C++ language
Lesson -1
output & input

C++ language

The C++ language is a significant extension to the syntax of C.
The major Innovations are the constructs which facilitate the object-oriented approach to software design and development.

C++ also includes many changes to the syntax of C which are less fundamental but nonetheless important in improving the language and in removing ambiguities and inconsistencies to be found in C.

C++ language

Significant extension to C

object-oriented approach

Standard Stream

'stdio.h' is replaced by a new header file, 'iostream.h'. The iostream.h was a header file used by the early 1990's I/O streams library. This was developed at AT&T for use with early C++. In that time C++ was not standardized.

Now <iostream> header file used by the C++ standard library. This is first published in 1998, to provide access to standard I/O streams.

C++ comes with libraries that provide us with many ways for performing input and output are performed in the form of a sequence of bytes or more commonly known as **streams**.

- **Input Stream:** If the direction of flow of bytes is from the device(for example, Keyboard) to the main memory then this process is called input.
- **Output Stream:** If the direction of flow of bytes is opposite, i.e. from main memory to device(display screen) then this process is called output.

cout

The *iostream* library contains a few predefined variables for us to use. One of the most useful is **std::cout**, which allows us to send data to the console to be printed as text. *cout* stands for "console output".
If we want to print separate lines of output to the console, we need to tell the console when to move the cursor to the next line.
One way to do that is to use ***std::endl***.

```
std::cout << "My name is Youssef ." << std::endl;
```

Note: "\n" equivalent to std::endl;

cin

The std::cin object in C++ is an object of class iostream. It is used to accept the input from the standard input device i.e. keyboard.
It is associated with the standard C input stream stdin.

```
std::cout << "Enter n: ";
std::cin >> n;
```

```
string mystr;

std::cout << "What's your name? ";
getline ( std::cin, mystr);
```

Standard Stream

C language

```
#include <stdio.h>

printf("%d",x);
printf("Hello youssf\n");

scanf("%d",&x);
```

C++ language

```
#include<iostream>

std::cout << x;
std::cout << "Hello Youssef"<<std::endl;

std::cin  >> x;
```

What is the difference between main() in C and C++

C language

```
void main(void)
{

}
```

```
void main(int argc, char *argv[])
{

}
```

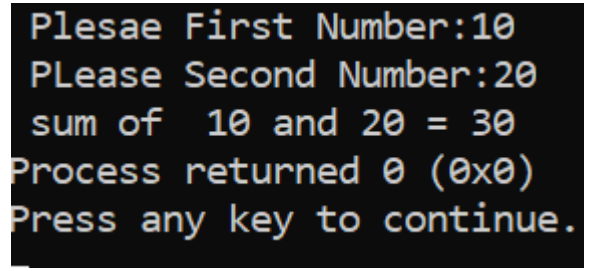
C++ language

```
int main()
{


    return 0;
}
```

```
int main(int argc, char *argv[])
{



   return 0;
}
```

First Program in C++

```
#include <iostream>

int main()
{
   int num1,num2;
   int sum;

   std::cout <<" Plesae First Number:";
   std::cin >> num1;
   std::cout <<" PLease Second Number:";
   std::cin >> num2;

   sum = num1+num2 ;
   std::cout << " sum of  "<< num1<<" and " << num2<<" = " << sum ;

   return 0;
}
```



using namespace std;

iostream is a C++ code file (more specifically a header file, but that doesn't matter). It contains the code that defines cout, cin, endl, etc. If we want to use them, we need them to be in our code, that's why we include it.

```
// iostream header file

namespace std
{
   ostream cout;
   istream  cin;
   // etc etc
};
```

std::cout
std::cin

Using the using keyword doesn't mean we add functionality, it means we say that we read things by default. If we say **using namespace std;** then we say: If we come across an object name that doesn't exist in our current namespace, check if there exists a namespace std in which it does exist, and use that object.

using namespace std;

cout
cin

Why std::cout instead of cout only ??

In the C++ standard, cout is defined in the **std** namespace, so you need to say std::cout or ***using namespace std***;

```
#include <iostream>
Using namespace std;
int main()
{
   int num1,num2;
   int sum;

   cout <<" Plesae First Number:";
   cin >> num1;
   cout <<" PLease Second Number:";
   cin >> num2;

   sum = num1+num2 ;
   cout << " sum of  "<< num1<<" and " << num2<<" = " << sum ;

   return 0;
}
```

```
 Plesae First Number:10
 PLease Second Number:20
 sum of  10 and 20 = 30
Process returned 0 (0x0)
Press any key to continue.
```

C++ language
Lesson -2
namespace

function of ***namespace*** in c++

A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it.

Namespaces are **used to organize code into logical groups and to prevent name collisions** that can occur especially when your code base includes multiple libraries.

namespace

```
1   #include <iostream>
2   using namespace std;
3   namespace youssef
4   {
5       int a =10;
6       int b= 2;
7   }
8
9   namespace mohab
10  {
11      int a=100;
12  }
13  int main()
14  {
15
16      cout << " a= "<< a;
17
18      return 0;
19  }
20
```

File	Line	Message
		=== Build: Debug in testdebug (compiler: GNU GCC Compiler) ===
D:\deb\testdeb...		In function 'int main()':
D:\deb\testdeb...	16	error: 'a' was not declared in this scope
D:\deb\testdeb...	16	note: suggested alternatives:
D:\deb\testdeb...	5	note: 'youssef::a'
D:\deb\testdeb...	11	note: 'mohab::a'
		=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===

namespace

```cpp
#include <iostream>
using namespace std;
namespace youssef
{
    int a =10;
    int b= 2;
}

namespace mohab
{
    int a=100;
}
int main()
{

    cout << " a= "<< youssef::a;

    return 0;
}

```

```
 a= 10
Process returned 0 (0x0)
Press any key to continue.
```

Namespaces

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

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace.

```
namespace identifier
{
    entities
}
```

Example

```
namespace myNamespace
{
  int a, b;
}
```

In this case, the variables a and b are normal variables declared within a namespace called myNamespace. In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::

For example, to access the previous variables from outside myNamespace we can write:

myNamespace::a ***or*** ***myNamespace::b***

Note:

The keyword ***using*** is used to introduce a name from a namespace into the current declarative region

using namespace myNamespace;

to access the previous variables from outside myNamespace we can write **a** or **b** without the namespace.

using namespace std;

iostream is a C++ code file (more specifically a header file, but that doesn't matter). It contains the code that defines cout, cin, endl, etc. If we want to use them, we need them to be in our code, that's why we include it.

```
// iostream header file

namespace std
{
   ostream cout;
   istream  cin;
   // etc etc
};
```

std::cout
std::cin

Using the using keyword doesn't mean we add functionality, it means we say that we read things by default. If we say **using namespace std;** then we say: If we come across an object name that doesn't exist in our current namespace, check if there exists a namespace std in which it does exist, and use that object.

using namespace std;

cout
cin

C++ language
Lesson -3
variables and data types

Variable Declarations

In C++, all declarations are statements. In C, this is not so and declarations must be placed at the start of a compound statement, usually that representing the function 'body'.

```
void myfunc ()
{
  int x;    // valid C and C++
  x=9;
  int y;    // valid C++, invalid C
  {
      int z;      // valid C and C++
  }
}
```

Because declarations are statements, they may appear any where in C++ (but not C) code:

```
for (int x=5;x<10;x++)
```

Data Types

In C++, the name (tag) of a structure or enumerator is its type. This means that the name of a structure, union or enumeration type is its type.

In C, the type of a structure, Union or enumeration type is the tag prefixed with the keywords 'struct', 'union' or 'enum'.

```
struct  filerec
{
        // some member declarations here
};

union one_member
{
        // some member declarations here
};

enum seasons {spring, summer, autumn, winter);
```

C/C++ language

```
struct filerec record;
union one-member union1;
enum seasons quarter1;
```

C++ language

```
filerec record;
one-member union1;
seasons quarter1;
```

new Types bool

One of the new data type is: **bool**
Syntax:

bool b1 = true; // declaring a Boolean variable with true value

In C++, the data type bool has been introduced to hold a Boolean value, true or false.
The values true or false have been added as keywords in the C++ language.
Important Points:

The default numeric value of true is 1 and false is 0.

It is also possible to convert implicitly the data type integers or floating point values to bool type

new Types string Class

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality.

One of the most useful data types supplied in the C++ libraries is the string.
A string is a variable that stores a sequence of letters or other characters, such as "Hello" or “My name is youssef!". Just like the other data types, to create a string we first declare it, then we can store a value in it.

string testString;
testString = “My name is Youssef.";

We can combine these two statements into one line:
string testString = “My name is Youssef.";
Often, we use strings as output, and cout works exactly like one would expect:

cout << testString << endl;

This string class is a part of the std namespace and is defined in the header <string>.
String header <string> needs to be included in the program to use the String class.

string methods

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string testString="My name is Youssef.";

   cout << testString.substr(7, 3) << endl;
   cout << testString.substr(7) << endl;
   testString.erase(7, 4);
   cout << testString << endl;
   testString.erase(testString.begin() + 5, testString.end() - 3);
   cout << testString << endl;
}
```

```
is
is Youssef.
My nameYoussef.
My naef.
```

data types in C++

C language

char
int
float
double

C++ language

char
int
float
double
bool
string

Example with new data types

string

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

int main()
{
    string name;
    int i;

    cout<<" Plesae your name:";
    cin>>name;
    cout<<" Hello ,"<<name<<endl;

    i=0;
    while(name[i])
    {
        cout<<name[i++]<<endl;
    }
    return 0;
}
```

```
 Plesae your name:youssef
 Hello ,youssef
y
o
u
s
s
e
f
```

bool

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

int main()
{
    bool ans;

    cout << "are you students ?:";
    cin >> ans;

    cout<< ans;
}
```

```
are you students ?:true
0
```

```
are you students ?:false
0
```

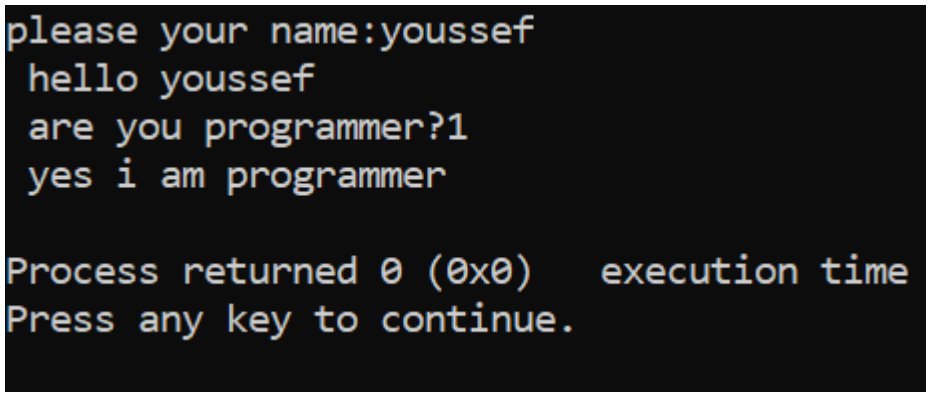
Why ?

