What is C++

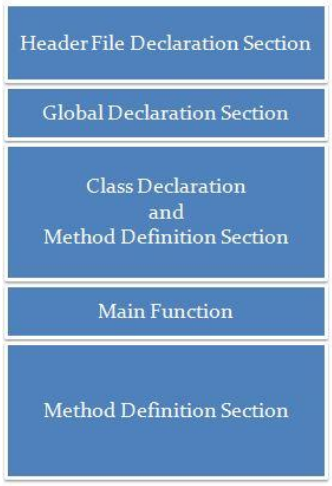
C++ is an object oriented programming language. It was developed by Bjarne Stroustrup in early 1980 at AT&T bell laboratories. Bjarne Stroustrup developed C++ by adding the feature of class so initially it was known as "C with Class". In 1983 it was given name C++. Because C++ is an incremented version of c it is called C++ using the concept of increment operator (++).

➢ *Applications of C++*

1. Games:
2. Graphic User Interface (GUI) based applications:
3. Web Browsers:
4. **Advance Computations and Graphics:**
5. Database Software:
6. **Operating Systems:**
7. Enterprise Software:
8. **Embedded (Medical and Engineering) Applications:**
9. **Compilers:**

➢ *Structure of C++ Program*

Figure: Structure of C++ Program

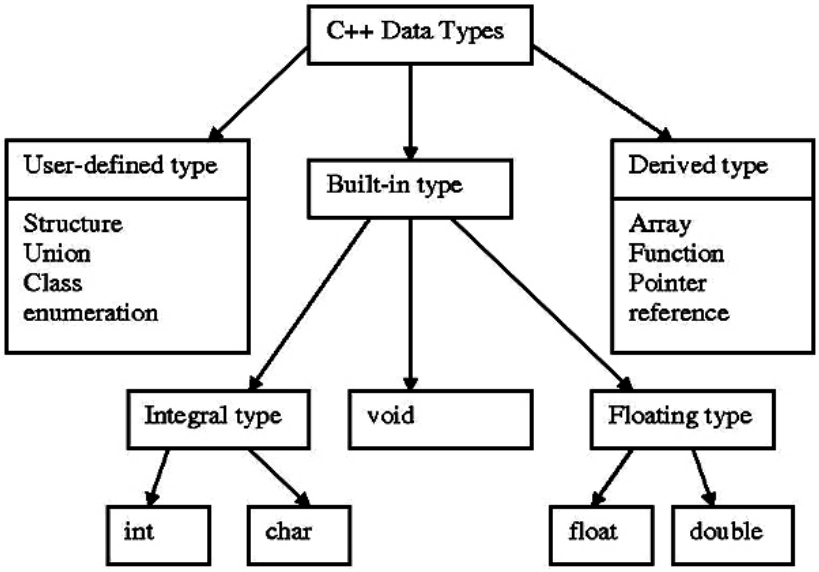


- In header file declarationsection, header files used in the program are listed here. Header File provides Prototype declaration for different library functions. We can also include user define header file. Basically all preprocessor directives are written in this section.
- In global declaration section local, variables are declared here. Global Declaration may include –Declaring Structure, Declaring Class, andDeclaring Variable.
- In class declaration Section, actuallyit can be considered as sub section for the global declaration section. Class declaration and all methods of that class are defined here.
- The main function - Each and every C++ program always starts with main function. This is entry point for all the function. Each and every method is called indirectly through main. We can create class objects in the main. Operating system calls this function automatically.
- The method definition section is optional section. Generally this method was used in C Programming.

Simple C++ Program

```
/* This is a simple C++ program.*/
#include <iostream>
using namespace std;
int main()
        {
cout<< "C++ is power programming.";
return 0;
}
```

➢ *Data Types*



Data types are used to tell the variables the type of data it can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data-type with which it is declared. Every data type requires different amount of memory.

➢ *Primitive data types available in C++.*

- **Integer**: Keyword used for integer data types is int. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.
- **Character:** Character data type is used for storing characters. Keyword used for character data type is char. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.
- **Boolean:** Boolean data type is used for storing boolean or logical values. A boolean variable can store either *true* or *false*. Keyword used for boolean data type is bool.
- **Floating Point:** Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is float. Float variables typically requires 4 byte of memory space.
- **Double Floating Point:** Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for double floating point data type is double. Double variables typically require 8 byte of memory space.
- **void** means without any value. void datatype represents a valueless entity. Void data type is used for that function which does not returns a value.
- **Wide Character:** Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype. Represented by wchar_t. It is generally 2 or 4 bytes long.

➢ *Datatype Modifiers*

As the name implies, datatype modifiers are used with the built-in data types to modify the length of data that a particular data type can hold. Data type modifiers available in C++ are:

1. Signed
2. Unsigned
3. Short
4. Long

Below table summarizes the modified size and range of built-in data types when combined with the type modifiers

Data Type	Size (bytes)	Size (bits)	Value Range
unsigned char	1	8	0 to 255
signed char	1	8	-128 to 127
char	1	8	either
unsigned short	2	16	0 to 65,535
short	2	16	-32,768 to 32,767
unsigned int	4	32	0 to 4,294,967,295
int	4	32	-2,147,483,648 to 2,147,483,647
unsigned long	8	64	0 to 18,446,744,073,709,551,616
long	8	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	8	64	0 to 18,446,744,073,709,551,616
long long	8	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	32	3.4E +/- 38 (7 digits)
double	8	64	1.7E +/- 308 (15 digits)
long double	8	64	1.7E +/- 308 (15 digits)
bool	1	8	false or true

➢ *User Defined Data Types*

C++ allows us to use one or more data types, group them as one, to create a new data type.

- ***Structures***

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

struct students
 {
char name[10];
int SSN;
char cardtype[10];
float balance;
}
```

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

A structure is one or a group of variables considered as a (custom) data type. To create a structure, use the **struct** keyword, followed by a name for the object, at least followed by a semi-colon. Therefore, fundamentally, a structure can be created as:

```
struct SomeName;
```

The struct keyword is required. The section between the curly brackets is referred to as the body of the structure. Inside of this body, you can declare at least one variable that would be used to define the structure. The variable must be declared complete with a known data type, a name and the ending semi-colon. Any variable you declare within the body of the structure is referred to as a member of that structure.

• *Classes*

In object-oriented programming language C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an object. A class is an extended concept similar to that of structure this class describes the data properties alone. The class describes both the properties (data) and behaviors (functions) of objects. Classes are not objects, but they are used to instantiate objects. Classes contain data known as members and member functions. As a unit, the collection of members and member functions is an object. Therefore, this unit of objects makes up a class.

The starting flower brace symbol {is placed at the beginning of the code. Following the flower brace symbol, the body of the class is defined with the member functions data. Then the class is closed with a flower brace symbol} and concluded with a colon;.

```
class example
{
data;
member functions;
……….
};
```

There are different access specifiers for defining the data and functions present inside a class. Access specifiers are used to identify access rights for the data and member functions of the class. There are three main types of access specifiers.

- private
- public
- protected

A private member within a class denotes that only members of the same class have accessibility. The private member is inaccessible from outside the class. Public members are accessible from outside the class. A protected access specifier is a stage between private and public access. If member functions defined in a class are protected, they cannot be accessed from outside the class but can be accessed from the derived class. When defining access Specifiers, the programmer must use the keywords: private, public or protected when needed, followed by a semicolon and then define the data and member functions under it.

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

General structure for defining a class is:

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

• *Unions*

A union is a user-defined data or class type that, at any given time, contains only one object from its list of members.

union [tag] { member-list } [declarators]; [union] tag declarators;

```
union DATATYPE{
 char   ch;
int i;
long   l;
float  f;
 double d; } var1;
```

Parameterstag : The type name given to the union.
member-list : List of the types of data the union can contain. See Remarks.
Declarators: Declarator list specifying the names of the union.

Declaring a Union

Begin the declaration of a union with the union keyword, and enclose the member list in curly braces:

A C++ union is a limited form of the class type. It can contain access specifiers (public, protected, private), member data, and member functions, including constructors and destructors. It cannot contain virtual functions or static data members. It cannot be used as a base class, nor can it have base classes. Default access of members in a union is public.

DATATYPE var3; // C++ declaration of a union variable

```
// using_a_union.cpp
#include <stdio.h>
union NumericType{
intiValue;
 long      lValue;
 double     dValue; };

int main() {
 union NumericType Values = { 10 };  //
iValue = 10
printf_s("%d\n", Values.iValue);
Values.dValue = 3.1416;
printf_s("%f\n", Values.dValue); }
```

A variable of a union type can hold one value of any type declared in the union. Use the member-selection operator (.) to access a member of a union:

```
var1.i = 6;          // Use variable as integer

var2.d = 5.327;      // Use variable as double
```

You can declare and initialize a union in the same statement by assigning an expression enclosed in braces. The expression is evaluated and assigned to the first field of the union.

- ***Enumerations***

An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword **enum** is used.

```
enum season {
spring = 0,
 summer = 4,
 autumn = 8,
winter= 12
};
```

```
enum season { spring, summer, autumn, winter };
```

Here, the name of the enumeration is season. And, spring, summer and winter are values of type season. By default, spring is 0, summer is 1 and so on. You can change the default value of an enum element during declaration (if necessary).

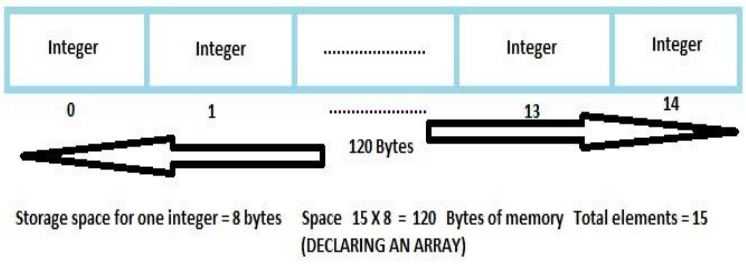
When you create an enumerated type, only blueprint for the variable is created. Here's how you can create variables of enum type.

