i++)
cout<<age[i]<<"";
        getch();
}
```

Q 5. What is Multidimensional array, how is Multidimensional arrays represented in C++, how to access the elements in the Multidimensional array.

Ans: A Multidimensional Array?
A Multidimensional array is an array of arrays.
Multidimensional Arrays represented in C++
Suppose a programmer wants to represent the two-dimensional array Exforsys as an array with three rows and four columns all having integer elements. This would be represented in C++ as:
int Exforsys[3][4];
It is represented internally as:
Exforsys
Data Type: int

	0	1	2	3
0				
1				
2				

Access the elements in the Multidimensional Array
Exforsys

Data Type: int

	0	1	2	3
0				
1				
2				

Highlighted cell represent Exforsys [1][2]

Based on the above two-dimensional arrays, it is possible to handle multidimensional arrays of any number of rows and columns in C++ programming language. This is all occupied in memory. Better utilization of memory must also be made.

Multidimensional Array Example**:**

```
#include <iostream.h>
const int ROW=4;
const int COLUMN =3;
void main()
{
  int i,j;
  int Exforsys[ROW][COLUMN];
  for(i=0;i<ROWS;i++)
  for(j=0;j<COLUMN;J++)
  {
  cout << "Enter value of Row "<<i+1;
  cout<<",Column "<<j+1<<":";
  cin>>Exforsys[i][j];
  }
  cout<<"\n\n\n";
  cout<< " COLUMN\n";
  cout<< " 1 2 3";
  for(i=0;i<ROW;i++)
  {
  cout<<"\nROW "<<i+1;
  for(j=0;j<COLUMN;J++)
  cout<<Exforsys[i][j];
}
```

The output of the above program is

```
Enter value of Row 1, Column 1:10
Enter value of Row 1, Column 2:20
Enter value of Row 1, Column 3:30
Enter value of Row 2, Column 1:40
Enter value of Row 2, Column 2:50
Enter value of Row 2, Column 3:60
Enter value of Row 3, Column 1:70
Enter value of Row 3, Column 2:80
Enter value of Row 3, Column 3:90
Enter value of Row 4, Column 1:100
Enter value of Row 4, Column 2:110
Enter value of Row 4, Column 3:120
```

Pointers in C++

Q 1. Explain Pointer in C++?

Ans: C++ pointers are easy and fun to learn. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined:

```
#include <iostream>
Void  main ()
{
  int  var1;
  char var2[10];

  cout << "Address of var1 variable: ";
  cout << &var1 << endl;

  cout << "Address of var2 variable: ";
  cout << &var2 << endl;

}
```

When the above code is compiled and executed, it produces result something as follows:

Address of var1 variable: 0xbfebd5c0

Address of var2 variable: 0xbfebd5b6

Q 2. What is pointer arithmetic's?

Ans: The C language allows you to perform integer addition or subtraction operations on pointers. If pnPtr points to an integer, pnPtr + 1 is the address of the next integer in memory after pnPtr. pnPtr - 1 is the address of the previous integer before pnPtr.

Note that pnPtr+1 does not return the *address* after pnPtr, but the *next object of the type* that pnPtr points to. If pnPtr points to an integer (assuming 4 bytes), pnPtr+3 means 3 integers after pnPtr, which is 12 addresses after pnPtr. If pnPtr points to a char, which is always 1 byte, pnPtr+3 means 3 chars after pnPtr, which is 3 addresses after pnPtr.

When calculating the result of a pointer arithmetic expression, the compiler always multiplies the integer operand by the size of the object being pointed to. This is called **scaling**.

The following program:

```
1int nValue = 7;
2int *pnPtr = &nValue;
3
4cout << pnPtr << endl;
5cout << pnPtr+1 << endl;
6cout << pnPtr+2 << endl;
7cout << pnPtr+3 << endl;
```

Outputs:

```
0012FF7C
0012FF80
0012FF84
0012FF88
```

As you can see, each of these addresses differs by 4 (7C + 4 = 80 in hexadecimal). This is because an integer is 4 bytes on the author's machine.

The same program using short instead of int:

```
1short nValue = 7;
2short *pnPtr = &nValue;
3
4cout << pnPtr << endl;
5cout << pnPtr+1 << endl;
6cout << pnPtr+2 << endl;
7cout << pnPtr+3 << endl;
```

Outputs:
0012FF7C
0012FF7E
0012FF80
0012FF82
Because a short is 2 bytes, each address differs by 2.
It is rare to see the + and – operator used in such a manner with pointers. However, it is more common to see the ++ or — operator being used to increment or decrement a pointer to point to the next or previous element in an array.

Q 3. Explain the process of Pointers and structure in C++ ?

Ans: The interaction of pointers and arrays can be confusing but here are two fundamental statements about it:

- A variable declared as an array of some type (e.g. int) acts as a pointer to that type. It points to the first element of the array.
- A pointer can be used like an array name.

Like any other type, structures can be pointed by its own type of pointers:

```
1 struct movies_t {
2   string title;
3   int year;
4 };
5
6 movies_t amovie;
7 movies_t * pmovie;
```

Here amovie is an object of structure type movies_t, and pmovie is a pointer to point to objects of structure type movies_t. So, the following code would also be valid:

```
pmovie = &amovie;
```

The value of the pointer pmovie would be assigned to a reference to the object amovie (its memory address).

We will now go with another example that includes pointers, which will serve to introduce a new operator: the arrow operator (->):

```
// pointers to structures
#include <iostream>
#include <string>
#include <sstream>

struct movies_t {
  string title;
  int year;
};

void main ()
{
  string mystr;

  movies_t amovie;
  movies_t * pmovie;
  pmovie = &amovie;

  cout << "Enter title: ";
  getline (cin, pmovie->title);
  cout << "Enter year: ";
  getline (cin, mystr);
  (stringstream) mystr >> pmovie->year;

  cout << "\nYou have entered:\n";
  cout << pmovie->title;
  cout << " (" << pmovie->year << ")\n";