```
are you students ?:1
1
```

```
are you students ?:0
0
```

Example with new data types

```
#include <iostream>
using namespace std;
int main()
{
    string myname;
    bool ans;

    cout <<"please your name:";
    cin >> myname;
    cout <<" hello "<< myname<<endl;

    cout <<" are you programmer?";
    cin >> ans;

    if (ans==true)
        cout << " yes i am programmer "<<endl;
    else
        cout << " no i am not programmer "<<endl;
}
```



C++ language
Lesson -4
io manipulation

Formatted Console I/O

In C++, the formatted console input/output functions are used for performing input/output operations at the console by formatting the data in a particular format.

C++ supports a number of features that could be used for formatting the output/input.

- ios class functions and flags included in <iostream>.
- Manipulators. Included in <iomanip>
- User-defined output functions

Formatted output
width – fill – precession

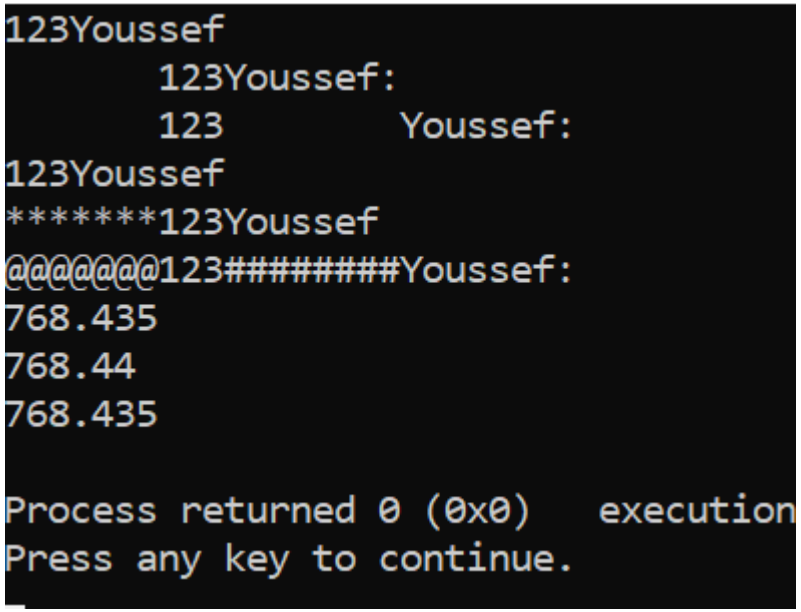
class functions

cout.width(int)
cout.fill(char)
cout.precision(int)

Manipulators functions

setw(int)
fill(char)
setprecision(int)

```
#include<iostream>
#include<iomanip>

using namespace std;
int main() {
		string text="Youssef";
		int number=123;
		float fnum = 768.43536;


		cout << number<<text<<endl;
		cout.width(10);
		cout << number<<text<<":"<<endl;
		cout <<setw(10)<<number<<setw(15)<<text<<":"<<endl;
		cout << number<<text<<endl;
		cout.width(10);
		cout.fill('*');
		cout << number<<text<<endl;
		cout <<setw(10)<<setfill('@')<<number<<setw(15)<<setfill('#')<<text<<":"<<endl;
		cout<<fnum<<endl;
		cout.precision(5);
		cout<<fnum<<endl;
		cout<<setprecision(6)<<fnum<<endl;

		return 0;
}
```



Formatted output
numeric base for integer

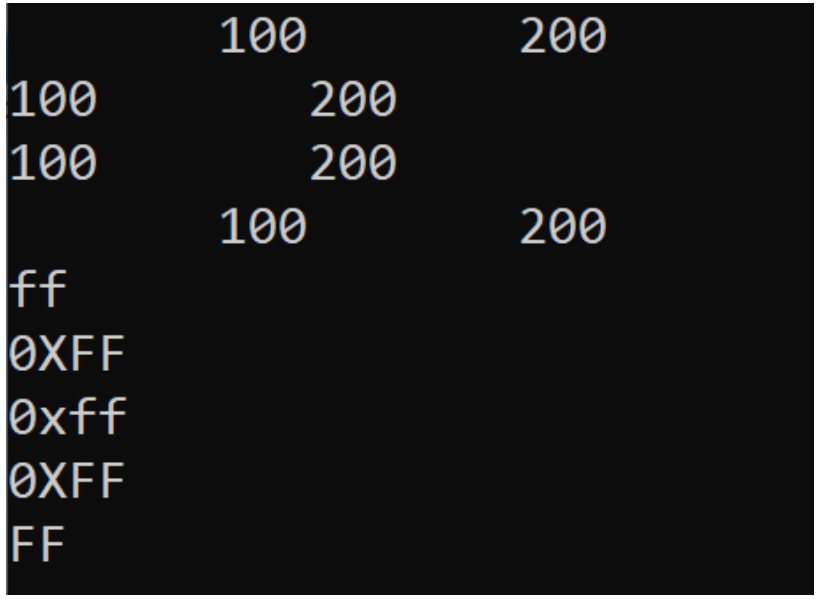
oct hex

```
#include<iostream>
#include<iomanip>

using namespace std;
int main() {

                int number=65;

   // Modifies the default numeric base for integer I/O.

                cout << "number in decimal\t\t="<<number<<endl;
                cout << "number in octal\t\t\t="<<oct<<number<<endl;
                cout << "number in Hexadecimal\t\t="<<hex<<number<<endl;
   cout <<endl;
                cout << "number in decimal\t\t="<<setbase(10)<<number<<endl;
                cout << "number in octal\t\t\t="<<setbase(8)<<number<<endl;
                cout << "number in Hexadecimal\t\t="<<setbase(16)<<number<<endl;
                cout <<endl;
                cout << " showbase flag : \t" << std::showbase
   << setbase(10)<<number << " "
                << setbase(8)<<number << " "
                <<setbase(16)<<number<<endl;
                cout << " noshowbase flag : \t" << std::noshowbase
   << setbase(10)<<number << " "
                << setbase(8)<<number << " "
                <<setbase(16)<<number<<endl;
                return 0;
}
```

```
number in decimal               =65
number in octal                 =101
number in Hexadecimal           =41

number in decimal               =65
number in octal                 =101
number in Hexadecimal           =41

 showbase flag :        65 0101 0x41
 noshowbase flag :      65 101 41
```

Setiosflags() / cout.setf()
resetiosflags()/ cout.unset()

```
#include <iostream>
#include <iomanip>
using namespace std;
int main ()
{
   cout << setw(10)<<100<<setw(10)<<200<<endl;
   cout << setw(10)<<setiosflags(ios::left)<<100<<setw(10)<<200<<endl;
   cout << setw(10)<<100<<setw(10)<<200<<endl;
   cout << setw(10)<<setiosflags(ios::right)<<100<<setw(10)<<200<<endl;
   // or resetiosflags(ios::left)

  cout << hex;
  cout << 255 << endl;
  cout << setiosflags (ios::showbase | ios::uppercase);
  cout << 255 << endl;
  cout<<nouppercase<< 255 << endl;
  cout.setf(ios::uppercase);
  cout <<255<<endl;
  cout.unsetf ( std::ios::showbase );
  cout <<255<<endl;
  return 0;
}
```



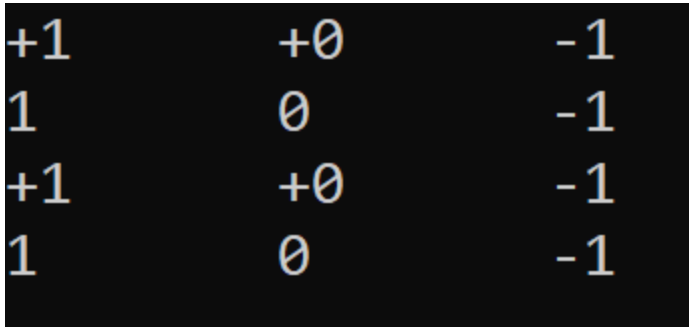
stream format flags

Bitmask type to represent stream format flags.
This type is used as its parameter and/or return value by the member functions flags, setf and unsetf.
The values of these constants can be combined into a single fmtflags value using the OR bitwise operator (|).
The values passed and retrieved by these functions can be any valid combination of the following member constants:

field	member constant	effect when set
independent flags	boolalpha	read/write bool elements as alphabetic strings (true and false).
	showbase	write integral values preceded by their corresponding numeric base prefix.
	showpoint	write floating-point values including always the decimal point.
	showpos	write non-negative numerical values preceded by a plus sign (+).
	skipws	skip leading whitespaces on certain input operations.
	unitbuf	flush output after each inserting operation.
	uppercase	write uppercase letters replacing lowercase letters in certain insertion operations.
numerical base (basefield)	dec	read/write integral values using decimal base format.
	hex	read/write integral values using hexadecimal base format.
	oct	read/write integral values using octal base format.
float format (floatfield)	fixed	write floating point values in fixed-point notation.
	scientific	write floating-point values in scientific notation.
adjustment (adjustfield)	internal	the output is padded to the field width by inserting fill characters at a specified internal point.
	left	the output is padded to the field width appending fill characters at the end.
	right	the output is padded to the field width by inserting fill characters at the beginning.

Setiosflags() / cout.setf()
resetiosflags()/ cout.unset()

```
#include <iostream>
#include <iomanip>
using namespace std;
int main () {
  int x = 1;
  int y = 0;
  int z = -1;
  cout << showpos   << x << '\t' << y << '\t' << z << '\n';
  cout << noshowpos << x << '\t' << y << '\t' << z << '\n';
  cout.setf(ios::showpos);
  cout << x << '\t' << y << '\t' << z << '\n';
  cout<<resetiosflags(ios::showpos)<< x << '\t' << y << '\t' << z << '\n';

  return 0;
}
```



Setiosflags() / cout.setf()
resetiosflags()/ cout.unset()

```
#include <iostream>
using namespace std;
int main () {
  bool b = true;
  cout << boolalpha << b << '\n';
  cout << noboolalpha << b << '\n';

  bool ans;

  cout << " are you student?:";
  cin >>boolalpha>>ans;
  cout << boolalpha<< ans;
  return 0;
}
```

```
true
1
 are you student?:true
true
```

C++ language
Lesson -5
new keywords and operators

New Keywords And Operators in C++

The following keywords are reserved for use by C++ and not by C. any C program which uses one of these keywords as an identifier is not a valid C++ program.

New keywords in C++

class	delete	this	virtual
friend	inline	new	operator
private	protected	public	template

Operator	Meaning
new	Allocate memory space
delete	De-allocate memory space
::	Scope resolution

Scope Resolution Operator ::

c++ introduces the scope resolution operator**'::',** which is used to render unambiguous the Scope of an identifier A structure in C++ defines its own scope.

Scope resolution operator (::) in C++ is used when we want to use a global variable but also has a local variable with the same name or to access Enum inside struct out side struct since In c, such declarations are visible outside the enclosing structure or define a function outside a class .