```
enum boolean { false, true };
// inside function
enum boolean check;
```

➢ *Derived Data Types*

- ***Arrays***

An Array is a collection of fixed-size finite number of objects, data value that belong to the same type. An array is a collection of data storage locations. Each of the locations holds same type of data. The Storage Location is called an element or component of the array. The individual element within the array is called subscript. The subscript is the number of elements in the array surrounded by square brackets [ ].Array elements are counted from 0.In the above figure 15 elements are numbered from [0] to [14]. An array may be one-dimensional, two-dimensional or multi-dimensional depending up the specification of elements.



- ***Pointer***

A pointer is a variable which holds a memory address within. Each variable is located at a particular position in the

memory which is known as Address. The address can be stored in a pointer. A program accesses the value in the address stored in the pointer by using indirection operator. **In a program, pointers are used for** Managing data on free storage locations of the memory Passing variables by reference to functions and Accessing class member data and functions

General Syntax of using pointers is **:-** ***Data_Type *VriableName;***

- ***Function***

A function is a subprogram that acts on data and returns a value. A function can be invoked from the other parts of the program. Every C++ program contains one function, **i.e., main().** When the program starts, main() automatically becomes active and can call otherfunctions. A function can perform a specific task for the programmer. It is better to use multiple functions in the lengthy task.Functions areof two types arebuilt-in functions and user-defined functions.

➢ *C++ Comments*

Each and every language will provide this great feature which is used to document source code. We can create more readable and eye catching program structure using comments. We should use as many as comments in C++ program. Comment is non executable Statement in the C++**.**We can have two types of comment in Programming –Single Line Comment and Multiple Line Comment.

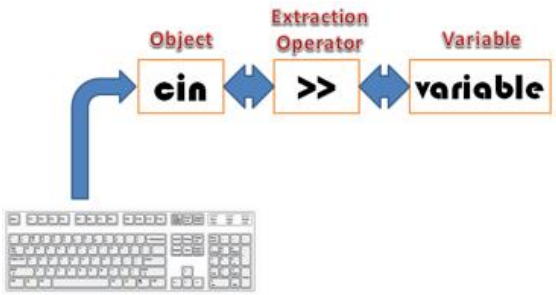
Single Line Comments in C++

- Single Line comment starts with "//" symbol.
- Remaining line after "//" symbol is ignored by browser.
- End of Line is considered as End of the comment.

Multiple Line Comments in C++

- Multi Line comment starts with "/*" symbol.
- Multi Line comment ends with "*/" symbol.

- ***Cin***

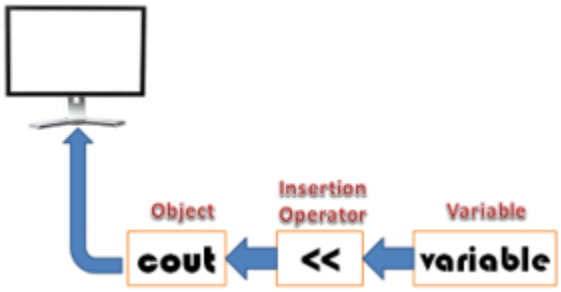


Cin is used for Extracting Input from User Using Keyboard. In C++ Extraction operator is used to accept value from the user. User can able to accept value from user and that value gets stored inside value container i.e. Variable. Syntax of cin is cin>> variable; we have to include iostream header file to use cin. #include<iostream.h>is for traditional C++ and #include<iostream>using namespace std; for ANSI Standard. cin is the **object of istream class**. Data flow direction is from input device to variable.

```
#include<iostream.h>
usingnamespacestd;
int main()
{
int number1;
int number2;
cout<<"Enter First Number: ";
cin>>number1;            //accept first
number
cout<<"Enter Second Number: ";
cin>>number2;            //accept first
number
cout<<"Addition : ";
cout<<number1+number2;      //Display
Addition
return 0; }
```

In above example, number1is declared as integer variable. When control comes over cin, Program waits for the input from user. We have written integer variable after extraction operator so it will wait and expect integer as input from user .If you request an integer you will get an integer, character for character, string for string etc. Multiple variables can be cascaded using single cin. For examplecin>> a >> b; It can be used to get the string that is cin>> string; When blank space is detected**,** extraction operator stops reading input.cin allow us to enter only one word**.**

- ***Cout***

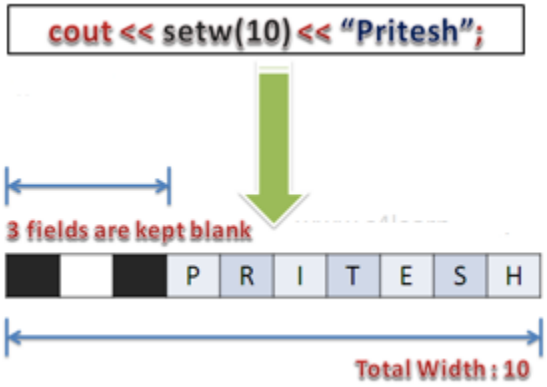



```
#include<iostream.h>
usingnamespace std;

int main(){
int number1;
int number2;
cout<<"Enter First Number: ";
cin>>number1;
cout<<"Enter Second Number: ";
cin>>number1;
cin>>number2;
cout<<"Addition : ";
cout<<number1+number2;
return 0; }
```

Cout is used to display Output to User Using Screen (Monitor). Insertion operator is used to display value to the user. The value may be some message in the form of string or variable. Syntax : cout<< variable; we have to include iostream header file to use cin. ***#include<iostream.h>*** is for traditional C++ and ***#include<iostream>*** using namespace std; for ANSI Standard. cout is used for displaying data on the screen. cout is the object of ostream class. Data flow direction is from variable to output device. Multiple outputs can be displayed using cout.

In above example, number1is declared as integer variable. When control comes over cout, Program will display the output to screen**.** Likewise we can display integer, character or string. Multiple variables can be cascaded using single cout. For example cout<< a << b; It can be used to display string that is cout<< string; .When blank space is detected, extraction operator stops reading input. Expression can be evaluated inside cout statement, that is cout<< "Addition is: " << num1+num2;

```
cout<< "Hello,\n";
cout<< "My name is \n Student...";

Or
cout << "Hello," <<endl;
cout << "My name is" <<endl;
cout<< " Student..." <<endl;
```



endl (Line Feed Operator) We can use ‘n**‘or ‘**endl**‘to print the output on new line.**

C++ setw() : Setting Field Width

setw() is library function in C++. setw() is declared inside #include<iomanip> **.** Itwill set field widthandsets the number of characters to be used as the *field width* for the next insertion operation.

Syntax : setw(num). Here num is width to be set in order to insert next value**.**

```
#include <iostream>
#include <iomanip>
usingnamespace std;

int main ()
{
 cout << setw (10);
 cout << "Pritesh" << endl;
return 0;
}
```

Now Length of String Pritesh is 7**.** We have set field width 10 so it will utilize 7 fields. Remaining 3 fields are kept blank**.** Default Justification is Left

➢ *Variables*

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

. The rules of naming identifiers in C++:

1. C++ is case-sensitive so that Uppercase Letters and Lower Case letters are different
2. The name of identifier cannot begin with a digit**.** However, Underscore can be used as first character while declaring the identifier.
3. Only alphabetic characters, digits and underscore **(_)** are permitted in C++ language for declaring identifier.
4. Other special characters are not allowed for naming a variable / identifier
5. Keywords cannot be used as Identifier.

Declaration and Dynamic initialization of Variables

In C we have to declare all the variables at the starting of any function or scope. However in C++ we can declare variable anywhere in the scope before its first use. There is no need to declare all the variables at the starting of the scope.
Example:
The process of initializing variable at the time of its declaration at run time is known as dynamic initialization of variable. Thus in dynamic initialization of variable a variable is assigned value at run time at the time of its declaration.

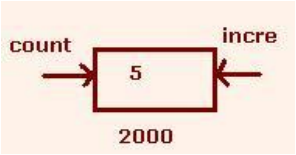
```
int main()
{
int n,sum=0;
for (int i = 1 ; i<=10;i++)
{
cin>>n;
sum=sum+n;
}
float average = sum/10;
}
```

```
int main()
{
int a;
cout<< “Enter Value of a”;
cin>> a;
int cube = a * a * a;
}
```

In above example variable cube is initialized at run time using expression a * a * a at the time of its declaration.

Reference variable

A reference variable allows us to create an alternative name for already defined variable. Once you define a reference variable that references already defined variable then you can use any of them alternatively in the program. Both refer to the same memory location. Thus if you change value of any of the variable it will affect both variables because one variable reference to another variable. The general syntax of creating reference variable is given below:
Data-type &Referace_Name = Variable_Name;
The reference variable must initialize at the time of declaration.



Example:

```
int count;
count = 5;
int &incre = count;
```

Here,we have already defined variable named count. Then we create a reference variable named incre that refer to the variable count.

Since both variable refer to same memory location if you change value of variable count using following statement: Count=count + 5;Will change the value of variable count and incre to 10. The major use of reference variable is in function. If we want to passreference of the variable at the time of function call so that the function works with original variable we can use reference variable.

Example :-
If Product is declared as a reference to the variable Multiplication, then both Product and Multiplication represent th e same variable. Both of these can be used interchangeably.