}
```

The previous code includes an important introduction: the arrow operator (->). This is a dereference operator that is used exclusively with pointers to objects

with members. This operator serves to access a member of an object to which we have a reference. In the example we used:

pmovie->title

Which is for all purposes equivalent to:

(*pmovie).title

Both expressions pmovie->title and (*pmovie).title are valid and both mean that we are evaluating the member title of the data structure **pointed by** a pointer called pmovie. It must be clearly differentiated from:

*pmovie.title

which is equivalent to:

*(pmovie.title)

Q 4. Explain the procedure to define the pointers to functions in C++?

Ans: C++ allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

```
#include <iostream>
#include <ctime>
void getSeconds(unsigned long *par);

void main ()
{
  unsigned long sec;
```

```
   getSeconds( &sec );

   // print the actual value
   cout << "Number of seconds :" << sec << endl;

}

void getSeconds(unsigned long *par)
{
   // get the current number of seconds
   *par = time( NULL );
   return;

}
```

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

Number of seconds: 1294450468

Q 5. Explain return pointer from functions in C++

Ans: To declare a function returning a pointer as in the following example:

```
int * myFunction()
{
.
.
.

}
```

Second point to remember is that, it is not good idea to return the address of a local variable to outside of the function so you would have to define the local variable as **static** variable.

Now consider the following function which will generate 10 random numbers and return them using an array name which represents a pointer i.e. address of first array element.

```
 #include <iostream>
#include <ctime>

// function to generate and retrun random numbers.
int * getRandom( )
{
  static int  r[10];

  // set the seed
  srand( (unsigned)time( NULL ) );
  for (int i = 0; i < 10; ++i)
  {
    r[i] = rand();
    cout << r[i] << endl;
  }

  return r;
}

// main function to call above defined function.
void main ()
{
   // a pointer to an int.
   int *p;

   p = getRandom();
   for ( int i = 0; i < 10; i++ )
   {
      cout << "*(p + " << i << ") : ";
      cout << *(p + i) << endl;
   }

   }
```

When the above code is compiled together and executed, it produces result something as follows:

```
624723190
1468735695
807113585
976495677
613357504
1377296355
1530315259
1778906708
1820354158
667126415
*(p + 0) : 624723190
*(p + 1) : 1468735695
*(p + 2) : 807113585
*(p + 3) : 976495677
*(p + 4) : 613357504
*(p + 5) : 1377296355
*(p + 6) : 1530315259
*(p + 7) : 1778906708
*(p + 8) : 1820354158

*(p + 9) : 667126415
```

Q 6. Define Bit fields typed
Ans: Bit-fields

- Bit fields provide a mechanism to optimize memory usage by allowing to specify the exact number of bits required to store data.
- Quite useful in embedded programming like mobile phones where memory is limited.
- The declaration of bit field members follow the syntax "variable name : number of bits".
- Unnamed bit fields with width 0 are used for alignment of the next bit field to the field type boundary.

Classes and structures can contain members that occupy less storage than an integral type. These members are specified as bit fields. The syntax for bit-field *member-declarator* specification follows:

declaratoropt : constant-expression

The *declarator* is the name by which the member is accessed in the program. It must be an integral type (including enumerated types). The *constant-expression* specifies the number of bits the member occupies in the structure. Anonymous bit fields — that is, bit-field members with no identifier — can be used for padding.

Note An unnamed bit field of width 0 forces alignment of the next bit field to the next *type* boundary, where *type* is the type of the member.

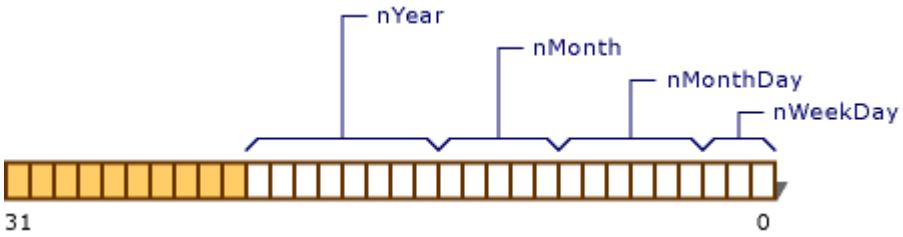
The following example declares a structure that contains bit fields:

```
// bit_fields1.cpp
struct Date
{
  unsigned nWeekDay  : 3;   // 0..7   (3 bits)
  unsigned nMonthDay : 6;   // 0..31  (6 bits)
  unsigned nMonth    : 5;   // 0..12  (5 bits)
  unsigned nYear     : 8;   // 0..100 (8 bits)
};

void main()
{
}
```

The conceptual memory layout of an object of type Date is shown in the following figure.

Memory Layout of Date Object



Note that nYear is 8 bits long and would overflow the word boundary of the declared type, **unsigned int**. Therefore, it is begun at the beginning of a new **unsigned int**. It is not necessary that all bit fields fit in one object of the underlying type; new units of storage are allocated, according to the number of bits requested in the declaration.

Q 7. Define enumerations in C++?

Ans: An enumeration is a user-defined type consisting of a set of named constants called enumerators.

```
enum [tag] {enum-list} [declarator];   // for definition of enumerated type
enum tag declarator;   // for declaration of variable of type tag
```

By default, the first enumerator has a value of 0, and each successive enumerator is one larger than the value of the previous one, unless you explicitly specify a value for a particular enumerator. Enumerators needn't have unique values within an enumeration. The name of each enumerator is treated as a constant and must be unique within the scope where the **enum** is defined. An enumerator can be promoted to an integer value. However, converting an integer to an enumerator requires an explicit cast, and the results are not defined if the integer value is outside the range of the defined enumeration.