```cpp
#include <iostream>

using namespace std;

char c = 'a';     // global variable (accessible to all functions)

int main() {
  char c = 'b';   // local variable (accessible only in main function)

  cout << "Local  variable: " << c << "\n";
  cout << "Global variable: " << ::c << "\n";  // Using scope resolution operator

  return 0;
}
```

```
struct filerec
{
   struct table {/* table members here*/};
   enum quarters
        { spring, summer, autumn, winter} ;
        .
        .
} ;
```

In C++, the names table, quarters and all four enumerators are not visible outside the scope of the structure type 'filerec'. In C, all these names are visible in the scope enclosing the declaration of 'filerec'. In C++ syntax requires:

```
table  sales_list;
```

is illegal because 'table' is not in scope; in C it is in scope. C++ syntax requires:

```
filerec::table sales_list;
```

with the scope of 'table' being resolved as being within that of 'filerec' by the scop-eresolution operator '::'.

new And delete Operators

'new' and 'delete' are the C++ replacements for the C library functions 'malloc' and 'free'. 'malloc' and 'free' may still be called, but 'new' and 'delete' are easier and safer to use.

The 'new' operator is almost always used in one of the following general forms**:**

<ptr> = new <type> (<initial value<)
<ptr> = new <type> [<size>] ;

new returns to the pointer on the left-hand side of the assignment a pointer to the memory allocated. The pointer is of the type specified on the right-hand side of the assignment. If for any reason the memory cannot be allocated, 'new' returns a NULL pointer, which may be used in application code to check for a memory allocation error.

The general forms of the 'delete' operator are these:

delete <ptr>;
delete [ ] <ptr>;

These de-allocate the space pointed to by the pointer '<ptr>' which was previously allocated by 'new'.

Dynamic Memory Allocation

As an alternative to the 'malloc' and 'free' function family of C, C++ introduces the 'new' and 'delete' operators, which do dynamic memory allocation like this:

Passing arguments to functions 'by reference' in C is in fact simulated using pointers; the C++ reference declaration also allows true function call-by-reference. The following program initializes a reference declaration.

'new' allocates memory heap space ==========➔ (casting) malloc(sizeof(….))
'delete' ==========➔ free()

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

int main ()
{
   int  *x;

    x = new int;
    *x = 5;
   cout << "Allocated-integer value:" << *x << "\n";
}
```

new And delete example

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

int main()
{
	int *iptr1 , *iptr2 , *iptr3 ;

	iptr1 = new int (5) ;

	iptr2 = new (int) (6) ;

	if ((iptr3 = new int [20]) == NULL )
		cout <<"Couldn't Allocate Array\n" ;

	// Display First Two Integer Values
	cout <<"Integer 1 :  "<< *iptr1 << "\n" ;
	cout <<"Integer 2 :  "<< *iptr2 << "\n" ;
	delete (iptr1) ;
	delete iptr2 ;

	delete [ ] iptr3 ;
}
```

```
Integer 1 :  5
Integer 2 :  6
```

In older Release of the C++ language definition, it was necessary to include. a size specifier between the square brackets in the final 'delete' operation:

delete [20] iptr3;

Implement Variable Length Array

```
#include<iostream>
using namespace std;
int main() {
		int x, n;
		cout << "How many numbers will you type?: " ;
		cin >>n;
		int *arr = new int[n];



		cout << "Enter " << n << " numbers" << endl;
		for (x = 0; x < n; x++)
		 {
		     cin >> arr[x];
		 }
		cout << "You typed: ";
		for (x = 0; x < n; x++) {
				cout << arr[x] << " ";
		}
		cout << endl;
		delete [] arr;

		return 0;
}
```

```
How many numbers will you type?: 3
Enter 3 numbers
1
2
3
You typed: 1 2 3

Process returned 0 (0x0)   execution time : 4.969 s
Press any key to continue.
```

Note :

C++ provides facilities for ***overloading*** the meaning of almost all its operators. ***Operator overloading*** is not provided in C syntax.

Implement Stack using linked list in c++

```
#include <stdio.h>
#include <iostream>
#include <conio.h>
using namespace std;
struct Node {
  int data;
  struct Node *next;
};
void display() ;
void push(int val);
void pop() ;

Node* top = NULL;

int main()
{
  int sel, val;


   #ifdef _WIN32
        system("cls");
   #elif __linux__
        system("clear");
   #else
        printf("Unidentified OS\n");
   #endif

 do {

    #ifdef _WIN32
        system("cls");
    #elif __linux__
        system("clear");
    #else
        printf("Unidentified OS\n");
    #endif
    cout<<"1) Push in stack"<<endl;
    cout<<"2) Pop from stack"<<endl;
    cout<<"3) Display stack"<<endl;
    cout<<"4) Exit"<<endl;
    cout<<"Enter choice: ";
    cin>>sel;
```

```
 switch(sel)
    {
      case 1:
        cout<<"Enter value to be pushed:";
        cin>>val;
        push(val);
        break;
      case 2:
        pop();
        break;
      case 3:
        display();
        break;
      case 4:
         cout<<"Exit"<<endl;
        break;
      default:
        cout<<"Invalid Choice"<<endl;
    }
  } while(sel!=4);

  return 0;
}
void push(int val)
{
  //struct Node* newnode = (struct Node*) malloc(sizeof(struct Node));
  Node * newnode = (Node*) new Node;
  newnode->data = val;
  newnode->next = top;
  top = newnode;
}
void pop()
{
   Node* ptr;
  if(top==NULL)
  {
    cout<<"Stack Underflow"<<endl;
    cout << "Press any key to continue !!";
    char ans=getch();
  }
  else {
    cout<<"The popped element is "<< top->data <<endl;
    cout << "Press any key to continue !!";
    char ans=getch();
    ptr=top;
    top = top->next;
    delete ptr;
  }
}
```

```
void display()
{
  Node* ptr;
  if(top==NULL)
  cout<<"stack is empty";
  else {
    ptr = top;
    cout<<"Stack elements are: ";
    while (ptr != NULL) {
      cout<< ptr->data <<" ";
      ptr = ptr->next;
    }
  }
  cout<<endl;
  cout << "Press any key to continue !!";
  char ans=getch();
}
```

```
1) Push in stack
2) Pop from stack
3) Display stack
4) Exit
Enter choice: 1
Enter value to be pushed:10
```

Use Operator overloading

In C++, we can make operators to work for user defined classes.

This means C++ has the ability to provide the operators with a ***special meaning for a data type***, this ability is known as ***operator overloading.***

For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

```
#include <stdio.h>
#include <string.h>

int main()
{
   char * str1="Youssef"; // run time error must allocate space and use strcpy function not use assignment operator
   char * str2="Shawky"; // run time error must allocate space and use strcpy function not use assignment operator

   char * str3; // error  must allocate space

   //str3 = str1 + str2 ;  // syntax error  invalid use of + operator with char *

    if (str1==str2)      //run time error must use strcmp  function not use Equality operator
       printf(" Equal string \n");
   else
       printf(" Not equal string \n");

   printf("%s\b",str1);
   printf("%s\b",str2);
   printf("%s\n",str3);

}
```

```
#include <iostream>
#include <string>
using namespace std;
// you can Assigns a new value to the string, replacing its current contents using = operator .
// You can concatenate two string objects in C++ using + operator.
// You can Performs the appropriate comparison operation between the string objects using relational operators
// all this feature called operator overloading
int main()
{
   string str1="Youssef";
   string str2="Shawky";
   string str3;

   str3 = str1 + str2 ;

   if (str1==str2)
      cout << " Equal string "<<endl;
   else
      cout << " Not equal string "<<endl;

   cout << str1<<endl;
   cout << str2<<endl;
   cout << str3<<endl;
}
```

C ++ language

C language

```
#include <stdio.h>
#include <string.h>
int main()
{
   char * str1;
   char * str2;
   char * str3;

   str1=malloc(7);
   if(str1!=NULL)   strcpy(str1,"Youssef");
   str2=malloc(6);
   if(str2!=NULL)   strcpy(str2,"Shawky");
   str3 = malloc (13);
   if (str3!=NULL)
   {
      strcpy(str3,"");
      strcat(str3,str1);
      strcat(str3,str2);
   }
   if (strcmp(str1,str2)==0)
      printf(" Equal string \n");
   else
      printf(" Not equal string \n");
   printf("%s\n",str1);
   printf("%s\n",str2);
   printf("%s\n",str3);
}
```

C++ language
Lesson -6
inline Function

Inline Functions

In C++,but not in C, a function prototype and definition may be prefixed with the keyword 'inline'.

This does not in any way change the meaning of the programmer of the call to the inline function. Instead, the compiler is requested to optimize the function call by substituting the function's code at the point where the function call is made.

With the use of the inline function, at the compiling time, the whole code of the inline function gets inserted or substituted. It is done at the point of the inline function call by the C++ compiler.

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

// prototype declaration
inline int leap(int) ;

int main (void)
{

     int year;

     cout << "Enter year number:";
     cin >> year;
     if (leap (year))
         cout << year<< " is a leap year\n";
}
//   'leap'definition
 inline int leap(int year)
{
     if (((year % 4) != 0) &&((year % 400) != 0))
     return (1);
}
```

In the example, 'leap' is declared before 'main' and the compiler knows what code to substitute for the function call in 'main', the function definition is its declaration; the prototype may be omitted without affecting program correctness.

Note :

1- Functions which are defined (rather than just declared) as part of a class in C++ are implicitly inline.

2- Inline functions must not contain definitions of ***static*** variables, ***loop*** statements, ***'switch***' statements or ***'goto***' statements. Inline functions must not define ***arrays*** or be ***recursive***.