```
int Product;
int& Product = Multiplication;
Multiplication = 256;
cout<< "Product=" << Product;
cout<< "\nMultiplication=" << Multiplication;
```

➢ *C++ Keyword*

In C++, keywords are reserved identifiers which cannot be used as names for the variables in a program.
List :

asm	**else**	**operator**	**default**	**continue**	**if**	**short**	**template**
auto	**enum**	**private**	**delete**	**inline**	**struct**	**goto**	**this**
break	**extern**	**protected**	**double**	**sizeof**	**signed**	**class**	**throw**
case	**new**	**public**	**virtual**	**switch**	**unsigned**	**union**	**try**
catch	**friend**	**register**	**long**	**void**	**const**	**static**	**typedef**

➢ *C++ Operators*

An operator is a symbol which tells compiler to take an action on operands and yield a value. The Value on which operator operates is called as operands. C++ supports wide verity of operators. Supported operators are listed below –

Operator	Explanation
Arithmetic Operators	Used for arithmetic operations
Relational Operators	Used for specifying relation between two operands
Logical Operators	Used for identifying the truth value of the expression
Bitwise Operators	Used for shifting the bits
Assignment Operators	Used for assigning the value to the variable

Consider that we have A = 20 and B = 10

- ***Arithmetic Operators:***

Operator	Description	Example
+	Adds two operands or variables	A + B = 30
-	Subtracts second operand from the first	A – B = 10
*	Multiplies both operands	A * B = 200
/	Divides numerator by denominator	A / B = 2
%	After dividing the numberator by denominator remainder will be returned after division	A % B = 0
++	Increment operator will increases integer value by one	A++ = 21
#8211; -	Decrement operator will decreases integer value by one	A- #8211; = 19

- ***Relational Operators:*** Consider that A = 40 and B = 20

Symbol	Meaning	Example
>	Greater than	A > B returns true
<	Less than	A < B returns false
>=	Greater than equal to	A >= B returns false
<=	Less than equal to	A <= B returns false
==	Equal to	A == B returns false
!=	Not equal to	A != B returns true

- ***Logical Operators:***

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

- ***Bitwise Operators:***

Operator	**Description**	**Example**
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = -61, i.e,. 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

- ***Assignment Operators***

Operator	**Description**	**Example**
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C= C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

➢ *Expressions*

An expression is a sequence of operators (Zero or more) and their operands (one or more), that specifies a computation. Expression evaluation may produce a result. It may also include function call and return values. Expression may be of following types. Expressions may combination of below expressions and are called as Compound expressions.

1. Constant
2. Integral

3. Float
4. Pointer
5. Relational
6. Logical
7. Bitwise.

Constant expression

A *constant expression* is an expression with a value that is determined during compilation. That value can be evaluat ed at runtime, but cannot be changed. Constant expressions can be composed of integer, character, floating-point, an d enumeration constants, as well as other constant expressions. Some constant expressions, such as string literals or address constants, are lvalues.

Expression Containing Constant	Constant
x = 42;	42
extern int cost = 1000;	1000
y = 3 * 29;	3 * 29

Integral, Float and Pointer expressions

Integral expressions are produces integer results after implicit and explicit type conversions.Example: b, b*g-7 and 10+ int(2.3) where b and g are integer variables.

Float expressions are produces float results after implicit and explicit type conversions.Example: b, b*g-7 and 10+ float(2) where b and g are float variables.

Pointer expressions produce address values. Example: &b, ptr, and ptr+1.

Relational expression

A condition or logical expression is an expression that can only take the values true or false. A simple form of logical expression is the relational expression. The following is an example of a relational expression:

x < y

which takes the value true if the value of the variable x is less than the value of the variable y.The general form of a relational expression is:

operand1 relational-operator operand2

Logical expression

It is possible to specify more complex conditions than those which can be written using only the relational operators described above. Since the value of a condition has a numerical interpretation it could be operated on by the usual arithmetic operators, this is not to be recommended. There are explicit logical operators for combining the logical values true and false.The simplest logical operator is not which is represented in C++ by !. It operates on a single operand, and returns false if its operand is true and true if its operand is false.The operator and, represented by &&, takes two operands and is true only if both of the operands are true. If either operand is false, the resulting value is false. or is the final logical operator and is represented by ||. It results in true if either of its operands is true. It returns false only if *both* its operands are false. Example: a>b &&y==35.

Bitwise logical expression

A bitwise logical expression (also called a masking expression) is an expression in which a logical operator operates on individual bits within integer, real, Cray pointer, or Boolean operands, giving a result of type Boolean. Each operand is treated as a single storage unit. This storage unit is a 32-bit word. The result is a single storage unit. Boolean values and bitwise logical expressions are contrasted to logical values and expressions. Example: b<<3.

➢ *Operator Overloading*

The process of giving special meaning to the existing c++ operator is known as "Operator Overloading". Using the concept of operator overloading we can use operators with object of the class. For example: We can use + to perform addition of two integer or floating point numbers. In the same way we can overload + operator so that we can perform addition of two objects of the class.
When you overload the existing operator the basic syntax rules of the operator is not changed. It remains as it is. In order to overload particular operator you need to use special function known as operator function.

Scope resolution operator (::)

The :: (scope resolution) operator is used to qualify hidden names so that you can still use them. You can use the unary scope operator if a namespace scope or global scope name is hidden by an explicit declaration of the same name in a block or class. For example:

```
int count = 0;
int main(void)
{
 int count = 0;
 ::count = 1;  // set global count to 1
 count = 2;    // set local count to 2
 return 0;
}
```

The declaration of count declared in the main function hides the integer named count declared in global namespace scope. The statement ::count = 1 accesses the variable named count declared in global namespace scope.

C++ Memory Management Operators

The concept of arrays has a block of memory reserved. The disadvantage with the concept of arrays is that the programmer must know, while programming, the size of memory to be allocated in addition to the array size remaining constant. In programming there may be scenarios where programmers may not know the memory needed until run time. In this case, the programmer can opt to reserve as much memory as possible, assigning the maximum memory space needed to tackle this situation. This would result in wastage of unused memory spaces. Memory management operators are used to handle this situation in C++ programming language. There are two types of memory management operators in C++:

- new
- delete

new operator:

The general syntax of new operator in C++ is as follows: pointer variable = new datatype; In the above statement, new is a keyword and the pointer variable is a variable of type datatype. For example:int *a=new int; In this example, the new operator allocates sufficient memory to hold the object of datatype int and returns a pointer to its starting point. The pointer variable a holds the address of memory space allocated. Dynamic variables are never initialized by the compiler. The assignment can be made in either of the two ways:

```
int*a =newint;
*a =20;
or
int*a =newint(20);
```

delete operator:

The delete operator in C++ is used for releasing memory space when the object is no longer needed. Once a new operator is used, it is efficient to use the corresponding delete operator for release of memory. The general syntax of delete operator in C++ is as follows: delete pointer variable; In this example, delete is a keyword and the pointer variable is the pointer that points to the objects already created in the new operator.
Example:

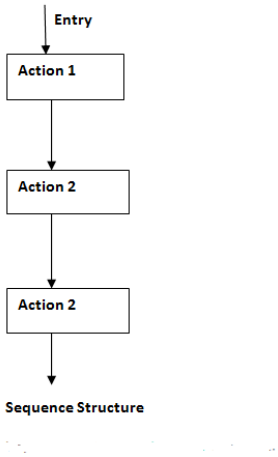
```
tart here | Untitled1.cpp
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      //Allocates using new operator memory space in memory for storing a i
6      int *a= new int;
7      *a=100;
8      cout << " The Output is:a= " << *a;
9      //Memory Released using delete operator
10     delete a;
11     return 0;
12 }
13
```

```
C:\Users\Admin\Desktop\Untitled1.exe
 The Output is:a= 100
Process returned 0 (0x0)   execution time : 0.316 s
Press any key to continue.
```

In the above program, the statement: int *a= new int;Holds memory space in memory for storing a integer datatype. The statement:*a=100 this denotes that the value present in address location pointed by the pointer variable a is 100 and this value of a is printed in the output statement giving the output shown in the example above. The memory allocated by the new operator for storing the integer variable pointed by a is released using the delete operator as: delete a;

➢ *Control structures*

Control structures are used to alter the flow of execution of the program. Why do we need to alter the program flow? The reason is "**decision making**"! In life, we may be given with a set of option like doing "Electronics" or "Computer science". We do make a decision by analyzing certain conditions (like our personal interest, scope of job opportunities etc). With the decision we make, we alter the flow of our life's direction. This is exactly what happens in a C/C++ program. We use control structures to make decisions and alter the direction of program flow in one or the other path(s) available.



There are three types of control structures available in C and C++.

1) Sequence structure (straight line paths).
2) Selection structure (one or many branches).
3) Loop structure (repetition of a set of activities).

All the 3 control structures and its flow of execution are represented in the flow charts given below.

Control statements in C++ to implement control structures

"***Control structures are the basic entities of a structured programming language***". To implements these "control structures" in a C++ program, the language provides 'control statements'. So to implement a particular control structure in a programming language, we need to learn how to use the relevant control statements in that particular language.

The control statements are:-

1. Switch
2. If
3. If Else
4. While

5. Do While
6. For

As shown in the flow charts:-

- Selection structures are implemented using **if**, **if else** and **switch** statements.
- Looping structures are implemented using **while**, **do While** and **for** statements.

Selection structures

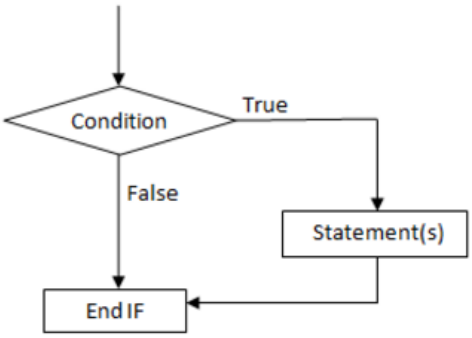
Selection structures are used to perform 'decision making' and then branch the program flow based on the outcome of decision making. If and if else statements are 2 way branching statements where as Switch is a multi branching statement.

The simple If statement

The syntax format of a simple if statement is as shown below.

```
if (expression)
{
statement 1;
statement 2;
}
```

statement 1;// Program control is transferred directly to this line, if the expression is FALSE
statement 2;



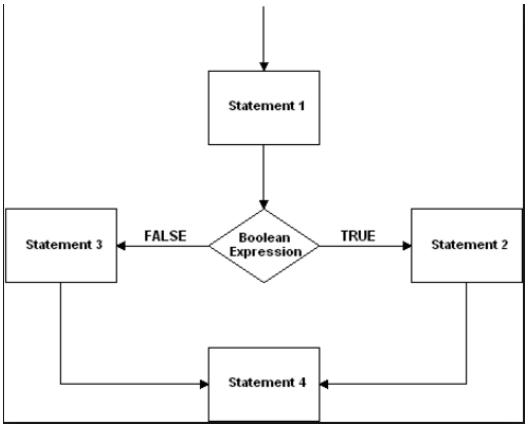
The expression given inside the brackets after if is evaluated first. If the expression is true, then statements inside the curly braces that follow if(expression) will be executed. If the expression is false, the statements inside curly braces will not be executed and program control goes directly to statements after curly braces.

Flow Chart:

The if else statement

Syntax format for if else statement is shown below.