```
#include <iostream.h>

void main()

{
```

```
 enum{ monday, tuesday, wednesday, thursday, friday, saturday, sunday }
day;
 day = wednesday;
 if(day == saturday || day == sunday)
     cout << "Day is a weekend day" << endl;
  else if(day == wednesday)
     cout << "Day is hump day - middle of the work week" << endl;
}
```

Classes & Object

Q 1. Explain Classes and member functions in C++ with example?

Ans: A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

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, that can be either data or function declarations, and optionally access specifiers.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access. For example:

```
class CRectangle
{
   int x, y;
```

```
  public:
      void set_values (int,int);
      int area (void);
} rect;
```

Declares a class (i.e., a type) called CRectangle and an object (i.e., a variable) of this class called rect. 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 level) and two member functions with public access: set_values() and area(), of which for now we have only included their declaration, not their definition.

Q 2. How to define an object of a class?

Ans: 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.

Just as we declare a variable of data type int as:

int x;

Objects are also declared as:

class_name followed_by object_name;

for example we create employee class and we create employee class object as

employee e1; in this e1 is a object of employee class.

Q 3. Explain Member function in C++/

Ans: Member functions

Member functions are operators and functions that are declared as members of a class. Member functions do not include operators and functions declared with the friend specifier. These are called friends of a class. You can declare a member function as static; this is called a static member function. A member function that is not declared as static is called a nonstatic member function.

The definition of a member function is within the scope of its enclosing class. The body of a member function is analyzed after the class declaration so that members of that class can be used in the member function body, even if the member function definition appears before the declaration of that member in the class member list. When the function add() is called in the following example, the data variables a, b, and c can be used in the body of add().

```
class student
{
public:
    int add()
    {return a+b+c;};
private:
    int a,b,c;
};
```

Q 4. Create a program which shows arrays of class objects concept?

Ans:

```
#include <iostream>
class DemoClass {
  int x;
public:
```

```
  void setX(int i) { x = i; }
  int getX() { return x; }
};

void main()
{
  DemoClass obs[4];
  int i;

  for(i=0; i < 4; i++)
    obs[i].setX(i);

  for(i=0; i < 4; i++)
    cout << "obs[" << i << "].getX(): " << obs[i].getX() << "\n";


}
```

Output

```
obs[0].getX(): 0
obs[1].getX(): 1
obs[2].getX(): 2
obs[3].getX(): 3
```

Q 5. How to use Pointer to classes in C++?

Ans: A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class you use the member access operator **->** operator, just as you do with pointers to structures. Also as with all pointers, you must initialize the pointer before using it.

```
#include <iostream>
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;
    }
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

void main(void)
{
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2
  Box *ptrBox;                // Declare pointer to a class.

  // Save the address of first object
  ptrBox = &Box1;

  // Now try to access a member using member access operator
  cout << "Volume of Box1: " << ptrBox->Volume() << endl;

  // Save the address of first object
  ptrBox = &Box2;

  // Now try to access a member using member access operator
  cout << "Volume of Box2: " << ptrBox->Volume() << endl;


}
```

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

Constructor called.
Constructor called.
Volume of Box1: 5.94
Volume of Box2: 102

Q 6. Explain the Nested classes in c++.

Ans: **Nested class** is a class defined inside a class, that can be used within the scope of the class in which it is defined. In C++ nested classes are not given importance because of the strong and flexible usage of inheritance. Its objects are accessed using "Nest::Display".

Example:

```
#include <iostream.h>
class Nest
 {
   public:
    class Display
     {
      private:
       int s;
      public:
       void sum( int a, int b)
         {  s =a+b; }
       void show( )
         { cout << "\nSum of a and b is:: " << s;}
     };
 };
void main()
 {
    Nest::Display x;
    x.sum(12, 10);
    x.show();
 }
```

Result:

Sum of a and b is::22

In the above example, the nested class "Display" is given as "public" member of the class "Nest".

Constructors & Destructors in C++

Q 1. Define constructor in C++?

Ans: a constructor is a method in the class which gets executed when its object is created. Constructors are special methods, used when instantiating a class. A constructor can never return anything, which is why you don't have to define a return type for it. A normal method is defined like this:
public string Describe()
A constructor can be defined like this:
public Car()
we have a Car class, with a constructor which takes a string as argument. Of course, a constructor can be overloaded as well, meaning we can have several constructors, with the same name, but different parameters. Here is an example:

```
public Car()
{

}

public Car(string color)
{
   this.color = color;
}
```

A constructor can call another constructor, which can come in handy in several situations. Here is an example:

```
public Car()
{
   Cout<<"Constructor with no parameters called!";
}

public Car(string color) : this()
{
   this.color = color;
   cout<<"Constructor with color parameter called!";
}
```

If you run this code, you will see that the constructor with no parameters is called first. This can be used for instantiating various objects for the class in the default constructor, which can be called from other constructors from the class.

Q 2. Explain Types of constructor?

Ans: There are following types of constructor as-

1.Parameterized constructors

Constructors that can take arguments are termed as parameterized constructors. The number of arguments can be greater or equal to one.

```
class example
{
    int p, q;
  public:
    example(int a, int b);                    //parameterized constructor
};
example :: example(int a, int b)
{
    p = a;
    q = b;
}
```

2. Default constructors

If the programmer does not supply a constructor for an insatiable class, a typical compiler will provide a *default constructor*. The behavior of the default constructor is language dependent. It may initialize data members to zero or other same values, or it may do nothing at all.

3. **Copy constructors**

Copy constructor defines the actions performed by the compiler when copying class objects. A copy constructor has one formal parameter that is the type of the class (the parameter may be a reference to an object).

4. **Conversion constructors**

Conversion constructors provide a means for a compiler to implicitly create an object belonging to one class based on an object of a different type. These constructors are usually invoked implicitly to convert arguments or operands to an appropriate type, but they may also be called explicitly.

Q 3. Explain Default constructor with example?

Ans: **Default Constructor:** A constructor without any parameters is called as default constructor. Drawback of default constructor is every instance of the class will be initialized to same values and it is not possible to initialize each instance of the class to different values.

What this means is that if no constructor is specified, a hidden one is automatically generated for you. You can call it anywhere. The constructor is a public parameterless instance constructor.

Definition in short: A **Default constructor** is that will either have no parameters, or all the parameters have default values. If no constructors are available for a class, the compiler implicitly creates a default parameterless constructor without a constructor initializer and a null body.

Example:

```
#include <iostream.h>
class Defal
 {
   public:
    int x;
    int y;
    Defal(){x=y=0;}
 };
void main()
 {
   Defal A;
   cout << "Default constructs x,y value::"<<
       A.x <<" , "<< A.y << "\n";
```

```
    return 0;
}
```

Result:

Default constructs x,y value:: 0,0

In the above example a default constructor has the default value of "0" for both the parameters.

Q 4. Explain Friend Function?

Ans: A function outside of a class can be defined to be a friend function by the class which gives the friend function free access to the private or protected members of the class. This is done by preceding the function prototype in the class declaration with keyword friend. For example:

```
private:
friend void set(int new_length, int new_width);
//friend method
friend int get_area(void) {return (length * width);}
//friend method
```

So, set() and get_area() functions still can be used to access members of the class. This in effect, opens a small hole in the protective shield of the class, so it should be used very carefully.

A single isolated function can be declared as a friend, as well as members of other classes, and even entire classes can be given friend status if needed in a program. Neither a constructor nor a destructor can be a friend function.

Q 5. Explain Inline member function with suitable example?

Ans: A function defined in the body of a class declaration is an inline function.the following class declaration, the Account constructor is an inline function. The member functionsGetBalance, Deposit, and Withdraw are not specified as **inline** but can be implemented as inline functions.

```
// Inline_Member_Functions.cpp
```

```
class Account
{
public:
   Account(double initial_balance) { balance = initial_balance; }
   double GetBalance();
   double Deposit( double Amount );
   double Withdraw( double Amount );
private:
   double balance;
};

inline double Account::GetBalance()
{
   return balance;
}

inline double Account::Deposit( double Amount )
{
   return ( balance += Amount );
}

inline double Account::Withdraw( double Amount )
{
   return ( balance -= Amount );
}
void main()
{
}
```

Note:
In the class declaration, the functions were declared without the inline keyword. the inline keyword can be specified in the class declaration ,the result is the same.

Q 6. What is Destructor?

Ans: Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.

A destructor is a member function with the same name as its class prefixed by a ~ (tilde).

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.

Q 7. How to destroy objects in C++?

Ans: **Destructors** are a type of member functions used to destroy the objects of a class created by a "Constructor". The destructors have the same name as the class whose objects are intialized but with a "~" or "tilde" symbol preceding the destructor declaration.

"Destructors" dont take any arguments or neither pass any values. But it is used to free the space used by the program. The C++ compiler calls the destructor implicitly when a program execution is exited.

Example:

```
#include <iostream.h>
int cnt=0;
```

```
class display
 {
   public:
     display()
      {
            cnt++;
            cout << "\nCreate Object::" << cnt;
      }
      ~display()
      {
            cout << "\nDestroyed Object::" << cnt;
        cnt--;
      }
 };
void main()
 {
   cout << "\nMain Objects x,y,z\n";
   display x,y,z;
     {
                        cout << "\n\nNew object 4\n";
            display B;
     }
   cout << "\n\nDestroy All objects x,y,z\n";
   return 0;
}
```

Result:

```
Main Objects x,y,z

Create Object::1
Create Object::2
Create Object::3

New objects 4
```

```
Create Object::4
Destroyed Object::4

Destroy All objects x,y,z
Destroyed Object::3
Destroyed Object::2
Destroyed Object::1
```

In the above example both a constructor "display()", destructor "~display()" is used. First three objects x,y,z are created, then a fourth object is created inside "{}". The fourth object is destroyed implicitly when the code execution goes out of scope defined by braces. Finally all the exisiting objects are destroyed.

Q 8. Write a program which is defines Constructor or Destructor?

Ans.

```
#include<iostream.h>
#include<conio.h>
class stu
{
        private: char name[20],add[20];
                 int roll,zip;
        public: stu ( );//Constructor
                 ~stu( );//Destructor
                 void read( );
                 void disp( );
};
stu :: stu( )
{
        cout<<"This is Student Details"<<endl;
}
void stu :: read( )
{
        cout<<"Enter the student Name";
```

```
        cin>>name;
        cout<<"Enter the student roll no ";
        cin>>roll;
        cout<<"Enter the student address";
        cin>>add;
        cout<<"Enter the Zipcode";
        cin>>zip;
}
void stu :: disp( )
{
        cout<<"Student Name :"<<name<<endl;
        cout<<"Roll no   is      :"<<roll<<endl;
        cout<<"Address is       :"<<add<<endl;
        cout<<"Zipcode is       :"<<zip;
}
stu : : ~stu( )
{
        cout<<"Student Detail is Closed";
}

void main( )
{
        stu s;
        clrscr( );
s.read ( );
s.disp ( );
getch( );
}
```

Q 9. Explain Constructor Overloading with example?

Ans: **Constructors Overloading** are used to increase the flexibility of a class by having more number of constructor for a single class. By have more than one way of initializing objects can be done using overloading constructors.

Example:

```
#include <iostream.h>
```

```
class Overclass
 {
            public:
            int x;
            int y;
   Overclass() { x = y = 0; }
            Overclass(int a) { x = y = a; }
            Overclass(int a, int b) { x = a; y = b; }
 };
void main()
 {
             Overclass A;
   Overclass A1(4);
   Overclass A2(8, 12);
   cout << "Overclass A's x,y value:: " <<
        A.x  << " , "<< A.y << "\n";
   cout << "Overclass A1's x,y value:: "<<
        A1.x << " ,"<< A1.y << "\n";
   cout << "Overclass A2's x,y value:; "<<
        A2.x << " , "<< A2.y << "\n";
   return 0;
}
```

Result:

```
Overclass A's x,y value:: 0 , 0
Overclass A1's x,y value:: 4 ,4
Overclass A2's x,y value:; 8 , 12
```

In the above example the constructor "Overclass" is overloaded thrice with different intialized values.

Inheritance & Types of inheritance

Q 1. What is Base class and Derived class?

Ans: Base & Derived Classes:

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

class derived-class: access-specifier base-class

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows:

```
#include <iostream>

using namespace std;

// Base class
class Shape
{
  public:
    void setWidth(int w)
    {
      width = w;
    }
    void setHeight(int h)
    {
      height = h;
    }
  protected:
```