Macros and inline function

Preprocessor macros are often used in C for efficiency: they void the overhead of a function call.
In C++, their use is often replaced by inline functions.

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

//Preprocessor Macro
#define min(a , b) ((a) < (b) ? (a) : (b))

//Equivalent C++ Inline Function
inline int MIN(int a , int b )
 {
   return (a < b ? a: b) ;
}


int main(void)
{
int x = 5 ;
int y = 5 ;

cout <<"Preprocessor Minimum using #define is "<< min (x++, 7) << "\n" ;
cout <<"C++ Code Minimum using inline is "<< MIN (y++, 7) << "\n" ;
return 0;
}
```

```
Preprocessor Minimum using #define is 6
C++ Code Minimum using inline is 5
```

Which shows that an inline. function can be used to avoid preprocessor problems while equaling the gain in efficiency offered by preprocessor macros.

Main Differences Between Inline Function And Macro

1.Inline improves the function and is safe to use and therefore it is more useful than a macro.
2.The keyword which is used to define an inline function is “inline”. For the macro, the “#define” keyword is used to define it.
3.The class’s data member can be accessed through the inline function. While they cannot get accessed by the macros.
4.The program can be easily debugged in the inline functions while it’s not easy to debug in the case of macros.
5.Inline is almost never used in competitive programming while on the other hand, Macro is widely used in competitive programming.
6.The arguments in the case of inline are evaluated only once while in the case of macro the arguments are evaluated every time the macro function is used in the program.

C++ language
Lesson -7
Functions

Functions in C++

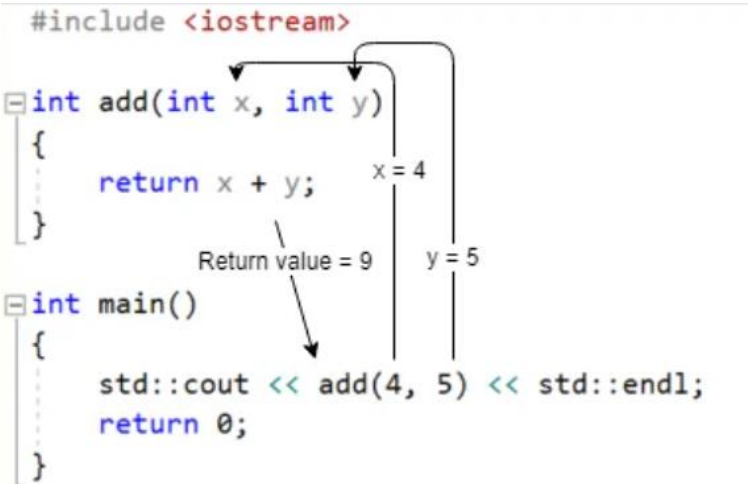
A function is a block of code that performs a specific task.
Dividing a complex problem into smaller chunks makes our program easy to understand and reusable.

C++ allows the programmer to define their own function.
A user-defined function groups code to perform a specific task and that group of code is given a name (identifier).
When the function is invoked from any part of the program, it all executes the codes defined in the body of the function.

Unlike some other programming languages, in C++, functions cannot be defined inside other functions.

A function parameter is a variable used in a function. Function parameters work almost identically to variables defined inside the function, but with one difference: they are always initialized with a value provided by the caller of the function.
Function parameters are defined in the function declaration by placing them in between the parenthesis after the function identifier, with multiple parameters being separated by commas.

An argument is a value that is passed from the caller to the function when a function call is made



x and y is function parameters
But
4 and 5 is function arguments

Functions Prototypes

The function prototype allows the C and C++ compilers to do ***much better type-checking*** on arguments used in function calls than was possible with the original C.

In C++, it is still legal, although ***unnecessary, to use 'void to specify an empty argument list.***

In C++ a return type must be specified; a missing return type does not default to int as is the case in C.

In C++, a function that has no parameters can have an empty parameter list.

```
int print (void); /* C style */
int print(); // C++ style
```

Function overloading

In C++, functions may be overloaded: there may be two or more functions with the **same name** but with differences between their parameters. Here is an example:

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

void sqr(int);
void sqr(int, int);

int main()
{
        int x,y;
        x=2;
        y=3;
        sqr(x);
        sqr(x,y);
}

void sqr(int x)
{
        cout<<x * x<<endl;
}

void sqr(int x,int y)
{
        cout<<x*x+y*y<<endl;
}
```

the compiler chooses whichever function definition is matched by a given call to the function 'sqr' checking first for an exact match and then for a match after type conversions.

If there is no match a compilation error results. if two function definitions are identical in name and argument list, a compilation error also results.

name mangling (also called name decoration) is a technique used to solve various problems caused by the need to resolve unique names for programming entities in many modern programming languages.

main function

A program shall contain a global function named main, which is the designated start of the program. It shall have one of the following forms:

```
int main ()
{

return 0;
}
```

or

```
int main (int argc, char *argv[])
{

return 0;
}
```

declarations and definitions

In C, declarations and definitions must occur at the beginning of a block.

In C++ declarations and definitions can be placed anywhere an executable statement can appear, except that they must appear prior to the point at which they are first used. This improve the readability of the program.

```
for (int i=0; i < 100; i++)
{ // i is declared in for loop
        ...
        ...
        ...
}
```

Default Arguments

C++ syntax includes, unlike C, the default argument as part of the function-calling sequence. A parameter in the called function may be assigned a default value if there is no matching argument in the function call.
Here is an example:

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

void myfunc(int = 3, int = 4);

int main (void)
{
int param1 = 5 ;
int param2 = 6 ;

myfunc() ;
myfunc(param1) ;
myfunc(param1 , param2) ;

}

void myfunc(int a,int b)
{
cout <<"Parameters Are: "<< a<<" "<< b << "\n" ;

}
```

The function 'myfunc' may be called with zero, one or two arguments, as it is from the 'main' function in the example. Here are the results of the function calls:

```
Parameters Are: 3 4
Parameters Are: 5 4
Parameters Are: 5 6
```

Note :

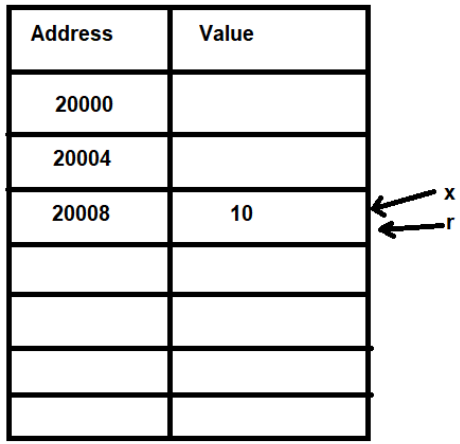
- The default values are specified in the prototype of 'myfunc'. It is legal, but unnecessary, to include argument names in the prototype:

void myfunc(int a = 3, int b = 4);

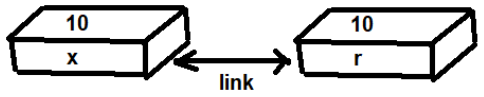
- The default values must be specified in the prototype and not in the function header, unless the function definition is also a declaration

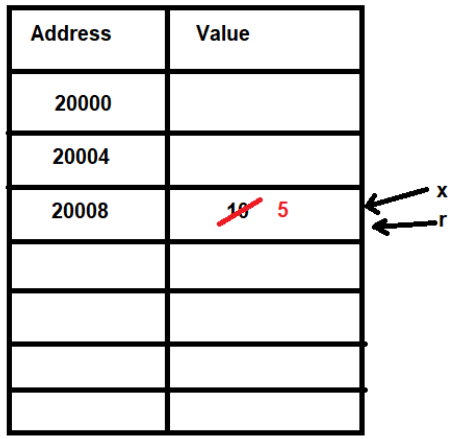
Reference

This allows a variable to be given an alternative name and to be accessed indirectly, without using pointers

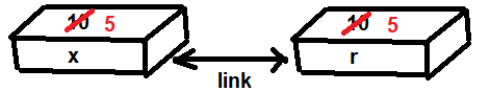


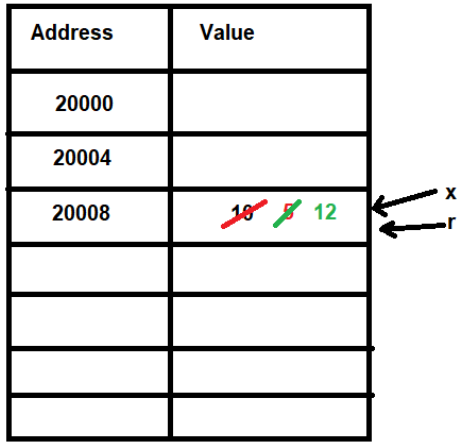

```
int  x=10;
int  &r=x;
```



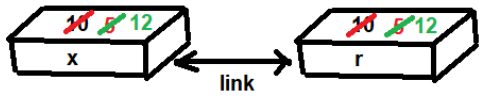


x=5 or r=5






```
int z=12;
r=z;
```



References

C++ introduces the reference to a declaration. This allows a variable to be given an alternative name and to be accessed indirectly, without using pointers, as is necessary in C.

Passing arguments to functions 'by reference' in C is in fact simulated using pointers; the C++ reference declaration also allows true function call-by-reference. The following program initializes a reference declaration.

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

int main ()

{
   int n1   = 7;
   int n2     = 8;
   int &ref_n = n1;       //ref_n is an alias  for n1

   ref_n *= 5;
   cout <<ref_n << "  " << n1 << "\n" ;

   ref_n = n2;     // n1 changed, not just ref_n
   cout << ref_n << " "<< n1 << "\n";
   return (0) ;
}
```

```
35   35
8 8
```

Once the reference ref_n is initialized to n1, it is simply an alias for n1. Ref_n cannot subsequently be made to be an alias to something else, as the example shows. After the assignment:

ref_n = n2;

the value of n1 is assigned that of n2.

The C++ compiler implements references to variables with standard C pointers and de-referencing, but this is transparent to the programmer.

C++ language
Lesson -8

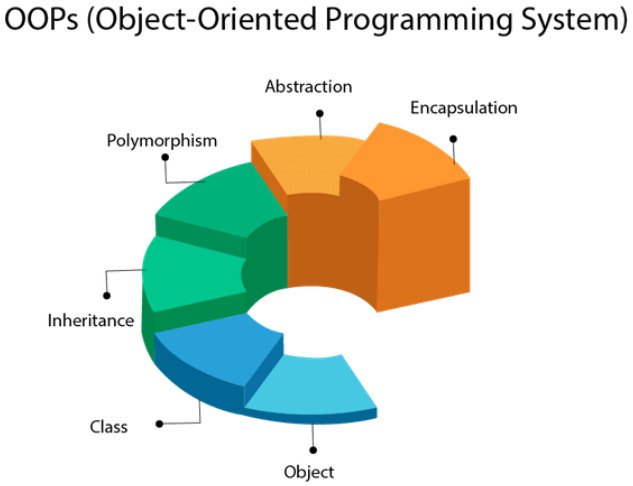
Introduction to OOP

Emergence of OOP

In the early day of computer, programmers wrote programs to solve simple mathematical , scientific , and computational problems .Procedural solutions provided flexibility in creation this type of software system .