```
if (expression)
{
statement 1;
statement 2;
}else
{
statement 1;
statement 2;
}
other statements;
```

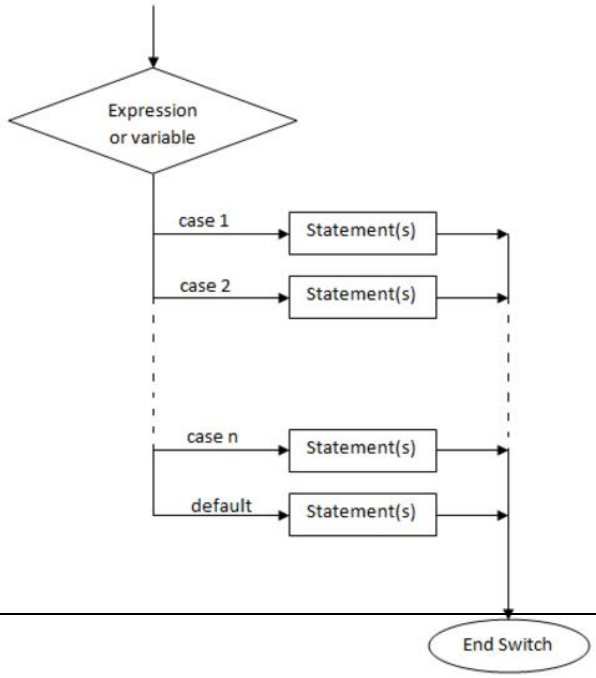


The execution begins by evaluation expression. If it is **TRUE**, then statements inside the immediate curly braces are evaluated. If it is **FALSE**, program control is transferred directly to immediate else statement (without any expression) is executed along with the statements 1 and 2 inside the curly braces of last **else** statement.

Switch statement

Switch is a multi branching control statement. **Syntax for switch statement is shown below.**

```
switch (expression
```



```
{
case value1:
statement 1;
statement 2;
break;
case value2:
statement 1;
statement 2;
break;
default:
statement 1;
statement 2;
break;
}
```

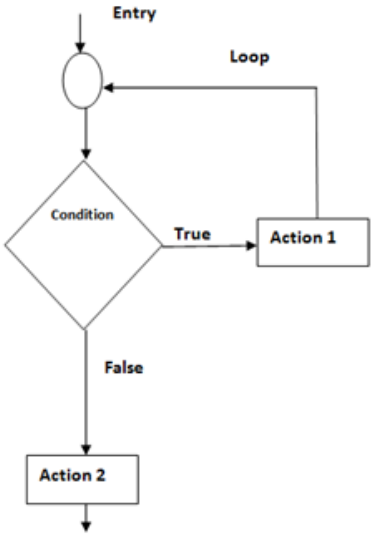
Execution of switch statement begins by evaluating the expression inside the switch keyword brackets. The expression should be an integer (1, 2, 100, 57 etc) or a character constant like 'a', 'b' etc. This expression's value is then matched with each case values. There can be any number of case values inside a switch statements block. If first case value is not matched with the expression value, program control moves to next case value and so on. When a case value matches with expression value, the statements that belong to a particular case value are executed.

Notice that last set of lines that begins with default. The word default is a keyword in C/C++. When used inside switch block, it is intended to execute a set of statements, if no case values matches with expression value. So if no case values are matched with expression value, the set of statements that follow default: will get executed.

Note: Notice the break statement used at the end of each case values set of statements.
Note: - Switch statement is used for multiple branching. The same can be implemented using **nested "If Else"** statements. But use of nested if else statements make program writing tedious and complex. Switch makes it much easier. Compare this program with above one.

Loop structures

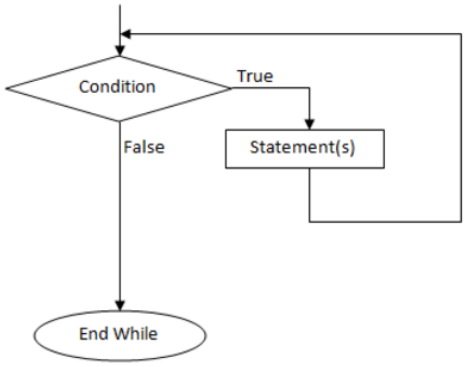
A loop structure is used to execute a certain set of actions for a predefined number of times or until a particular condition is satisfied. There are 3 control statements available in C/C++ to implement loop structures. **While, Do while and for statements.**



The while statement

Syntax for while loop is shown below:

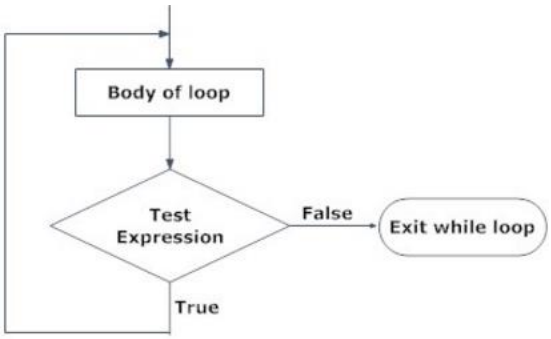
```
while (condition){
        statement 1;
        statement 2;
        statement 3;
}
```



The condition is checked for TRUE first. If it is TRUE then all statements inside curly braces are executed.Then program control comes back to check the condition has changed or to check if it is still TRUE. The statements inside braces are executed repeatedly, as long as the condition is TRUE. When the condition turns FALSE, program control exits from while loop.
Flow Chart:

The do while statement
Syntax for do while loop is shown below:

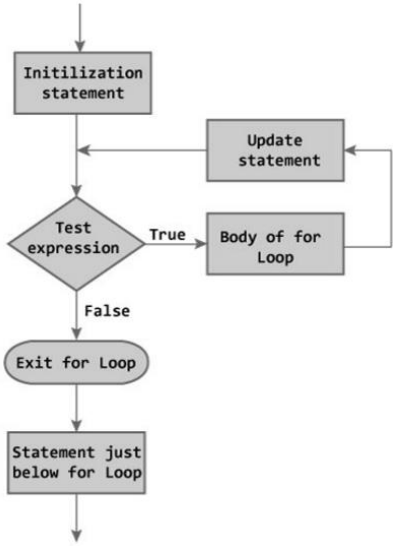
```
do
{
        statement 1;
        statement 2;
        statement 3;
}
while(condition);
```



Unlike while, do while is an exit controlled loop. Here the set of statements inside braces are executed first. The condition inside while is checked only after finishing the first time execution of statements inside braces. If the condition is TRUE, then statements are executed again. This process continues as long as condition is TRUE. Program control exits the loop once the condition turns FALSE.
Flow Chart:

The for statement
Syntax of for statement is shown below:

```
for (initialization statements; test condition; iteration statements)
{
        statement1;
        statement2;
        statement3;
}
```



The for statement is an entry controlled loop. The difference between while and for is in the number of repetitions. The for loop is used when an action is to be executed for a predefined number of times. The while loop is used when the number of repetitions is not predefined. The program control enters the for loop. At first it executes the statements given as initialization statements. Then the condition statement is evaluated. If conditions are TRUE, then the block of statements inside curly braces is executed. After executing curly brace statements fully, the control moves to the "iteration" statements. After executing iteration statements, control comes back to condition statements.

Flow Chart:

- *Content beyond Syllabus:*

Storage Classes in C++ Programming

Storage class of a variable defines the lifetime and visibility of a variable. Lifetime means the duration till which the variable remains active and visibility defines in which module of the program the variable is accessible. There are four types of storage classes in C++. They are:

- Automatic
- External
- Static
- Register

1. Automatic Storage Class

Automatic storage class assigns a variable to its default storage type. auto keyword is used to declare automatic variables. However, if a variable is declared without any keyword inside a function, it is automatic by default. This variable is visible only within the function it is declared and its lifetime is same as the lifetime of the function as well. Once the execution of function is finished, the variable is destroyed.Syntax of Automatic Storage Class Declaration is datatype var_name1 [= value];orauto datatype var_name1 [= value];Example of Automatic Storage Class

```
auto int x;
float y = 5.67;
```

2. External Storage Class

External storage class assigns variable a reference to a global variable declared outside the given program. extern keyword is used to declare external variables. They are visible throughout the program and its lifetime is same as the lifetime of the program where it is declared. This visible to all the functions present in the program.Syntax of External Storage Class Declaration is extern datatype var_name1;

For example,

```
extern float var1;
```

Example

File: sub.cpp

```
int test=100;

void multiply(int n)
{
   test=test*n;
}
```

File: main.cpp

```
#include<iostream>
#include "sub.cpp"  // includes the content of sub.cpp
using namespace std;

extern int test;  // declaring test

int main()
{
cout<<test<<endl;
   multiply(5);
cout<<test<<endl;
   return 0;
}
```

A variable test is declared as external in main.cpp. It is a global variable and it is assigned to 100 in sub.cpp. It can be accessed in both files. The function multiply() multiplies the value of test with the parameter passed to it while invoking it. The program performs the multiplication and changes the global variable test to 500.

3. Static Storage Class

Static storage class ensures a variable has the visibility mode of a local variable but lifetime of an external variable. It can be used only within the function where it is declared but destroyed only after the program execution has finished. When a function is called, the variable defined as static inside the function retains its previous value and operates on it. This is mostly used to save values in a recursive function.Syntax of Static Storage Class Declaration is static datatype var_name1 [= value];
For example,
static int x = 101;
static float sum;

4. Register Storage Class
Register storage assigns a variable's storage in the CPU registers rather than primary memory. It has its lifetime and visibility same as automatic variable. The purpose of creating register variable is to increase access speed and makes program run faster. If there is no space available in register, these variables are stored in main memory and act similar to variables of automatic storage class. So only those variables which requires fast access should be made register.Syntax of Register Storage Class Declaration is register datatype var_name1 [= value];
For example,
register int id;
register char a;

MODULE II

Functions, Classes and Objects

- A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main (). You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.
- A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.
- The C++ standard library provides numerous built-in functions that your program can call. For example, function strcat() to concatenate two strings, function memcpy() to copy one memory location to another location and many more functions.
- A function is known with various names like a method or a sub-routine or a procedure etc.

Defining a Function
The general form of a C++ function definition is as follows −

```
return_type function_name (parameter list)
{
  //Body of the function
}
```

A function definition consists of a function header and a function body. Here are all the parts of a function −

1. Return Type : A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.
2. Function Name : This is the actual name of the function. The function name and the parameter list together constitute the function signature.
3. Parameters : A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
4. Function Body : The function body contains a collection of statements that define what the function does.

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two −

```
int max(int num1, int num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

Function Declarations

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. A function declaration has the following parts .