```
    int width;
    int height;
};

// Derived class
class Rectangle: public Shape
{
  public:
    int getArea()
    {
      return (width * height);
    }
};

int main(void)
{
  Rectangle Rect;

  Rect.setWidth(5);
  Rect.setHeight(7);

  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;

  return 0;

}
```

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

```
Total area: 35
```

Q 2. What is Access Control and Inheritance?

Ans: Access Control and Inheritance:

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Q 3. What is inherited?

Ans: When inheritance is done, various links and tables (index, virtual etc) are created which are used to provide the accessibility of the members of the base class in derived class and in other class hierarchy. This means saying **"public members are inherited"** is better to say as **"public members become accessible"**.

A derived class inherits every member of a base class except:

- its constructor and its destructor
- its friends
- its operator=() members

Q 4. Explain the various types of Inheritance available in c++?

Ans: When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

1. **Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's**private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
2. **Protected Inheritance:** When deriving from a **protected** base class, **public** and**protected** members of the base class become **protected** members of the derived class.
3. **Private Inheritance:** When deriving from a **private** base class, **public** and **protected**members of the base class become **private** members of the derived class.

The Various Types of Inheritance those are provided by C++ are as followings:

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

In Inheritance Upper Class whose code we are actually inheriting is known as the Base or Super Class and Class which uses the Code are known as Derived or Sub Class.

1) In **Single Inheritance** there is only one Super Class and Only one Sub Class Means they have one to one Communication between them

2) In **Multilevel Inheritance** a Derived class can also inherited by another class Means in this When a Derived Class again will be inherited by another Class then it creates a Multiple Levels.

3) **Multiple Inheritances** is that in which a Class inherits the features from two Base Classes When a Derived Class takes Features from two Base Classes.

4) **Hierarchical Inheritance** is that in which a Base Class has Many Sub Classes or When a Base Class is used or inherited by many Sub Classes.

5) **Hybrid Inheritance: -** This is a Mixture of two or More Inheritance and in this Inheritance a Code May Contains two or three types of inheritance in Single Code.

Q 5. Explain Multiple Inheritances with example?

Ans: A C++ class can inherit members from more than one class and here is the extended syntax:

class derived-class: access baseA, access baseB....

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example:

```
#include <iostream>

using namespace std;

// Base class Shape
class Shape
{
  public:
    void setWidth(int w)
    {
      width = w;
    }
    void setHeight(int h)
    {
      height = h;
    }
  protected:
```

```
    int width;
    int height;
};

// Base class PaintCost
class PaintCost
{
  public:
    int getCost(int area)
    {
      return area * 70;
    }
};

// Derived class
class Rectangle: public Shape, public PaintCost
{
  public:
    int getArea()
    {
      return (width * height);
    }
};

void main()
{
  Rectangle Rect;
  int area;

  Rect.setWidth(5);
  Rect.setHeight(7);

  area = Rect.getArea();

  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;
```

```
  // Print the total cost of painting
  cout << "Total paint cost: $" << Rect.getCost(area) << endl;

}
```

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

Total area: 35

Total paint cost: $2450

Q 6. Explain payroll system by using single inheritance?

Ans:

```
 #include<iostream.h>
#include<conio.h>

class emp
{
  public:
    int eno;
    char name[20],des[20];
    void get()
    {
          cout<<"Enter the employee number:";
          cin>>eno;
          cout<<"Enter the employee name:";
          cin>>name;
          cout<<"Enter the designation:";
          cin>>des;
    }
};

class salary:public emp
{
    float bp,hra,da,pf,np;
```

```
  public:
   void get1()
   {
          cout<<"Enter the basic pay:";
          cin>>bp;
          cout<<"Enter the Humen Resource Allowance:";
          cin>>hra;
          cout<<"Enter the Dearness Allowance :";
          cin>>da;
          cout<<"Enter the Profitablity Fund:";
          cin>>pf;
   }
   void calculate()
   {
          np=bp+hra+da-pf;
   }
   void display()
   {

 cout<<eno<<"\t"<<name<<"\t"<<des<<"\t"<<bp<<"\t"<<hra<<"\t"<<da<<"\t
"<<pf<<"\t"<<np<<"\n";
   }
};

void main()
{
   int i,n;
   char ch;
   salary s[10];
   clrscr();
   cout<<"Enter the number of employee:";
   cin>>n;
   for(i=0;i<n;i++)
   {
          s[i].get();
```

```
            s[i].get1();
            s[i].calculate();
    }
    cout<<"\ne_no \t e_name\t des \t bp \t hra \t da \t pf \t np \n";
    for(i=0;i<n;i++)
    {
            s[i].display();
    }
    getch();
}
```

Output:

```
Enter the Number of employee:1
Enter the employee No: 150
Enter the employee Name: ram
Enter the designation: Manager
Enter the basic pay: 5000
Enter the HR allowance: 1000
Enter the Dearness allowance: 500
Enter the profitability Fund: 300

E.No  E.name  des     BP   HRA  DA   PF   NP
150   ram     Manager 5000 1000 500  300  6200
```

Q 7. Explain Multilevel Inheritance with example?

Ans: **Multilevel Inheritance** is a method where a derived class is derived from another derived class.

Example:

```
#include <iostream.h>
class mm
 {
   protected:
    int rollno;
   public:
```

```
    void get_num(int a)
     { rollno = a; }
    void put_num()
     { cout << "Roll Number Is:\n"<< rollno << "\n"; }
 };
class marks : public mm
 {
  protected:
   int sub1;
   int sub2;
  public:
   void get_marks(int x,int y)
    {
      sub1 = x;
      sub2 = y;
    }
   void put_marks(void)
    {
     cout << "Subject 1:" << sub1 << "\n";
     cout << "Subject 2:" << sub2 << "\n";
    }
 };
class res : public marks
 {
  protected:
   float tot;
  public:
   void disp(void)
     {
      tot = sub1+sub2;
      put_num();
      put_marks();
      cout << "Total:"<< tot;
     }
 };
void main()
 {
```