Reasons led to Emergence of OOP:

1- rapid growth of the hardware industry was not matched by growth in the software industry . with the advent of better hardware , there was higher demand for software solution suited to a variety of problem domains.
2- the problem domain have changed. Instead of mathematical and scientific problems , today computer programs are used in cruise control system ,computer – aided manufacturing and flight simulators .
3- object- oriented technology uses real world metaphors to provide solutions to problems.
4- enforce Modularity for easier troubleshooting .
5- programs can implemented and organized as co-operative collection of entities , each representing an instance of some entity in the real world .



What is OOP introduce?

1-it simulates the activity of real world.

2- it has user-defined type .

3- it hides the implementation details .

4- it reuses code.

5- it allows the same name for different functionalities.

6- increase security -by preventing objects from obtaining references to other objects to which they should not have access.

What is OOP?

2- OOP is the **most popular programming paradigm** and is taught as the standard way to code for most of a programmers educational career.
3- Object Oriented programming (OOP) is a programming paradigm that relies on the concept of **classes** and **objects**.
4-It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects.
5- a **programming paradigm** is a way to build a program or a methodology to apply .
6- the OOP is a programming paradigm improve the structure programming so it is an enhancement of procedural programming .
7- most modern programming language support the OOP **paradigm** like (C++,java ,python,C#)
8- The following languages are not object oriented languages: C,FORTRAN,COBOL,PASCAL,BASIC

programming paradigms

Paradigm can also be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach.

Imperative programming paradigm:
It is one of the oldest programming paradigm. It features close relation to machine architecture. It is based on Von Neumann architecture. It performs step by step task The main focus is on ***how*** to achieve the goal.

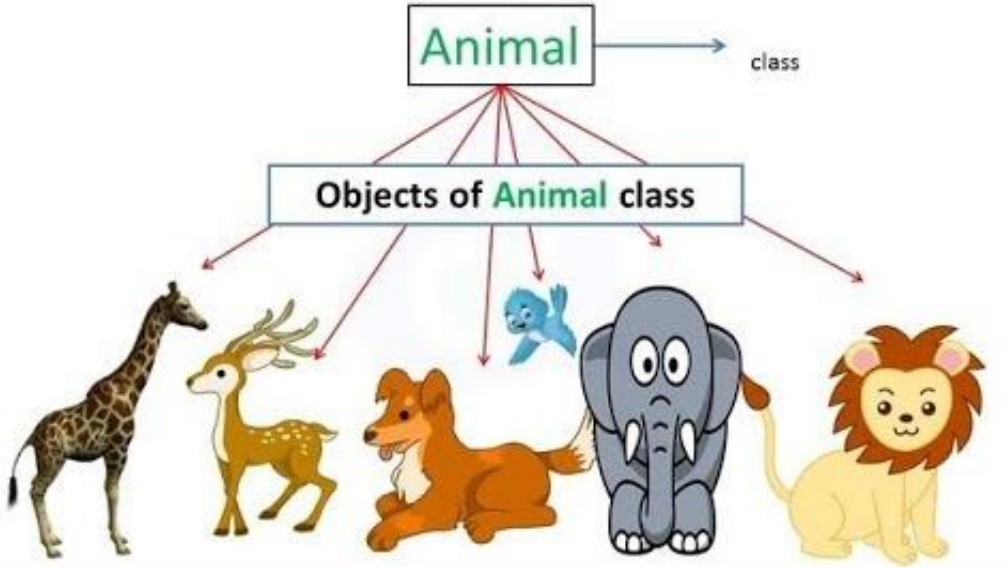
Declarative programming paradigm:
In computer science the *declarative programming* is a style of building programs that expresses logic of computation without talking about its control flow. The focus is on ***what*** needs to be done rather how it should be done basically emphasize on what code is actually doing.

Note:
the only difference between imperative ***(how to do)*** and declarative ***(what to do)*** programming paradigms.

Class and Objects?

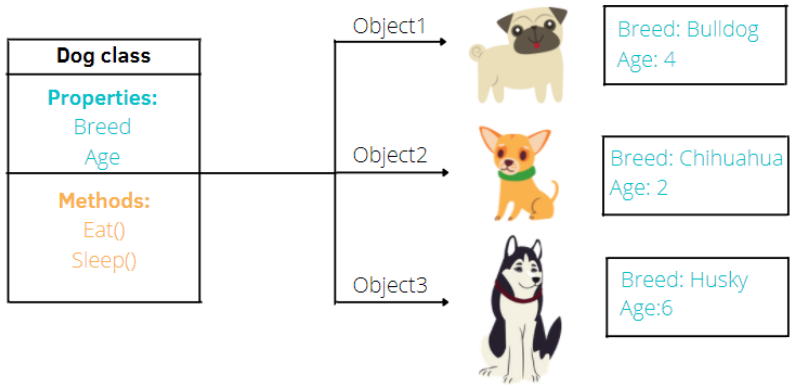
Class:

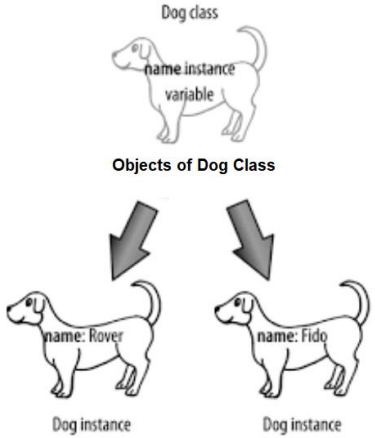
A class is the building block that leads to Object-Oriented Programming. It is a user-defined data type, that holds its own data members and member functions, which can be accessed and used by creating an instance of that class (or object). It is the blueprint of any object.



Objects:

An object is an instance of a class. All data members and member functions of the class can be accessed with the help of objects.
When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created), memory is allocated.
For example, considering the objects for the Student class is the First student, Second student ,etc.





Class vs Object

Class : *It is a user-defined data type, that holds its own data members and member functions, which can be accessed and used by creating an instance of that class (or object).*

Object : *An object is an instance of a class.*

Class Vs. Object

Class Vs. Object

Here is the important difference between class and object:

Class	Object
A class is a template for creating objects in program.	The object is an instance of a class.
A class is a logical entity	Object is a physical entity
A class does not allocate memory space when it is created.	Object allocates memory space whenever they are created.
You can declare class only once.	You can create more than one object using a class.
Example: Car.	Example: Jaguar, BMW, Tesla, etc.
Class generates objects	Objects provide life to the class.
Classes can't be manipulated as they are not available in memory.	They can be manipulated.
It doesn't have any values which are associated with the fields.	Each and every object has its own values, which are associated with the fields.
You can create class using "class" keyword.	You can create object using "new" keyword in Java

What is the Object

An object is a user-defined composite datatype that encapsulates a data along with the functions and procedures needed to manipulate the data (actions).

The data are called attributes, properties ,status, member variable .
The actions are called methods , behavior , actions, operations

Example :
Student Data:

National ID
University_ID
Name
Address
Gender
Birth date
GPA
faculty
Study_level

Example :
Student Action:

Register
Login
Logout
Start_exam
End_Exam
Update_GPA

OOP four pillars ?

Encapsulation:

refers to the bundling of data, along with the methods that operate on that data, into a single unit. Many programming languages use ***encapsulation*** frequently in the form of ***classes***. Encapsulation may also refer to a mechanism of restricting the direct access to some components of an object .

Abstraction:

Abstraction occurs when a programmer hides any irrelevant data about an object or an instantiated class to reduce complexity and help users interact with a program more efficiently and easily . This mechanism should ***hide internal implementation*** details. It should only reveal operations relevant for the other objects.

Inheritance:

Inheritance is that it provides code ***re-usability***. In place of writing the same code, again and again, we can simply inherit features of one class into the other .

Polymorphism:

It describes the concept that different classes can be used with the same interface. Each of these classes can provide its own implementation of the interface.

Benefits of OOP

Modularity:
Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.

Reusability:
Code can be reused through inheritance, meaning a team does not have to write the same code multiple times.

Productivity:
Programmers can construct new programs quicker through the use of multiple classes and reusable code.

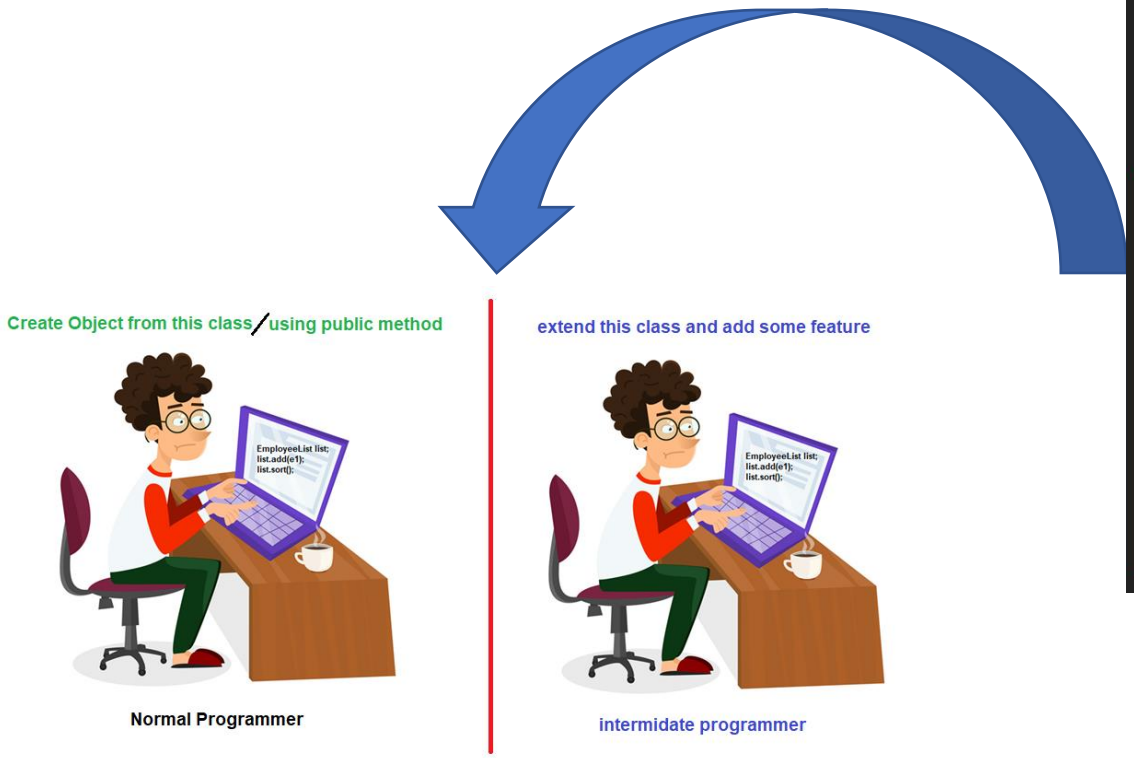
Easily upgradable and scalable:
Programmers can implement system functionalities independently.