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration.

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration.

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function. When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program. To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function. The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function −

S.No	Call Type & Description
1	Call by Value: This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Call by Pointer: This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3	Call by Reference: This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Function call by value

The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

```
void swap(int x, int y)
{
  int temp;
  temp = x; /* save the value of x */
  x = y;    /* put y into x */
  y = temp; /* put x into y */
  return;
}
```

Now, let us call the function swap() by passing actual values as in the following example.

```
#include <iostream>
using namespace std;
// function declaration
void swap(int x, int y);
int main ()
{
  // local variable declaration:
  int a = 100;
  int b = 200;
  cout << "Before swap, value of a :" << a << endl;
  cout << "Before swap, value of b :" << b << endl;
  // calling a function to swap the values.
  swap(a, b);
  cout << "After swap, value of a :" << a << endl;
  cout << "After swap, value of b :" << b << endl;
  return 0;
}
```

Function call by Pointer

The call by pointer method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument. To pass the value by pointer, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

```
void swap(int *x, int *y)
{
  int temp;
  temp = *x; /* save the value at address x */
  *x = *y; /* put y into x */
  *y = temp; /* put x into y */
  return;
}
```

For now, let us call the function swap() by passing values by pointer as in the following example.

```
#include <iostream>
using namespace std;
// function declaration
void swap(int *x, int *y);
```

```
int main ()
{
  // local variable declaration:
  int a = 100;
  int b = 200;
  cout << "Before swap, value of a :" << a << endl;
  cout << "Before swap, value of b :" << b << endl;
  /* calling a function to swap the values.
     * &a indicates pointer to a ie. address of variable a and
     * &b indicates pointer to b ie. address of variable b.
  */
  swap(&a, &b);
  cout << "After swap, value of a :" << a << endl;
  cout << "After swap, value of b :" << b << endl;
  return 0;
}
```

Function call by reference

The call by reference method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument. To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

```
void swap(int &x, int &y)
{
  int temp;
  temp = x; /* save the value at address x */
  x = y;    /* put y into x */
  y = temp; /* put x into y */
  return;
}
#include <iostream>
using namespace std;
void swap(int &x, int &y);
int main () {
  // local variable declaration:
  int a = 100;
  int b = 200;
  cout << "Before swap, value of a :" << a << endl;
  cout << "Before swap, value of b :" << b << endl;
  /* calling a function to swap the values using variable reference.*/
  swap(a, b);
  cout << "After swap, value of a :" << a << endl;
  cout << "After swap, value of b :" << b << endl;
  return 0;
}
```

Inline Functions

Inline function is one of the important features of C++. When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of

the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
   // function code
}
```

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

1) If a function contains a loop. (for, while, do-while).
2) If a function contains static variables.
3) If a function is recursive.
4) If a function return type is other than void, and the return statement doesn't exist in function body.
5) If a function contains switch or goto statement.

The following program demonstrates the use of use of inline function.

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
   return s*s*s;
}
int main()
{
   cout << "The cube of 3 is: " << cube(3) << "\n";
   return 0;
}
```

Overloading (Operator and Function)

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 is 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 have 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.

Function Overloading

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters. Function overloading can be considered as an example of polymorphism feature in C++. Following is a simple C++ example to demonstrate function overloading.

```
#include <iostream>
using namespace std;
void print(int i) {
```

```
  cout << " Here is int " << i << endl;
}
void print(double  f)
{
  cout << " Here is float " << f << endl;
}
void print(char* c)
 {
   cout << " Here is char* " << c << endl;
}

int main()
{
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Steps to select functions in function overloading concept by the compiler.

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

```
char to int
float to double
```

to find a match.

3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

```
long square(long n)
double square(double x)
```

A function call such as

```
square(10)
```

will cause an error because **int** argument can be converted to either **long** or **double**, thereby creating an ambiguous situation as to which version of **square()** should be used.

4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

Specifying a class

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new abstract data type that can be treated like any other build-in data type.

Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type scope of its members. The class function definitions describe how the class functions are implemented. The general form of a class declaration is:

```
class class_name
{
private:
variable declaration;
function declaration;
public:
variable declaration;
```

```
function declaration;
};
```

A class is a blueprint for the object. We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.

How to define a class in C++?

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

```
class className
  {
  // some data
  // some functions
  };
```

Example: Class in C++

```
class Test
{
   private:
      int data1;
      float data2;

   public:
      void function1()
      {   data1 = 2;  }

      float function2()
      {
         data2 = 3.5;
         return data2;
      }
  };
```

Here, we defined a class named Test.

This class has two data members: data1 and data2 and two member functions: function1() and function2().

Keywords: private and public

You may have noticed two keywords: private and public in the above example.

The private keyword makes data and functions private. Private data and functions can be accessed only from inside the same class.

The public keyword makes data and functions public. Public data and functions can be accessed out of the class.

Here, data1 and data2 are private members where as function1() and function2() are public members.

If you try to access private data from outside of the class, compiler throws error. This feature in OOP is known as data hiding.

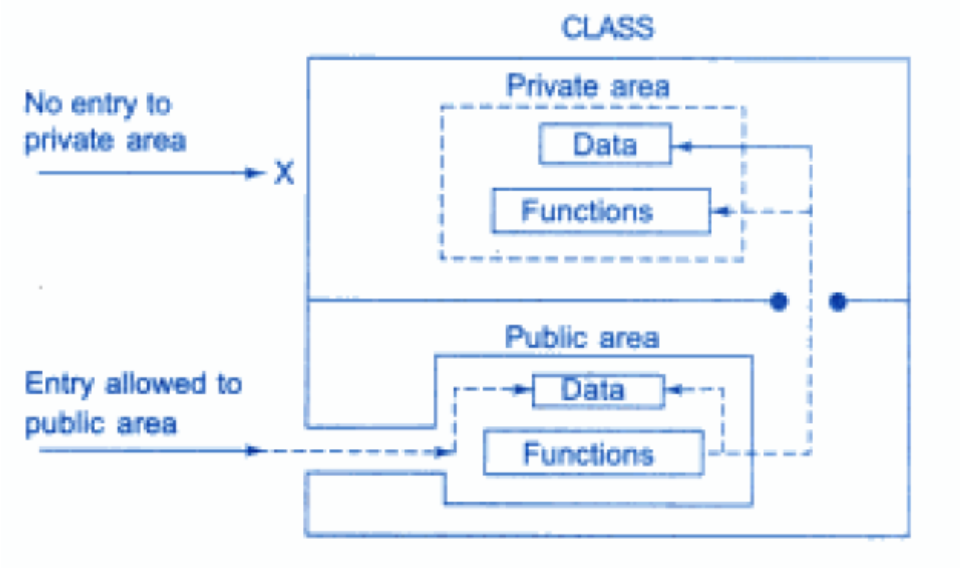


Figure: Data hiding in classes

Types of Member functions:
Access functions,
Utility Functions,
Constructors,
Destructors,
Overloaded Operators

C++ Objects
When class is defined, only the specification for the object is defined; no memory or storage is allocated.
To use the data and access functions defined in the class, you need to create objects.
Syntax to Define Object in C++

```
className objectVariableName;
```

You can create objects of Test class (defined in above example) as follows:

```
class Test
{
   private:
      int data1;
      float data2;

   public:
      void function1()
      {   data1 = 2;  }

      float function2()
      {
         data2 = 3.5;
         return data2;
      }
};
```

```
int main()
{
   Test o1, o2;
}
```

Here, two objects o1 and o2 of Test class are created.
In the above class Test, data1 and data2 are data members and function1() and function2() are member functions.

How to access data member and member function in C++?

You can access the data members and member functions by using a . (dot) operator. For example,

```
o2.function1();
```

This will call the function1() function inside the Test class for objects o2.
Similarly, the data member can be accessed as:

```
o1.data2 = 5.5;
```

It is important to note that, the private members can be accessed only from inside the class.
So, you can use o2.function1(); from any function or class in the above example. However, the code o1.data2 = 5.5; should always be inside the class Test.

Example: Object and Class in C++ Programming

// Program to illustrate the working of objects and class in C++ Programming

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

class Test
{
   private:
      int data1;
      float data2;

   public:

      void insertIntegerData(int d)
      {
        data1 = d;
        cout << "Number: " << data1;
       }

      float insertFloatData()
      {
         cout << "\nEnter data: ";
         cin >> data2;
         return data2;
       }
};

int main()
{
    Test o1, o2;
    float secondDataOfObject2;

    o1.insertIntegerData(12);
    secondDataOfObject2 = o2.insertFloatData();

    cout << "You entered " << secondDataOfObject2;
    return 0;
}
```

Output

```
Number: 12
Enter data: 23.3
```

You entered 23.3
In this program, two data members data1 and data2 and two member functions insertIntegerData() and insertFloatData() are defined under Test class.
Two objects o1 and o2 of the same class are declared.
The insertIntegerData() function is called for the o1 object using:
o1.insertIntegerData(12);
This sets the value of data1 for object o1 to 12.
Then, the insertFloatData() function for object o2 is called and the return value from the function is stored in variable secondDataOfObject2 using:
secondDataOfObject2 = o2.insertFloatData();
In this program, data2 of o1 and data1 of o2 are not used and contains garbage value.