```
  res std1;
  std1.get_num(5);
  std1.get_marks(10,20);
  std1.disp();
  return 0;
}
```

Result:

```
Roll Number Is:
5
Subject 1: 10
Subject 2: 20
Total: 30
```

In the above example, the derived function "res" uses the function "put_num()" from another derived class "marks", which just a level above. This is the multilevel inheritance OOP's concept in C++.

Q 8. Explain Hybrid Inheritance with example?

Ans: **Hybrid Inheritance** is a method where one or more types of inheritance are combined together and used.

Example:

```
#include <iostream.h>
class mm
 {
  protected:
   int rollno;
  public:
   void get_num(int a)
    { rollno = a; }
   void put_num()
    { cout << "Roll Number Is:"<< rollno << "\n"; }
```

```
 };
class marks : public mm
 {
  protected:
    int sub1;
    int sub2;
  public:
    void get_marks(int x,int y)
     {
       sub1 = x;
       sub2 = y;
     }
    void put_marks(void)
     {
       cout << "Subject 1:" << sub1 << "\n";
       cout << "Subject 2:" << sub2 << "\n";
     }
 };

class extra
 {
   protected:
     float e;
   public:
   void get_extra(float s)
    {e=s;}
   void put_extra(void)
    { cout << "Extra Score::" << e << "\n";}
 };

class res : public marks, public extra{
  protected:
    float tot;
  public:
    void disp(void)
     {
       tot = sub1+sub2+e;
```

```
        put_num();
        put_marks();
        put_extra();
        cout << "Total:"<< tot;
      }
};

void main()
{
  res std1;
  std1.get_num(10);
  std1.get_marks(10,20);
  std1.get_extra(33.12);
  std1.disp();
  return 0;
}
```

Result:

```
  Roll Number Is: 10
  Subject 1: 10
  Subject 2: 20
  Extra score:33.12
  Total: 63.12
```

In the above example the derived class "res" uses the function "put_num()". Here the "put_num()" function is derived first to class "marks". Then it is deriived and used in class "res". This is an example of "multilevel inheritance-OOP's concept". But the class "extra" is inherited a single time in the class "res", an example for "Single Inheritance". Since this code uses both "multilevel" and "single" inheritence it is an example of "Hybrid Inheritance".

Q 9. What is Polymorphism and explain run and compile time polymorphism?

Ans: **Polymorphism** is the ability of an object or reference to take many different forms at different instances. These are of two types one is the "compile time polymorphism" and other one is the "run-time polymorphism".

Compile time polymorphism:

In this method object is bound to the function call at the compile time itself.

Run time polymorphism:

In this method object is bound to the function call only at the run time.

Example:

```
#include <iostream.h>
class Value
 {
   protected:
   int val;
   public:
   void set_values (int a)
     { val=a;}
 };
class Cube: public Value
 {
 public:
 int cube()
   { return (val*val*val); }
 };
int main () {
 Cube cb;
 Value * ptr = &cb;
 ptr->set_values (10);
 cout << "The cube of 10 is::" << cb.cube() << endl;
 return 0;
}
```

Result:

The cube of 10 is:: 1000

In the above OOPs example "Cube" is a derived class of "Value". To implement polymorphism a pointer "ptr" is used to reference to the members of the class "Cube". This is an example for "Compile time polymorphism."

Overloading

Q.1 What is overloading?

Ans: An *overloaded declaration* is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different types.

If you call an overloaded function name or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called *overload resolution*,

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

Q 2. Different types of operator overloading?

Ans: 1. Arithmetic Operator Overloading

Syntax:

Declairation:-

```
Class_Name Operator arithmetic_operator (ClassName);
```

Defination:-

```
Class_Name Operator arithmetic_operator (ClassName obj) {
  Assign each member of obj to the corresponding member of the owner.

  return obj;
}
```

2. Assignment Operator Overloading

The general form of assignment operator overloading is as follows-

Syntax

Declairation:-

```
Class_Name& Operator= (ClassName&);
```

Defination:-

```
Class_Name& Operator= (ClassName& obj) {
  Assign each member of obj to the corresponding member of the owner.

  return *this;
}
```

3. Arithmetic Assignment Operator Overloading

In C++ has five arithmetic assignment operator they are- plus and equal to(+=), minus and equal to(-=) multiply and equal to(*=), divide and equal to(/=) and modulus and queal to(%=). the general form of arithmetic assignment operator overloading is as follows-

Syntax

Declairation:-

```
Class_Name& Operator arithmetic_assignment_operator (ClassName);
```

Defination:-

```
Class_Name& Operator arithmetic_assignment_operator (ClassName obj) {
  Assign each member of obj to the corresponding member of the owner.

  return *this;
}
```

Q 3. What operator overloading explain with example?

Ans: Operators overloading :In Operator overloading we can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Box operator+(const Box&);

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows:

Box operator+(const Box&, const Box&);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below:

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

class Box
{
  public:

    double getVolume(void)
    {
      return length * breadth * height;
    }
    void setLength( double len )
    {
```

```
      length = len;
   }

   void setBreadth( double bre )
   {
      breadth = bre;
   }

   void setHeight( double hei )
   {
      height = hei;
   }
   // Overload + operator to add two Box objects.
   Box operator+(const Box& b)
   {
     Box box;
     box.length = this->length + b.length;
     box.breadth = this->breadth + b.breadth;
     box.height = this->height + b.height;
     return box;
   }
 private:
   double length;     // Length of a box
   double breadth;    // Breadth of a box
   double height;     // Height of a box
};
// Main function for the program
void main( )
{
  Box Box1;          // Declare Box1 of type Box
  Box Box2;          // Declare Box2 of type Box
  Box Box3;          // Declare Box3 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;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

}
```

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

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

Volume of Box3 : 5400
```

Q 4. Explain Function overloading with example?

Ans: Function overloading :

We can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or

the number of arguments in the argument list. we can not overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types:

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

class printData
{
  public:
    void print(int i) {
      cout << "Printing int: " << i << endl;
    }

    void print(double  f) {
      cout << "Printing float: " << f << endl;
    }

    void print(char* c) {
      cout << "Printing character: " << c << endl;
    }
};

void main()
{
  printData pd;

  // Call print to print integer
  pd.print(5);
  // Call print to print float
  pd.print(500.263);
  // Call print to print character
  pd.print("Hello C++");
```

```
}
```

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

```
Printing int: 5
Printing float: 500.263

Printing character: Hello C++
```

Q 5. What is unary operator overloading ?