Interface descriptions:
Descriptions of external systems are simple, due to message passing techniques that are used for objects communication.

Security:
Using encapsulation and abstraction, complex code is hidden, software maintenance is easier and internet protocols are protected.

Flexibility:
Polymorphism enables a single function to adapt to the class it is placed in. Different objects can also pass through the same interface.

OOP reduce complexity



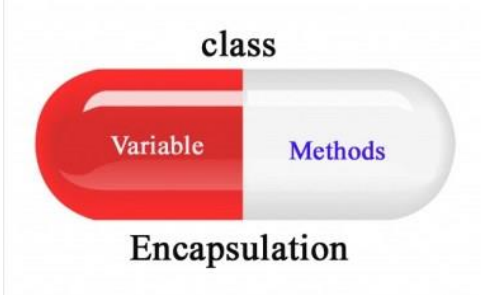
Abstraction occurs when a programmer hides any irrelevant data about an object or an instantiated class to reduce complexity and help another programmers interact with a object more efficiently and easily .

C++ language
Lesson -9

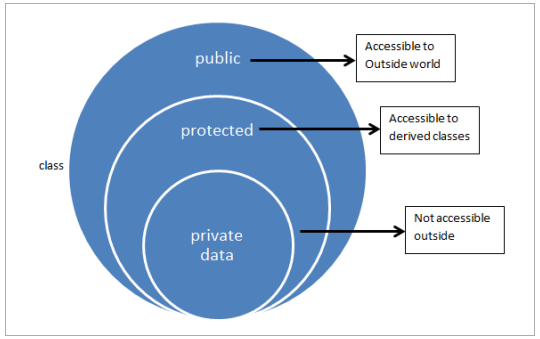
Encapsulation & Classes

Encapsulation

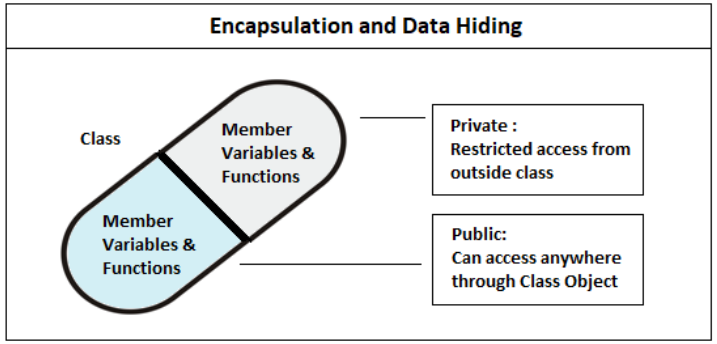
refers to the bundling of data, along with the methods that operate on that data, into a single unit.
Many programming languages
use encapsulation frequently in the form of ***classes***.



Encapsulation may also refer to a mechanism of ***restricting*** the direct access to some components of an object .



Data hiding is a software development technique specifically used in (OOP) to hide internal object details (data members).
Data hiding ensures exclusive data access to class members and ***protects*** object integrity by ***preventing*** unintended or intended changes.



Classes

C++ introduces classes to provide language support for the object-oriented programming approach. The class is a generalization of the structure ('struct') construct found in C. The class, along with inheritance and virtual functions, is the central innovation made by C++.

Like a structure, a class consists of a number of members. Unlike the structure, the members can be functions. A class is used to describe a ***real- world object***.

Consider the case of a clock. A clock records information including (at least) hours and minutes.

At least two operations are possible on this information: the minutes may be advanced by 1 and the hours may be advanced by 1.

In c++ the clock's information and possible operations might be recorded like this:

```
class clock
{
private:
      int seconds;
      int minutes;
      int hours;
public:
      Void adv_seconds();
      void adv _minute( );
      void adv_hour();
};
```

Class object

This is a class declaration; to define an instance of the class - often also referred to as a 'class object' of 'class variable' - is similar to definition of a 'struct' instance in C:

class clock wall_clock;
or
clock wall_clock;

The second definition takes account of the fact that, in C++, the tag name, 'clock', is itself a type.

access specifiers in c++

In C++, there are three access specifiers:

public :
members are accessible from outside the class.

Private:
members cannot be accessed (or viewed) from outside the class.

Protected:
members cannot be accessed from outside the class, however, they can be accessed in ***inherited*** classes.

Access Modifiers

	Own class	Derived class	Main()
Private	✓	✗	✗
Protected	✓	✓	✗
public	✓	✓	

There are two parts to the class 'clock', ***private*** and ***public***.

private	**public**
The 'private' keyword means that the class members declared following it are only accessible to member functions of the class 'clock' - 'adv_minute' and 'adv_hour'.	The 'public' keyword means that any other class or function may make a call to either of the functions 'adv_minute' or 'adv_hour'.

Object is an instance of a Class

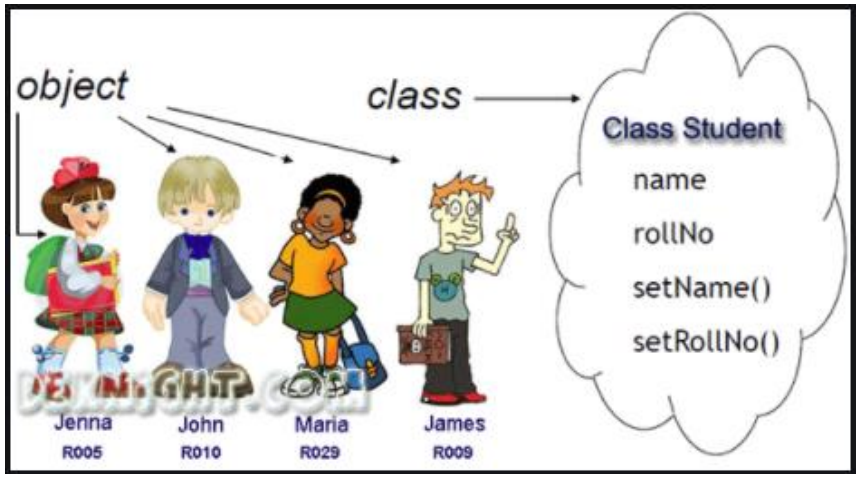
An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Before making such a call for any method in class , an instance of the class must be defined as above.

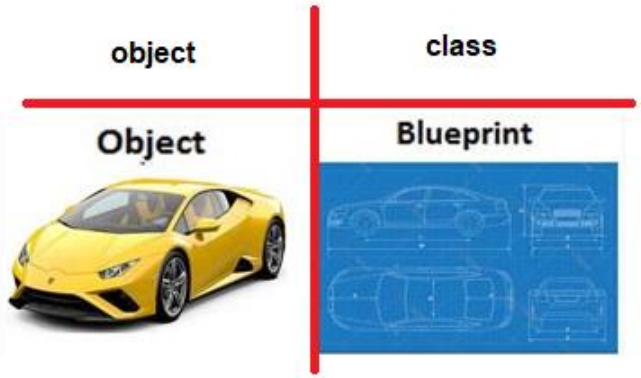
clock wall_clock;

Then the call the method like this:

wall_clock. adv_minute();



Definition : Object is a real world entity, for example, chair, car, mobile …etc. object is an entity that has state and behavior. Here, state means data and behavior means functionality.



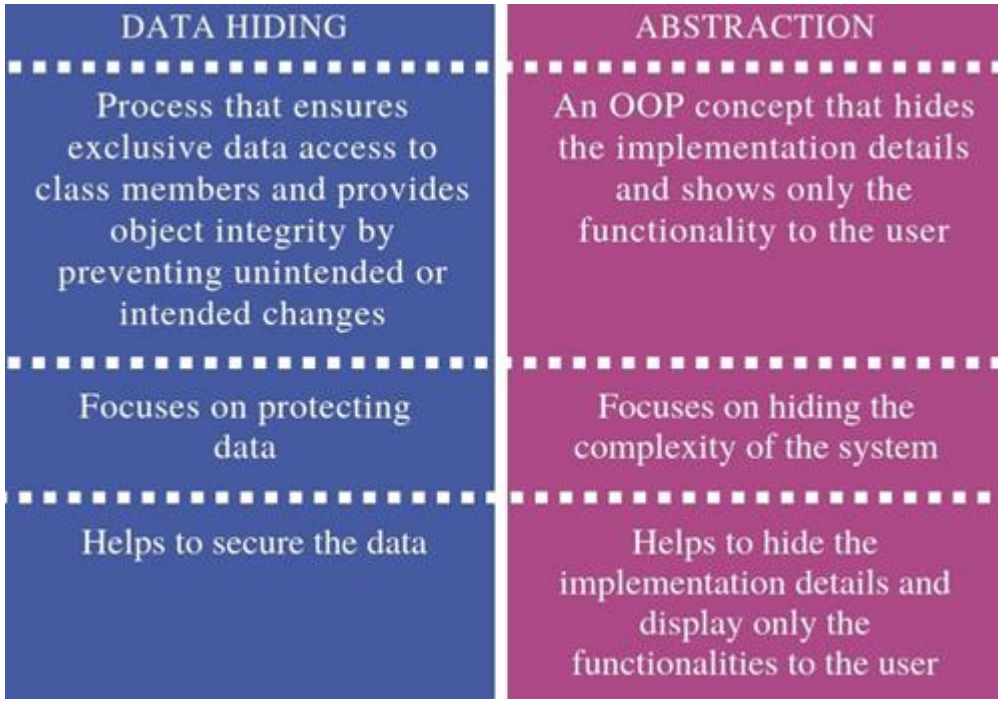
Definition: A class is a blueprint that defines the variables and the methods common to all objects of a certain kind.

data hiding

The data hiding which is enforced by the private part of the class means that the private members cannot be directly used by code other than that defined in the member functions of the same class.