Arrays within Class:

Arrays can be declared as the members of a class. The arrays can be declared as private, public or protected members of the class. To understand the concept of arrays as members of a class, consider this example.

```
#include<iostream>
using namespace std;
const int size=5;
class student
{
int roll_no;
int marks[size];
public:
void getdata ();
void tot_marks ();
} ;
void student :: getdata ()
{
cout<<"\nEnter roll no: ";
Cin>>roll_no;
for(int i=0; i<size; i++)
{
cout<<"Enter marks in subject"<<(i+1)<<": ";
cin>>marks[i] ;
}
void student :: tot_marks() //calculating total marks
{
int total=0;
for(int i=0; i<size; i++)
total+ = marks[i];
cout<<"\n\nTotal marks "<<total;
}
int main()
student stu;
stu.getdata() ;
stu.tot_marks() ;
return 0;
}
```

The output of the program is

```
Enter roll no: 101
Enter marks in subject 1: 67
Enter marks in subject 2 : 54
```

```
Enter marks in subject 3 : 68
Enter marks in subject 4 : 72
Enter marks in subject 5 : 82
Total marks = 343
```

In this example, an array marks is declared as a private member of the class student for storing a student's marks in five subjects. The member function tot_marks () calculates the total marks of all the subjects and displays the value. Similar to other data members of a class, the memory space for an array is allocated when an object of the class is declared. In addition, different objects of the class have their own copy of the array. Note that the elements of the array occupy contiguous memory locations along with other data members of the object. For instance, when an object stu of the class student is declared, the memory space is allocated for both rollno and marks

Memory allocation for objects:

Once a class is defined, it can be used to create variables of its type known as objects. The relation between an object and a class is the same as that of a variable and its data type. The syntax for declaring an object is

```
class_name = object_list;
```

where,
class_name = the name of the class
object_list = a comma-separated list of objects.

To understand the concept of instantiation, consider this example.

Example : Declaring objects of a class

```
class book
{
II body of the class
} ;
int main ()
{
book bookl, book2, book3; //objects of class book
return 0;
}
```

In this example, three objects namely, book1, book2 and book3 of the class book have been created in main ().
Note that the syntax for the declaration of objects of a particular class is same as that for the declaration of variables of a built-in data type. To understand this concept, consider these statements.

```
int a,b,c; // declaration of variables
book bookl, book2, book3; // declaration of objects
```

Like structures, objects of a class can also be declared at the time of defining the class as shown here.

```
class class_name
{
Object_list;
}
```

Hence, the objects of the class book can also be declared with the help of these statements.

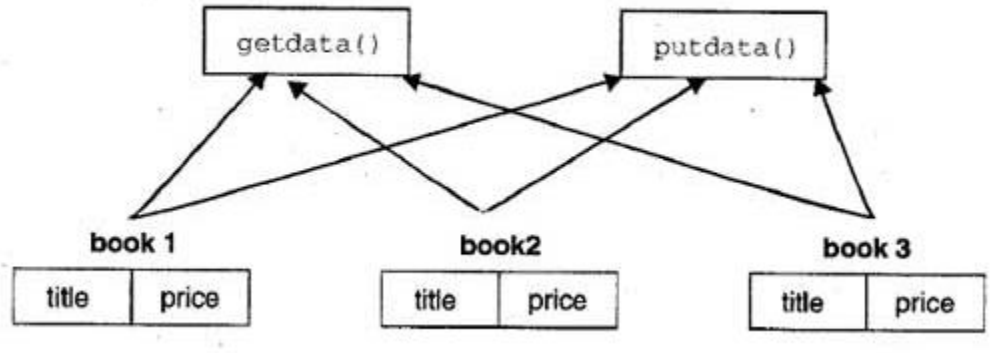
```
class book
{
// body of class
}bookl,book2,book3; // declaration of objects
```

Memory Allocation for Objects

1. Before using a member of a class, it is necessary to allocate the required memory space to that member. The way the memory space for data members and member functions is allocated is different regardless of the fact that both data members and member functions belong to the same class.
2. The memory space is allocated to the data members of a class only when an object of the class is declared, and not when the data members are declared inside the class. Since a single data member can have different

values for different objects at the same time, every object declared for the class has an individual copy of all the data members.

3. On the other hand, the memory space for the member functions is allocated only once when the class is defined. In other words, there is only a single copy of each member function, which is shared among all the objects.
4. For instance, the three objects, namely, book1, book2 and book3 of the class book have individual copies of the data members title and price. However, there is only one copy of the member functions getdata () and putdata () that is shared by all the three objects.



Arrays of objects

Like array of other user-defined data types, an array of type class can also be created. The array of type class contains the objects of the class as its individual elements. Thus, an array of a class type is also known as an array of objects. An array of objects is declared in the same way as an array of any built-in data type. The syntax for declaring an array of objects is

class_name array_name [size] ;

To understand the concept of an array of objects, consider this example.

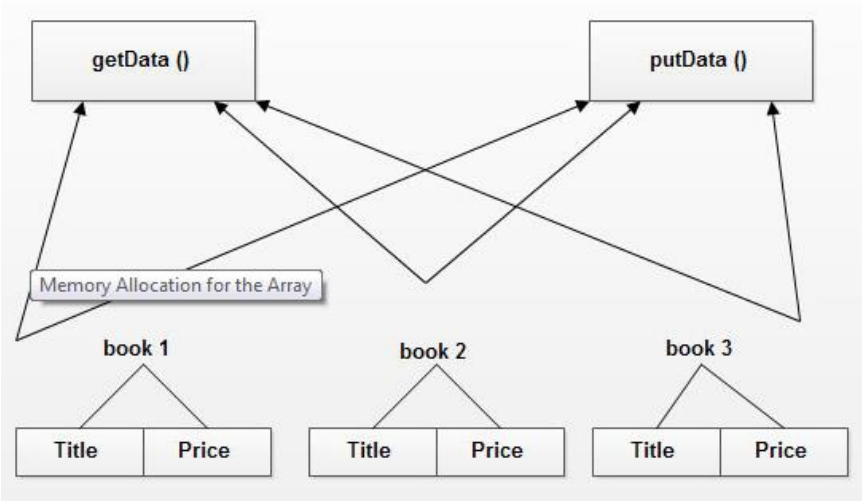
```
#include<iostream>
using namespace std;
class books
{
char tit1e [30];
float price ;
public:
void getdata ();
void putdata ();
} ;
void books :: getdata ()
{
cout<<"Title:";
Cin>>title;
cout<<"Price:";
cin>>price;

}
void books :: putdata ()
{
cout<<"Title:"<<title<< "\n";
cout<<"Price:"<<price<< "\n";
```

```
const int size=3 ;
int main ()
{
books book[size] ;
for(int i=0;i<size;i++)
{
cout<<"Enter details o£ book "<<(i+1)<<"\n";
book[i].getdata();
}
for(int i=0;i<size;i++)
{
cout<<"\nBook "<<(i+l)<<"\n";
book[i].putdata() ;
}
return 0;
}
```

The output of the program is

```
Enter details of book 1
Title: c++
Price: 325
Enter details of book 2
Title: DBMS
Price:. 455
Enter details of book 3
Title: Java
Price: 255
Book 1
Title: c++
Price: 325
Book 2
Title: DBMS
Price: 455
Book 3
Title: Java
Price: 255
```

In this example, an array book of the type class books and size three is declared. This implies that book is an array of three objects of the class books. Note that every object in the array book can access public members of the class in the same way as any other object, that is, by using the dot operator. For example, the statement book [i] . getdata () invokes the getdata () function for the ith element of array book. When an array of objects is declared, the memory is allocated in the same way as to multidimensional arrays. For example, for the array book, a separate copy of title and price is created for each member book[0], book[l] and book[2]. However, member functions are stored at a different place in memory and shared among all the array members. For instance, the memory space is allocated to the the array of objects book of the class books,



friend Function:

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function. The complier knows a given function is a friend function by the use of the keyword **friend**. For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

Declaration of friend function

```
class class_name
{
   ... .. ...
   friend return_type function_name(argument/s);
   ... .. ...
}
```

Now, you can define the friend function as a normal function to access the data of the class. No friend keyword is used in the definition.

```
class className
{
   ... .. ...
   friend return_type functionName(argument/s);
   ... .. ...
}

return_type functionName(argument/s)
{
   ... .. ...
   // Private and protected data of className can be accessed from
   // this function because it is a friend function of className.
   ... .. ...
}
```

```
/* C++ program to demonstrate the working of friend function.*/
#include <iostream>
using namespace std;

class Distance
{
   private:
      int meter;
```

```
    public:
        Distance(): meter(0) { }
        //friend function
        friend int addFive(Distance);
};

// friend function definition
int addFive(Distance d)
{
    //accessing private data from non-member function
    d.meter += 5;
    return d.meter;
}

int main()
{
    Distance D;
    cout<<"Distance: "<< addFive(D);
    return 0;
}
```

Here, friend function addFive() is declared inside Distance class. So, the private data metercan be accessed from this function. A more meaningful use would to when you need to operate on objects of two different classes. That's when the friend function can be very helpful. You can definitely operate on two objects of different classes without using the friend function but the program will be long, complex and hard to understand.

Following are some important points about friend functions and classes:

1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

2) Friendship is not mutual. If a class A is friend of B, then B doesn't become friend of A automatically.

3) Friendship is not inherited

Pointers to class members

Just like pointers to normal variables and functions, we can have pointers to class member functions and member variables.

Defining a pointer of class type

We can define pointer of class type, which can be used to point to class objects.

```
class Simple
{
 public:
 int a;
};

int main()
{
 Simple obj;
 Simple* ptr;   // Pointer of class type
 ptr = &obj;

 cout << obj.a;
 cout << ptr->a;  // Accessing member with pointer
}
```

Here you can see that we have declared a pointer of class type which points to class's object. We can access data members and member functions using pointer name with arrow -> symbol.

Pointer to Data Members of class

We can use pointer to point to class's data members (Member variables).

Syntax for Declaration :

datatype class_name :: *pointer_name ;

Syntax for Assignment :

pointer_name = &class_name :: datamember_name ;

Both declaration and assignment can be done in a single statement too.

datatype class_name::*pointer_name = &class_name::datamember_name ;

Using with Objects

For accessing normal data members we use the dot . operator with object and -> qith pointer to object. But when we have a pointer to data member, we have to dereference that pointer to get what its pointing to, hence it becomes,

Object.*pointerToMember

and with pointer to object, it can be accessed by writing,

ObjectPointer->*pointerToMember

Lets take an example, to understand the complete concept.

```
class Data
{
 public:
 int a;
 void print() { cout << "a is "<< a; }
};

int main()
{
 Data d, *dp;
 dp = &d;     // pointer to object

 int Data::*ptr=&Data::a;   // pointer to data member 'a'

 d.*ptr=10;
 d.print();

 dp->*ptr=20;
 dp->print();
}
```

Output :

a is 10 a is 20

Pointer to Member Functions

Pointers can be used to point to class's Member functions.