Ans: The unary operators operate on a single operand and following are the examples of Unary operators:

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

```
#include <iostream>
class Distance
{
  private:
    int feet;          // 0 to infinite
    int inches;        // 0 to 12
  public:
    // required constructors
    Distance(){
      feet = 0;
      inches = 0;
    }
```

```
    Distance(int f, int i){
      feet = f;
      inches = i;
    }
    // method to display distance
    void displayDistance()
    {
      cout << "F: " << feet << " I:" << inches <<endl;
    }
    // overloaded minus (-) operator
    Distance operator- ()
    {
      feet = -feet;
      inches = -inches;
      return Distance(feet, inches);
    }
};
void main()
{
  Distance D1(11, 10), D2(-5, 11);

  -D1;                  // apply negation
  D1.displayDistance();   // display D1

  -D2;                  // apply negation
  D2.displayDistance();   // display D2
}
```

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

F: -11 I:-10

F: 5 I:-11

Q 6. What is polymorphism?

Ans: *polymorphism* means that some code or operations or objects behave differently in different contexts.

Polymorphism is the ability to use an operator or method in different ways. Polymorphism gives different meanings or functions to the operators or methods. Poly, referring to many, signifies the many uses of these operators and methods. A single method usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts.

For example, the + (plus) operator in C++:

```
4 + 5       <-- integer addition
3.14 + 2.0  <-- floating point addition
s1 + "bar"  <-- string concatenation!
```

In C++, that type of polymorphism is called *overloading*.

Q 7. What are the Features and Advantages of the concept of Polymorphism?

Ans: Applications are Easily Extendable:

Once an application is written using the concept of polymorphism, it can easily be extended, providing new objects that conform to the original interface. It is unnecessary to recompile original programs by adding new types. Only re-linking is necessary to exhibit the new changes along with the old application. This is the greatest achievement of C++ object-oriented programming. In programming language, there has always been a need for adding and customizing. By utilizing the concept of polymorphism, time and work effort is reduced in addition to making future maintenance easier.

- Helps in reusability of code.
- Provides easier maintenance of applications.
- Helps in achieving robustness in applications.

Q 8. Explain different Types of Polymorphism?

Ans: C++ provides three different types of polymorphism.

- Virtual functions
- Function name overloading
- Operator overloading

In addition to the above three types of polymorphism, there exist other kinds of polymorphism:

- run-time
- compile-time
- ad-hoc polymorphism
- parametric polymorphism

Other types of polymorphism defined:

run-time:
The run-time polymorphism is implemented with inheritance and virtual functions.

compile-time:
The compile-time polymorphism is implemented with templates.

ad-hoc polymorphism:
If the range of actual types that can be used is finite and the combinations must be individually specified prior to use, this is called ad-hoc polymorphism.

parametric polymorphism:
If all code is written without mention of any specific type and thus can be used transparently with any number of new types it is called parametric polymorphism

Q 9. What early and late binding

Ans: Early Binding:-

Early Binding reffers to the evernts that occours at compile time. Early binding occours when the information needed to call a function is known at the compile time. Differently, early binding means that an object and a function call bound during compilation. Early binding is accounplished by function overloading and operator overloading. The main advantages of early binding is efficiency. Since all information needed to to call a function is determine at the compile time. these type of function calls are very fast.

Late Binding:-

Late Binding reffers to the function call that are not resource untill run time. Virtual function are used to achived late binding. When access is done via base class pointer, is determine by the type of object poited to by the pointer. Since in most cases it can not be determine at the compile time, the objects an function are not linked untill run time. The main advantage of late binding is flexibility. Late binding is slower than early binding.

Q 10. What is Function Overriding?
Ans: Function Overriding

When a function defined in the base class is redefined in the derived class to fit its own needs in assense, is called function overriding. Overriding function in the derived classes have the same name and argument types as the function in the base class but the defination is different. In C++, function overriding is achived by using virtual function where the declairation of the function in the base class to be overriding is precided by the keyword virtual.

Q 11. Explain Virtual Function?
Ans: In object-oriented programming, a virtual function or virtual method is a function or method whose behavior can be overridden within an inheriting

class by a function with the same signature. This concept is a very important part of the polymorphism portion of object-oriented programming (OOP).
Used to support polymorphism with pointers and references.Declared virtual in a base class can be overridden in derived class. Overriding only happens when signatures are the same otherwise it just overloads the function or operator name. Ensures derived class function definition is resolved dynamically. E.g., those destructors farther down the hierarchy get called

```
class A {
public:
 void x() {cout<<"A:x";};
  virtual void y() {cout<<"A:y";};
};
class B : public A {
public:
void x() {cout<<"B:x";};
  virtual void y() {cout<<"B:y";};
};
int main () {
   B b;
   A *ap = &b; B *bp = &b;
   b.x (); // prints "B:x"
 b.y (); // prints "B:y"
   bp->x (); // prints "B:x"
   bp->y (); // prints "B:y"
   ap.x (); // prints "A:x"
   ap.y (); // prints "B:y"
   return 0;
};
```

Q 12. Explain Pure Virtual Function?

Ans: A virtual function or virtual method is a function or method whose behavior can be overridden within an inheriting class by a function with the same signature. This concept is a very important part of the polymorphism portion of object-oriented programming (OOP)

Syntax:

virtual return_type function_name(argument_list) = 0;

A is an Abstract Base Class Similar to an interface in Java Declares pure virtual functions (=0)
Derived classes override pure virtual methods
B overrides x(), C overrides y()
Can't instantiate class with declared or inherited pure virtual functions
A and B are abstract, can create a C
Can still have a pointer to an abstract class type Useful for polymorphism
For example:

```
class A {
public:
  virtual void x() = 0;
  virtual void y() = 0;
};
class B : public A {
public:
  virtual void x();
};
class C : public B {
public:
  virtual void y();
};
int main () {
  A * ap = new C;
  ap->x ();
  ap->y ();
  delete ap;
  return 0;
};
```

Q 13. What is Abstract class?