All that is ***available to outside*** code is the class's function call ***interface***; the internal implementation of the class remains a 'black box'.

This mechanism results in the production of highly modular code which may be ***re-used*** by many programmers, without their having to know anything about the code other than how to call it.



Access Private Members

The general class 'clock' as declared above To access a private attribute, use public "get" and "set" methods:

```
class clock
{
private:
        int minutes;
        int hours;
public:
        void adv _minute( );
        void adv_hour();
        set_ minute(int m);
        get_minute();
        set_hour(int h);
        get_hour();
};
```

```
class clock
{
private:
        int minutes;
        int hours;
public:
        void adv _minute( );
        void adv_hour();
        set_ minute(int m);
        {
                minutes=m;
        };
        int get_minute()
        {
                return(minutes);
        }
        ;
        set_hour(int h)
        {
                hours=h;
        };
        int get_hour()
        {
                return(hours);
        };
};
```

Class Methods

The Methods are functions that belongs to the class.
There are two ways to define functions that belongs to a class:

- Inside class definition

```
class clock
{
private:
        int minutes;
        int hours;
public:
        void adv_minute( )
        {
                minutes++;
        };
        void adv_hour();
};
```

- Outside class definition

```
class clock
{
private:
        int minutes;
        int hours;
public:
        void adv_minute( );
        void adv_hour();
};

void  clock :: adv_hour()
{
                hours++;
}
```

C++ language
Lesson -10

Constructor & Destructor

Constructor

if the programmer forgets the initialization of the instance of class (object) every subsequent use of the class instance is an error.

A constructor is a member function of a class which normally 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.

A constructor is different from normal method in following ways:

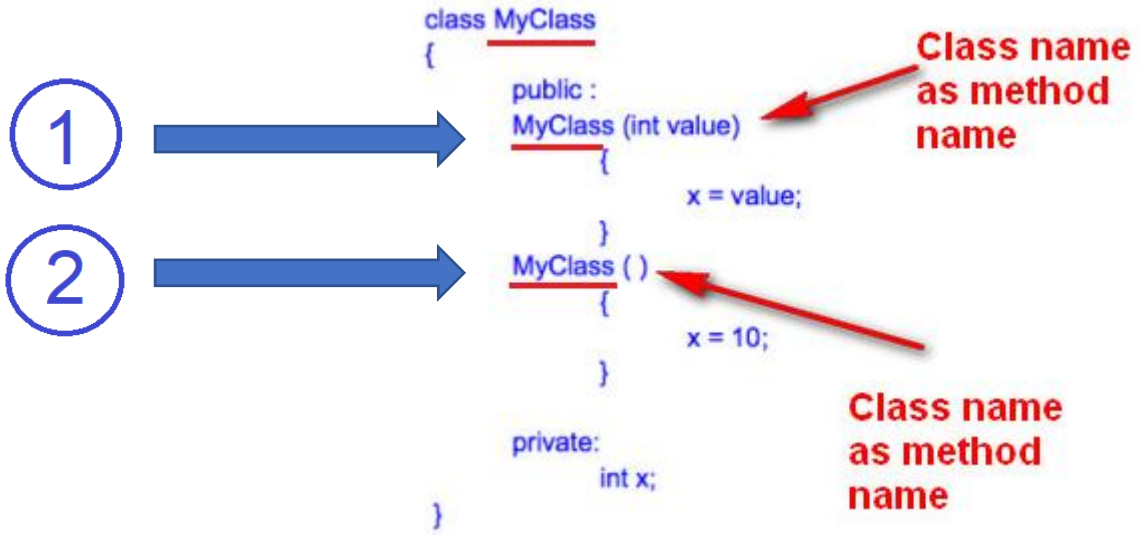
- Constructor has same name as the class itself.
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

```
#include<iostream>
using namespace std;
class Point
{
  public:
      int x;
      int y;

      Point()
        {  x=0;
           y=0;
        }

   int getX() { return (x);}
   int getY() { return (y);}
};
int main()
{
      // invoking a static member function
      Point p1;
      cout <<" p1 : x= "<< p1.getX()<<" y="<<p1.getY()<<endl;


}
```

Constructor for Class is automatically called when object(instance of class) create.

Constructor Overloading

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading and is quite similar to function overloading.

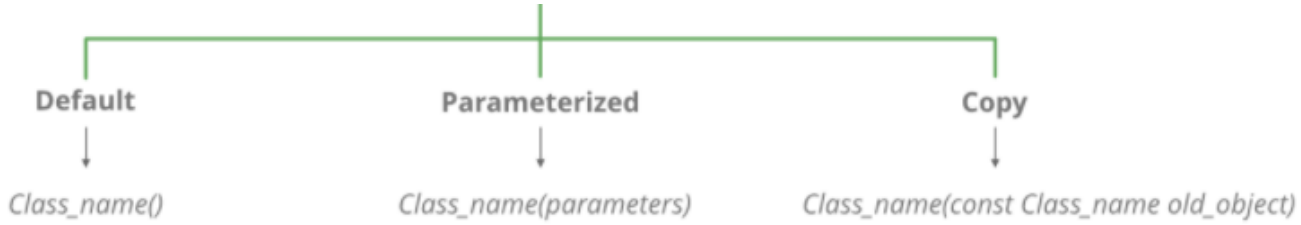


Constructor Example

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

class Point {
private:
   int x, y;  // Private data members
public:
    int getX()  { return (x); };
    void setX(int ix) { x=ix;};
    int getY() { return (y); };
    void setY(int y) { this->y=y; };

    Point() { x=0; y=0; };
    Point( int ix,int iy) { x=ix; y=iy; };
    Point(const Point& tp) { x=tp.x; y=tp.y; };
};

int main()
{
    Point p1,p2(200,300);
    cout <<p1.getX()<<" : "<<p1.getY()<<endl;
    p1.setX(44);  p1.setY(400);
    cout <<p1.getX()<<" : "<<p1.getY()<<endl;
    cout <<p2.getX()<<" : "<<p2.getY()<<endl;
    Point p3(p1);
    cout <<p3.getX()<<" : "<<p3.getY()<<endl;
}
```

```
0 : 0
44 : 400
200 : 300
44 : 400
```

Types of Constructors:



Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a ***default constructor*** implicitly.

*A **copy constructor** is a member function which initializes an object using another object of the same class.*

You can use default parameter :

Point(int i = 0, int j = 0) { x = i; y = j; };

copy constructor

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

The copy constructor is used to:

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

The most common form of copy constructor is shown here

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

If a copy constructor is not defined in a class, the compiler itself defines one.
(Default copy constructor)

Default copy constructor

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

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

class Point {
private:
  int x, y;  // Private data members
public:
   int getX()  { return (x); };
   void setX(int ix) { x=ix;};
   int getY() { return (y); };
   void setY(int y) { this->y=y; };

   Point() { x=0; y=0; };
   Point( int ix,int iy) { x=ix; y=iy; };
   Point(const Point& tp) { x=tp.x; y=tp.y; };
};

int main()
{
   Point p1,p2(200,300);
   cout <<p1.getX()<<" : "<<p1.getY()<<endl;
   p1.setX(44);  p1.setY(400);
   cout <<p1.getX()<<" : "<<p1.getY()<<endl;
   cout <<p2.getX()<<" : "<<p2.getY()<<endl;
   Point p3(p1);
   cout <<p3.getX()<<" : "<<p3.getY()<<endl;
}
```

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

class Point {
private:
  int x, y;  // Private data members
public:
   int getX()  { return (x); };
   void setX(int ix) { x=ix;};
   int getY() { return (y); };
   void setY(int y) { this->y=y; };

   Point() { x=0; y=0; };
   Point( int ix,int iy) { x=ix; y=iy; };
   Point(const Point& tp) { x=tp.x; y=tp.y; };
};

int main()
{
   Point p1,p2(200,300);
   cout <<p1.getX()<<" : "<<p1.getY()<<endl;
   p1.setX(44);  p1.setY(400);
   cout <<p1.getX()<<" : "<<p1.getY()<<endl;
   cout <<p2.getX()<<" : "<<p2.getY()<<endl;
   Point p3(p1);
   cout <<p3.getX()<<" : "<<p3.getY()<<endl;
}
```

When is the copy constructor called?

In C++, a Copy Constructor may be called in the following cases:

- When an object of the class is returned by value.
- When an object of the class is passed (to a function) by value as an argument.
- When an object is constructed based on another object of the same class.
- When the compiler generates a temporary object.

It is, however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example is the **return value optimization** (sometimes referred to as RVO).

no need for this line since

If a copy constructor is not defined in a class, the compiler itself defines one.

Form 1:
Point p2(p1);

Form2:
Point p2=p1;

copy constructor

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

When is the copy constructor called?

In C++, a Copy Constructor may be called in the following cases:

- When an object of the class is returned by value.
- When an object of the class is passed (to a function) by value as an argument.
- When an object is constructed based on another object of the same class.
- When the compiler generates a temporary object.

It is, however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example is the **return value optimization** (sometimes referred to as RVO).

Constructors and initializer lists

Initializer List is used in initializing the data members of a class. The list of members to be initialized is indicated with constructor as a comma-separated list followed by a colon.