Syntax :

return_type (class_name::*ptr_name) (argument_type) = &class_name::function_name ;

Below is an example to show how we use ppointer to member functions.

```
class Data
{ public:
  int f (float) { return 1; }
};

int (Data::*fp1) (float) = &Data::f;   // Declaration and assignment
int (Data::*fp2) (float);        // Only Declaration
```

```
int main(0
{
 fp2 = &Data::f;   // Assignment inside main()
}
```

Some Points to remember

1. You can change the value and behaviour of these pointers on runtime. That means, you can point it to other member function or member variable.
2. To have pointer to data member and member functions you need to make them public.

Constructors, Destructors and Operator overloading

Constructors:

A constructor is a special member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object (instance of class) create. It is special member function of the class.

How constructors are different from a normal member function?

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).
-

Default Constructors: Default constructor is the constructor which doesn't take any argument. It has no parameters.

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

class construct
{
public:
   int a, b;

      // Default Constructor
   construct()
   {
      a = 10;
      b = 20;
```

```
    }
};

int main()
{
        // Default constructor called automatically
        // when the object is created
    construct c;
    cout << "a: "<< c.a << endl << "b: "<< c.b;
    return 1;
}
```

Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly. The default value of variables is 0 in case of automatic initialization.

Parameterized Constructors:

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

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

class Point
{
    private:
        int x, y;
    public:
        // Parameterized Constructor
        Point(int x1, int y1)
        {
            x = x1;
            y = y1;
        }

        int getX()
        {
            return x;
        }
        int getY()
        {
            return y;
        }
    };
```

```
int main()
{
   // Constructor called
   Point p1(10, 15);

   // Access values assigned by constructor
   cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

   return 0;
}
```

Output:
p1.x = 10, p1.y = 15

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.
Example e = Example(0, 50); // Explicit call
Example e(0, 50); // Implicit call

Uses of Parameterized constructor:

1. It is used to initialize the various data elements of different objects with different values when they are created.
2. It is used to overload constructors.

Copy Constructor:

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
ClassName (const ClassName &old_obj);
```

Following is a simple example of copy constructor.

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

class Point
{
private:
   int x, y;
public:
   Point(int x1, int y1) { x = x1; y = y1; }

   // Copy constructor
   Point(const Point &p2) {x = p2.x; y = p2.y; }
```

```
    int getX()          {  return x; }
    int getY()          {  return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;
}
```

Run on IDE
Output:
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15

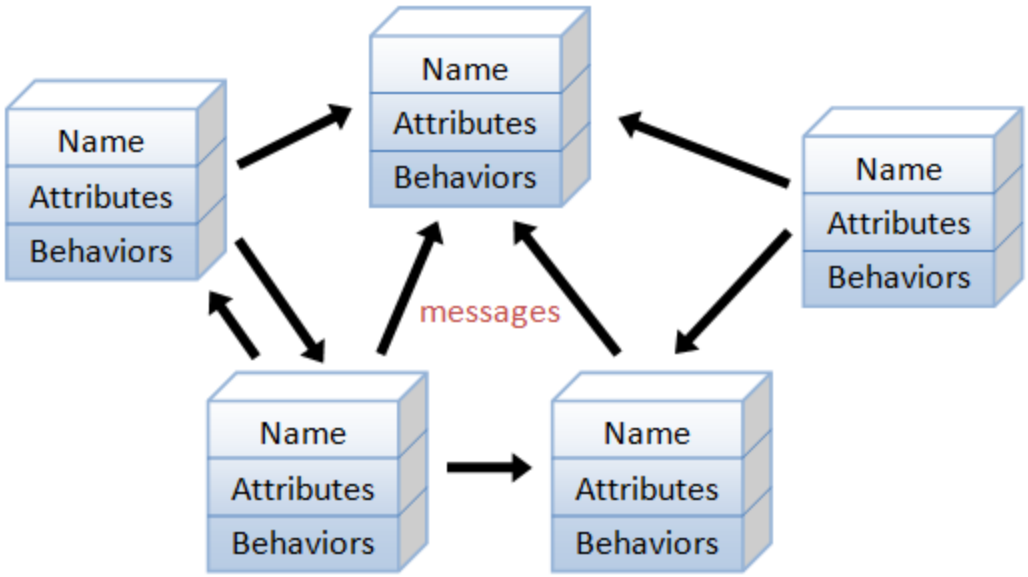
When is copy constructor called?
In C++, a Copy Constructor may be called in following cases:
1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When compiler generates a temporary object.

Multiple constructors

A class can have multiple constructors that assign the fields in different ways. Sometimes it's beneficial to specify every aspect of an object's data by assigning parameters to the fields, but other times it might be appropriate to define only one or a few.

```
Spot sp1, sp2;

void setup() {
  size(640, 360);
  background(204);
  noLoop();
  // Run the constructor without parameters
  sp1 = new Spot();
  // Run the constructor with three parameters
  sp2 = new Spot(width*0.5, height*0.5, 120);
```

```
}

void draw() {
  sp1.display();
  sp2.display();
}

class Spot {
  float x, y, radius;

  // First version of the Spot constructor;
  // the fields are assigned default values
  Spot() {
    radius = 40;
    x = width*0.25;
    y = height*0.5;
  }

  // Second version of the Spot constructor;
  // the fields are assigned with parameters
  Spot(float xpos, float ypos, float r) {
    x = xpos;
    y = ypos;
    radius = r;
  }
  void display() {
    ellipse(x, y, radius*2, radius*2);
  }

}
```



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

OOP languages permit *higher level of abstraction* for solving real-life problems. The traditional procedural language (such as C and Pascal) forces you to think in terms of the structure of the computer (e.g. memory bits and bytes, array, decision, loop) rather than thinking in terms of the problem you are trying to solve. The OOP languages (such as Java, C++, C#) let you think in the problem space, and use software objects to represent and abstract entities of the problem space to solve the problem.

The basic unit of OOP is a *class*, which encapsulates both the *static attributes* and *dynamic behaviors* within a "box", and specifies the public interface for using these boxes. Since the class is well-encapsulated (compared with the function), it is easier to reuse these classes. In other words, OOP combines the data structures and algorithms of a software entity inside the same box.

Constructors, Destructors and Operator overloading

Constructors

- A constructor is a member function of a class which initializes objects of a class.
- Constructor is automatically called when object(instance of class) create.
- It is special member function of the class. Becauseconstructor has same name as the class itself.
- Constructors don't have return type.
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Syntax: Class Class_Name

```
{
        //Private Mambers
        Public:
        Class_name(void);       //Constructor Declaraion
};
Class_name::Class_Name(void)     //Constructor Definition
{
  //Initialize Data Members
}
```

Characterstics:

6. They should be declared in the public section.
7. They are invoked directly when an object is created.
8. They don't have return type, not even void and hence can't return any values.
9. They can't be inherited; through a derived class, can call the base class constructor.
10. Like other C++ functions, they can have default arguments.
11. Constructors can't be virtual.
12. Constructors can be inside the class definition or outside the class definition.
13. Constructor can't be friend function.
14. They can't be used in union.
15. They make implicit calls to the operators new and delete when memory allocation is required.
16. When a constructor is declared for a class, initialization of the class objects becomes necessary after declaring the constructor.

Default Constructors: Default constructor is the constructor which doesn't take any argument. It has no parameters.

```
#include <iostream>
using namespace std;
class construct
{
public:
```

```
    int a, b;

       // Default Constructor
    construct()
    {
       a = 10;
       b = 20;
    }
};
 int main()
{
     // Default constructor called automatically
     // when the object‘c’ is created.
    construct c;
// Access values assigned by constructor
    cout<< "a: "<<c.a<<endl<< "b: "<<c.b;
    return 0;
}
```

Output: a:10
b:20

Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly. The default value of variables is 0 in case of automatic initialization.

Parameterized Constructors:

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

```
#include<iostream>
using namespace std;
class Point
{
   private:
      int x, y;
   public:
      // Parameterized Constructor
      Point(int x1, int y1)
      {
         x = x1;
         y = y1;
      }
      intgetX()
      {
         return x;
      }
```

```
    intgetY()
    {
        return y;
    }
};

int main()
{
    // Constructor called
    Point p1(10, 15);
    // Access values assigned by constructor
    cout<< "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    return 0;
}
```

Output:
p1.x = 10, p1.y = 15

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

```
Point p1 = Point(0, 50); // Explicit call
Point p1(0, 50);         // Implicit call
```

Uses of Parameterized constructor:

3. It is used to initialize the various data elements of different objects with different values when they are created.
4. It is used to overload constructors.

Copy Constructor:

- A copy constructor is a member function which initializes an object using another object of the same class.
- A copy constructor has the following general function prototype:

  ClassName (constClassName&old_obj);
- Acopy constructor may be called in following cases:
  1. When an object of the class is returned by value.
  2. When an object of the class is passed (to a function) by value as an argument.
  3. When an object is constructed based on another object of the same class.
  4. When compiler generates a temporary object.

Example

```
#include<iostream>
usingnamespacestd;

classPoint
{
private:
```

```
    intx, y;
  public:
    Point(intx1, inty1)
  {
  x = x1;
  y = y1;
  }
    // Copy constructor
    Point(Point &p2)
  {
  x = p2.x;
  y = p2.y;
  }
    intgetX(){  returnx; }
    intgetY(){  returny; }
  };

  intmain()
  {
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; or Point p2(p1); // Copy constructor is called here
    // Let us access values assigned by constructors
    cout<< "p1.x = "<< p1.getX() << ", p1.y = "<< p1.getY();
    cout<< "\np2.x = "<< p2.getX() << ", p2.y = "<< p2.getY();
    return0;
  }
```

Output:
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15

Multiple constructors/Constructor Overloading

- Constructor can be overloaded in a similar way as function overloading.
- Overloaded constructors have the same name (name of the class) but different number of arguments.
- Depending upon the number and type of arguments passed, specific constructor is called.
- Since, there are multiple constructors present, argument to the constructor should also be passed while creating an object.