Ans: Abstract Base Class

An abstract base class is a class that has one or more than one virtual function. A pure virtual function that has no implimentation in its own class. A abstract base class can not be instantiated. This means that we can not create any object of an abstract base class. This existance of a pure virtual function in a class requires that every one of its concrite derived sub classes impliment the functions. A concrite derived class is a class that does not have any pure virtual function.

MCQ'S

Q.1 In case of arguments passed by values when calling a function such as z=addidion(x,y),

a.Any modifications to the variables x & y from inside the function will not have any effect outside the function.

b.The variables x and y will be updated when any modification is done in the function

c.The variables x and y are passed to the function addition

d.None of above are valid.

Ans. *a. Any modifications to the variables x & y from inside the function will not have any effect outside the function*

Q.2. In case of pass by reference

a.The values of those variables are passed to the function so that it can manipulate them

b.The location of variable in memory is passed to the function so that it can use the same memory area for its processing

c.The function declaration should contain ampersand (& in its type declaration

d. All of above

Ans. *b. The location of variable in memory is passed to the function so that it can use the same memory area for its processing*

Q.3 Overloaded functions are

a.Very long functions that can hardly run

b. One function containing another one or more functions inside it.

c. Two or more functions with the same name but different number of parameters or type.

d. None of above

Ans. *d. None of above*

Q.4 What is the correct value to return to the operating system upon the successful completion of a program?

A. -1

B. 1

C. 0

D. Programs do not return a value.

Ans. *C. 0*

Q.5 What is the only function all C++ programs must contain?
A. start()
B. system()
C. main()
D. program()

Ans. *main()*

Q.6 What punctuation is used to signal the beginning and end of code blocks?
A. { }
B. -> and <-
C. BEGIN and END
D. (and)

Ans. *{ }*

Q.7 What punctuation ends most lines of C++ code?
A. . (dot)
B. ; (semi-colon)
C. : (colon)
D. ' (single quote)

Ans. *; (semi-colon)*

Q.8 Which of the following is a correct comment?
A. */ Comments */
B. ** Comment **
C. /* Comment */
D. { Comment }

Ans. */* Comment */*

Q.9 Which of the following is not a correct variable type?

A. float
B. real
C. int
D. double

Ans. *real*

Q.10 Which of the following is the correct operator to compare two variables?

A. :=
B. =
C. equal
D. ==

Ans. ==

Keywords

Early Binding:-

Early Binding reffers to the evernts that occours at compile time. Early binding occours when the information needed to call a function is known at the compile time. Differently, early binding means that an object and a function call bound during compilation. Early binding is accounplished by function overloading and operator overloading. The main advantages of early binding is efficiency. Since all information needed to to call a function is determine at the compile time. these type of function calls are very fast.

Late Binding:-

Late Binding reffers to the function call that are not resource untill run time. Virtual function are used to achived late binding. When access is done via base class pointer, is determine by the type of object poited to by the pointer. Since in most cases it can not be determine at the compile time, the objects an function are not linked untill run time. The main advantage of late binding is flexibility. Late binding is slower than early binding.

Function Overriding

When a function defined in the base class is redefined in the derived class to fit its own needs in assense, is called function overriding. Overriding function in the derived classes have the same name and argument types as the function in the base class but the defination is different. In C++, function overriding is achived by using virtual function where the declairation of the function in the base class to be overriding is precided by the keyword virtual.

Abstract Base Class

An abstract base class is a class that has one or more than one virtual function. A pure virtual function that has no implimentation in its own class. A abstract base class can not be instantiated. This means that we can not create any object of an abstract base class. This existance of a pure virtual function in a class requires that every one of its concrite derived sub classes impliment the functions. A concrite derived class is a class that does not have any pure virtual function.

enumerations in C++

An enumeration is a user-defined type consisting of a set of named constants called enumerators.

enum [tag] {enum-list} [declarator]; // for definition of enumerated type
enum tag declarator; // for declaration of variable of type tag

By default, the first enumerator has a value of 0, and each successive enumerator is one larger than the value of the previous one, unless you explicitly specify a value for a particular enumerator. Enumerators needn't have unique values within an enumeration. The name of each enumerator is treated as a constant and must be unique within the scope where the enum is defined. An enumerator can be promoted to an integer value. However, converting an integer to an enumerator requires an explicit cast, and the results are not defined if the integer value is outside the range of the defined enumeration.

enum{ monday, tuesday, wednesday, thursday, friday, saturday, sunday }

Friend Function

A function outside of a class can be defined to be a friend function by the class which gives the friend function free access to the private or protected members of the class. This is done by preceding the function prototype in the class declaration with keyword friend. For example:

```
private:
friend void set(int new_length, int new_width);
//friend method
friend int get_area(void) {return (length * width);}
//friend method
```

So, set() and get_area() functions still can be used to access members of the class. This in effect, opens a small hole in the protective shield of the class, so it should be used very carefully.

A single isolated function can be declared as a friend, as well as members of other classes, and even entire classes can be given friend status if needed in a program. Neither a constructor nor a destructor can be a friend function.

inherited

Ans:When inheritance is done, various links and tables (index, virtual etc) are created which are used to provide the accessibility of the members of the base class in derived class and in other class hierarchy. This means saying"public members are inherited" is better to say as "public members become accessible".

A derived class inherits every member of a base class except:

- its constructor and its destructor
- its friends
- its operator=() members

Bit-fields

- Bit fields provide a mechanism to optimize memory usage by allowing to specify the exact number of bits required to store data.
- Quite useful in embedded programming like mobile phones where memory is limited.
- The declaration of bit field members follow the syntax "variable name : number of bits".
- Unnamed bit fields with width 0 are used for alignment of the next bit field to the field type boundary.