Example:

```
#include <iostream>
using namespace std;
class Area
{
   private:
int length;
int breadth;
```

```
   public:
     // Constructor with no arguments
     Area()
         {
         length =5;
         breadth =2;
         }

     // Constructor with two arguments
     Area(int l, int b)
         {
         length =l;
         breadth =b;
         }

     void GetLength()
     {
cout<< "Enter length and breadth respectively: ";
cin>> length >> breadth;
     }

intAreaCalculation()
         {
         return length * breadth;
          }

     void DisplayArea(int temp)
     {
cout<< "Area: " << temp <<endl;
     }
};

int main()
{
   Area A1, A2(2, 1);
int temp;

cout<< "Default Area when no argument is passed." <<endl;
   temp = A1.AreaCalculation();
   A1.DisplayArea(temp);

cout<< "Area when (2,1) is passed as argument." <<endl;
   temp = A2.AreaCalculation();
   A2.DisplayArea(temp);
   return 0;
```

}

- For object A1, no argument is passed while creating the object.Thus, the constructor with no argument is invoked which initializes length to 5 and breadthto 2. Hence, area of the object A1 will be 10.
- For object A2, 2 and 1 are passed as arguments while creating the object.Thus, the constructor with two arguments is invoked which initializes length to l (2 in this case) and breadth to b (1 in this case). Hence, area of the object A2 will be 2.
- Output of above Program:

Default Area when no argument is passed.

Area: 10

Area when (2,1) is passed as argument.

Area: 2

Distructors

1. Destructors are special member functions of the class required to free the memory of the object whenever it goes out of scope.
2. Destructors are parameter less functions.
3. Name of the Destructor should be exactly same as that of name of the class. But preceded by '~' (tilde).
4. Destructors does not have any return type. Not even void.
5. The Destructor of class is automatically called when object goes out of scope.

Example:

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

class Marks
{
public:
intmaths;
int science;

  //constructor
  Marks() {
cout<< "Inside Constructor"<<endl;
cout<< "C++ Object created"<<endl;
  }

  //Destructor
  ~Marks() {
cout<< "Inside Destructor"<<endl;
cout<< "C++ Object destructed"<<endl;
```

```
  }
};

int main( )
{
  Marks m1;
  Marks m2;
  return 0;
}
```

Output

```
Inside Constructor
C++ Object created
Inside Constructor
C++ Object created

Inside Destructor
C++ Object destructed
Inside Destructor
C++ Object destructed
```

Operator Overloading

- The meaning of an operator is always same for variable of basic types like: int, float, double etc. For example: To add two integers, + operator is used.
- However, for user-defined types (like: objects), you can redefine the way operator works. For example: If there are two objects of a class that contains string as its data members. You can redefine the meaning of + operator and use it to concatenate those strings.
- This feature allows programmer to redefine the meaning of an operator (when they operate on class objects) is known as operator overloading.
- Why is operator overloading used?You can write any C++ program without the knowledge of operator overloading. However, operator operating are profoundly used by programmers to make program intuitive. For example, You can replace the code like:
  calculation = add(multiply(a, b),divide(a, b));
  to
  calculation = (a*b)+(a/b);
- How to overload operators?To overload an operator, a special operator function is defined inside the class as:

```
class className
{
... .. ...
public
returnType operator symbol (arguments)
{
... .. ...
}
```

```
... .. ...
};
```

- Here, returnType is the return type of the function.The returnType of the function is followed by operator keyword.Symbol is the operator symbol you want to overload. Like: +, <, -, ++. You can pass arguments to the operator function in similar way as functions.

Things to remember

- Operator overloading allows you to redefine the way operator works for user-defined types only (objects, structures). It cannot be used for built-in types (int, float, char etc.).
- Two operators = and & are already overloaded by default in C++. For example: To copy objects of same class, you can directly use = operator. You do not need to create an operator function.
- Operator overloading cannot change the precedence and associatively of operators. However, if you want to change the order of evaluation, parenthesis should be used.
- There are 4 operators that cannot be overloaded in C++. They are :: (scope resolution), . (member selection), .* (member selection through pointer to function) and ?:(ternary operator).

Example:

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

class Test
{
  private:
int count;

  public:
     Test(): count(5){}

     void operator ++()
     {
       count = count+1;
     }
     void Display() { cout<<"Count: "<<count; }
};

int main()
{
   Test t;
   // this calls "function void operator ++()" function
   ++t;
t.Display();
   return 0;
}
```

Output

Count: 6
This function is called when ++ operator operates on the object of Test class (object t in this case).In the program,void operator ++ () operator function is defined (inside Test class).This function increments the value of count by 1 for t object.

Unary Operator Overloading

Prefix Increment ++ Operator Overloading

```
#include <iostream>
using namespace std;
class Check
{
   private:
inti;
   public:
     Check(): i(0) {  }
     void operator ++()
       { ++i; }
     void Display()
       { cout<<"i=" <<i<<endl; }
};


int main()
{
   Check obj;
   // Displays the value of data member i for object obj
obj.Display();
   // Invokes operator function void operator ++( )
   ++obj;
   // Displays the value of data member i for object obj
obj.Display();
   return 0;
}
```

Output
i=0
i=1

- Initially when the object obj is declared, the value of data member i for object obj is 0 (constructor initializes i to 0).
- When ++ operator is operated on obj, operator function void operator++( ) is invoked which increases the value of data member i to 1.**Postfix Increment ++ Operator Overloading**

```
#include <iostream>
using namespace std;
class Check
{
  private:
```

```
inti;
  public:
    Check(): i(0) {  }
    Check operator ++ ()
    {
       Check temp;
temp.i = ++i;
       return temp;
    }
    // Notice int inside barcket which indicates postfix increment.

    Check operator ++ (int)
    {
       Check temp;
temp.i = i++;
       return temp;
    }
    void Display()
    { cout<<"i = "<<i<<endl; }
};

int main()
{
    Check obj, obj1;
obj.Display();
    obj1.Display();
    // Operator function is called, only then value of obj is assigned to obj1
    obj1 = ++obj;
obj.Display();
    obj1.Display();
    // Assigns value of obj to obj1, only then operator function is called.
    obj1 = obj++;
obj.Display();
    obj1.Display();
    return 0;
}
```

Output

```
i = 0
i = 0
i = 1
i = 1
i = 2
i = 1
```

- When increment operator is overloaded in prefix form; Check operator ++ () is called but, when increment operator is overloaded in postfix form; Check operator ++ (int) is invoked.

- Notice, the int inside bracket. This int gives information to the compiler that it is the postfix version of operator.
- Don't confuse this int doesn't indicate integer.

Program for unary minus (-) operator overloading.

```
#include<iostream>
using namespace std;
class NUM
{
   private:
      int n;
   public:
      //function to get number
      void getNum(int x)
      {
         n=x;
      }
      //function to display number
      void dispNum(void)
      {
         cout<< "value of n is: " << n;
      }
      //unary - operator overloading
      void operator - (void)
      {
         n=-n;
      }
};
int main()
{
   NUM num;
   num.getNum(10);
   -num;
   num.dispNum();
   cout<<endl;
   return 0;
}
```

Binary Operators Overloading

- The binary operators take two arguments
- When + operator is operated on obj1 and obj2, operator function complex operator+( ) is invoked which will add complex numbers.
- When - operator is operated on obj1 and obj2, operator function complex operator-( ) is invoked which will substract complex numbers.
- Right side of the binary operator is the parameter for operator function definition.

Example:

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

```
class complex {
int a, b;
public:

   void getvalue() {
cout<< "Enter the value of Complex Numbers a,b:";
cin>> a>>b;
   }

   complex operator+(complex ob) {
      complex t;
t.a = a + ob.a;
t.b = b + ob.b;
      return (t);
   }

   complex operator-(complex ob) {
      complex t;
t.a = a - ob.a;
t.b = b - ob.b;
      return (t);
   }

   void display() {
cout<< a << "+" << b << "i" << "\n";
   }
};

void main()
{
   complex obj1, obj2, result, result1;
   obj1.getvalue();
   obj2.getvalue();
   result = obj1 + obj2;
   result1 = obj1 - obj2;
cout<< "Input Values:\n";
   obj1.display();
   obj2.display();
cout<< "Result:";
result.display();
   result1.display();
return 0;
}
Output:
Enter the value of Complex Numbers a, b
4               5
```

```
Enter the value of Complex Numbers a, b
2               2
Input Values
4 + 5i
2 + 2i
Result
6 + 7i
2 + 3i
```

String manpulation using operator overloading

- We shall able to use statements like string3 = string1 + string2; if (string1 >= string2) string3 = string1;
- Here strings are class objects which can be manipulate.
- We use new to allocate memory for each string and pointer variable to point the string array.
- Length and location are necessary for string manipulation.
- Typical string class is

```
Class string
{
   char *p;    //pointer to string
int len;     //length of string
   public:
     ......
  };
```

Example:

```
        Class string
        {
           char *p;    //pointer to string
           int len;     //length of string
           public:
string()
                  {
                  len =0;
                  p=0;
                  }
        string(char *s)
                  {
                  len = strlen(s);
                  P = new char[len+1];
                  strcpy(p,s);
                  }
             string()
                  {
                  delete p;
                  }
friend string opertor+(string &s, string &t);
friend int opertor<=(string &s, string &t);
```

```
        };
string opertor+(string &s, string &t)
                {
                string temp;
                temp.len = s.len +t.len;
                temp.p = new char[temp.len+1];
                strcpy(temp.p,s.p);
                strcat(tem.p,t.p);
                return (temp);
                }

int opertor<=(string &s, string &t)
                {
                int m = strlen(s.p);
                int n =strlen(t.p);
                if(m<=n) return 1;
                else return 0;
                }
                void show()
                {
                cout<<s.p;
                }

                int main()
                {
                string s1 =”New”;
                string s2 =”York”;
                string s3 =”Delhi”;
                string t1,t2,t3;
                t1=s1;
                t2=s2;
                t3=s1+s3;
                if(t1<=t3)
                {
                show(t1);
                cout<<”samller than”;
                show(t3);
                }
                esle
                {
                show(t3);
                cout<<”samller than”;
                show(t1);
                }
```

```
return 0;
}
```

Output: New smaller than New Delhi