```
#include<iostream>
using namespace std;
class Point
{
  public:
      int x;
      int y;

      Point()
         {
            x=0;
            y=0;
         }
       Point(int tx,int ty)
         {
            x=tx;
            y=ty;
         }

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

      Point p1;
      cout <<" p1 : x= "<< p1.getX()<<" y="<<p1.getY()<<endl;
      Point p2 (20,30);
      cout <<" p2 : x= "<< p2.getX()<<" y="<<p2.getY()<<endl;


}
```

```
#include<iostream>
using namespace std;
class Point
{
  public:
      int x;
      int y;

      Point() :x(0),y(0)
         {
         }
       Point(int tx,int ty) : x(tx),y(ty)
         {
         }

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

      Point p1;
      cout <<" p1 : x= "<< p1.getX()<<" y="<<p1.getY()<<endl;
      Point p2 (20,30);
      cout <<" p2 : x= "<< p2.getX()<<" y="<<p2.getY()<<endl;


}
```

Destructor

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

Destructor is a member function which call when destructs or deletes an object.

Properties of Destructor:

- Destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of destructor.

A destructor function is called automatically when the ***object goes out of scope***:
(1) the function ends
(2) the program ends
(3) a block containing local variables ends
(4) a delete operator is called

Destructor Example

```
~Point()
        {
           cout << "Destructor"<<endl;
        }
```

```
0 : 0
44 : 400
200 : 300
44 : 400
Destructor
Destructor
Destructor
```

```
#include <iostream>
using namespace std;
class Point {
private:
  int x, y;  // Private data members
public:
   Point()
   {
      cout << " Constructor "<<endl;
   };
   ~Point()
   {
      cout << " Destructor"<<endl;
   }
   //Point(int i = 0, int j = 0) { x = i; y = j; };
};

int main()
{
   // step 1
   Point p1;
   //step 2
   Point *p2;
   p2= new Point();
   // step 3
   delete p2;

}
```

Destructor Example

```cpp
#include <iostream>
using namespace std;
class Employee {
private:
  string * p_name;
  int * p_age;
  float * p_salary;
public:
   Employee(string name ,int age , float salary)
   {
      cout << " Constructor "<<endl;
       p_name = new string ;
      *p_name = name;
      p_age = new int;
      *p_age = age ;
      p_salary = new float;
      *p_salary = salary;
   };
   ~Employee()
   {
      delete p_name;
      delete p_age;
      delete p_salary;
      cout << " Destructor Deallocate memory "<<endl;
   }
      print()
      {
         cout << "name = "<<*p_name<<" , " <<"age = "<<*p_age<<" , "<< "salary = "<<*p_salary<<endl;

      }
};
```

```cpp
int main()
{
      // step 1
      Employee * emp1 = new Employee("youssef",59,20000);
      emp1->print();
      //step 2
      delete emp1;

}
```

```
 Constructor
name = youssef , age = 59 , salary = 20000
 Destructor
 Deallocate memory

Process returned 0 (0x0)   execution time : 0.067 s
Press any key to continue.
```

Shallow Copy and Deep Copy

Depending upon the resources like dynamic memory held by the object, either we need to perform Shallow Copy or Deep Copy in order to create a replica of the object. In general, if the variables of an object have been dynamically allocated then it is required to do a Deep Copy in order to create a copy of the object.

Shallow Copy:

In shallow copy, an object is created by simply copying the data of all variables of the original object. This works well if none of the variables of the object are defined in the heap section memory.

If some variables are dynamically allocated memory from heap section, then copied object variable will also reference then same memory location.

This will create ambiguity and run-time errors dangling pointer. Since both objects will reference to the same memory location, then change made by one will reflect those change in another object as well.

Deep Copy:

In Deep copy, an object is created by copying data of all variables and it also allocates similar memory resources with the same value to the object.

In order to perform Deep copy, we need to explicitly define the copy constructor and assign dynamic memory as well if required.

Also, it is required to dynamically allocate memory to the variables in the other constructors, as well.

Shallow Copy

Shallow Copy:

In shallow copy, an object is created by simply copying the data of all variables of the original object. This works well if none of the variables of the object are defined in the stack section memory.

```cpp
#include <iostream>
using namespace std;
class Point
{
    int x;
    int y;
public:
    Point()
    {
        x=0;
        y=0;
    }
    Point(int x,int y)
    {
        this->x=x;
        this->y=y;
    }
    print()
    {
        cout << " point =("<<x<<","<<y<<")"<<endl;
    }
    Point(const Point& p)
    {
        this->x= p.x;
        this->y=p.y;
    }
    setxy(int nx,int ny)
    {
        x=nx;
        y=ny;
    }
};
int main()
{
    Point p1(4,5);
    Point p2(p1);

    p1.print();
    p2.print();
    p2.setxy(10,20);
    p1.print();
    p2.print();

}
```

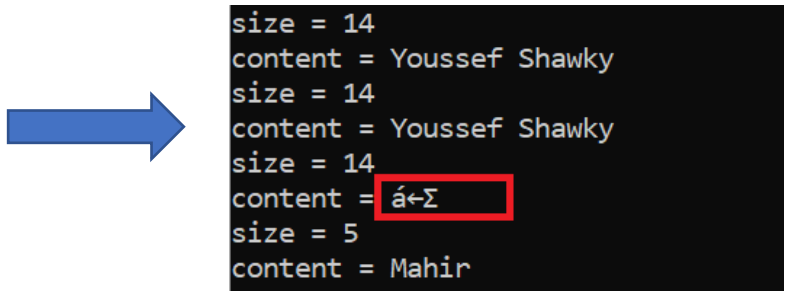
```
point =(4,5)
point =(4,5)
point =(4,5)
point =(10,20)
```

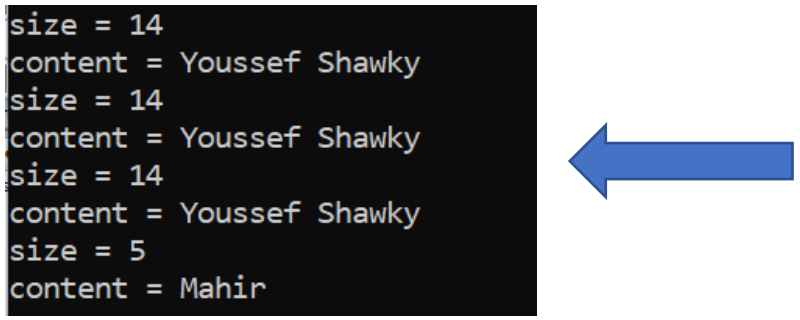
Deep Copy and wrong shallow copy

```
/* Without implement Copy Constructor */
#include <cstring>
#include <iostream>
using namespace std;
class String {
private:
   char* s;
   int size;
public:
   String(const char* str = NULL); // constructor
   ~String() { delete[] s; } // destructor
    void print()
   {  cout<< "size = " <<size<<endl;
      cout << "content = "<<s << endl;
   }
   void change(const char*); // Function to change
};

String::String(const char* str)
{
   size = strlen(str);
   s = new char[size + 1];
   strcpy(s, str);
}
void String::change(const char* str)
{
   delete[] s;
   size = strlen(str);
   s = new char[size + 1];
   strcpy(s, str);
}
int main()
{
   String str1("Youssef Shawky");
   String str2 = str1;
   str1.print(); // what is printed ?
   str2.print();
   str2.change("Mahir");
   str1.print(); // what is printed now ?
   str2.print();
   return 0;
}
```

Deep Copy:

In Deep copy, an object is created by copying data of all variables and it also allocates similar memory resources with the same value to the object.






```
/* With implement Copy Constructor */
#include <cstring>
#include <iostream>
using namespace std;
class String {
private:
   char* s;
   int size;
public:
   String(const char* str = NULL); // constructor
   ~String() { delete[] s; } // destructor
    void print()
   {  cout<< "size = " <<size<<endl;
      cout << "content = "<<s << endl;
   }
   void change(const char*); // Function to change
   String(const String& old_str);
};
String::String(const String& old_str)
{
   size = old_str.size;
   s = new char[size + 1];
   strcpy(s, old_str.s);
}
String::String(const char* str)
{
   size = strlen(str);
   s = new char[size + 1];
   strcpy(s, str);
}
void String::change(const char* str)
{
   delete[] s;
   size = strlen(str);
   s = new char[size + 1];
   strcpy(s, str);
}
int main()
{
   String str1("Youssef Shawky");
   String str2 = str1;
   str1.print(); // what is printed ?
   str2.print();
   str2.change("Mahir");
   str1.print(); // what is printed now ?
   str2.print();
   return 0;
}
```

C++ language
Lesson -11

Friends

Friends

Friend Function

A ***friend function*** of a class is defined outside that class' scope but it has the right to access all ***private*** and ***protected*** members of the class.

Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows

A friend function can be:

a) A method of another class

b) A global function

Friend class

A ***friend class*** is a class that can access the ***private*** and ***protected*** members of a class in which it is declared as **friend**.

This is needed when we want to allow a particular class to access the private and protected members of a class.

Friend Function and class

```cpp
#include <iostream>
using namespace std;
class MyClass {
    private:
      int num;
      char ch;
    public:
       MyClass()
       {
           num=100;
           ch='A';
       }
       friend void display(MyClass obj);
};

//Global Function
void display(MyClass obj){
   cout<<obj.num<<endl;
   cout<<obj.ch<<endl;
}

int main() {
   MyClass obj;
   display(obj);
   return 0;
}
```

```cpp
#include <iostream>
using namespace std;
class Class_A
{
    private:
       int A_value;
    public:
       Class_A() // Default constructor
       {
           A_value = 10;
       }
    friend class Class_B; // Friend Class
};
class Class_B
{
    private:
       int B_value;
    public:
    /*Accessing private members of the Class_A class using friend class Class_B */
    void display(Class_A x)
    {
       cout<<"The Class_A private value accessed using friend class is: " << x.A_value<<endl;
    }
};
int main()
{
    Class_A obj_a;
    Class_B obj_b;
    obj_b.display(obj_a);
    return 0;
}
```

Friend Function

one common and good use of a friend function occurs when ***two different types of classes*** have some quantity in common that ***needs to be compared***.

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

// class truck;  forward reference
class car
{
      int speed;
      int passengers;
   public:
      car (int s, int p) { speed = s; passengers=p;}
      friend int sp_greater( car c , truck t);
};
class truck
{
   int speed;
   int weight;
   public:
      truck( int s, int w) { speed = s; weight = w;}
      friend int sp_greater(car c , truck t);
};

int sp_greater(car c, truck t)
{
   return c.speed-t.speed;
}
```

```
int main()
{
   int t;
   car c1(180,4);
   truck t1(140,400);

   t = sp_greater(c1,t1);
   if (t < 0)  cout << " truck faster \n";
   else if (t==0) cout<< " equal speed\n";
         else cout<< " car faster\n";
}
```

In C++, ***Forward declarations are usually used for Classes***. In this, the class is pre-defined before its use so that it can be called and used by other classes that are defined before this.

Friend Function

a function may be a member of one class (car) and a friend of another (truck)

```
1  #include <iostream>
2  using namespace std;
3
4  class truck;  //forward reference
5  class car
6  {
7        int speed;
8        int passengers;
9     public:
10       car (int s, int p) { speed = s; passengers=p;}
11       int sp_greater(truck t)
12       {
13         return speed-t.speed;
14       };
15 };
16 class truck
17 {
18    int speed;
19    int weight;
20    public:
21       truck( int s, int w) { speed = s; weight = w;}
22       friend int car::sp_greater(truck t);
23
24 };
25 int main()
26 {
27    int t;
28    car c1(180,4);
29    truck t1(140,400);
30
31    t = c1.sp_greater(t1);
32    if (t < 0)  cout << " truck faster \n";
33    else if (t==0) cout<< " equal speed\n";
34          else cout<< " car faster\n";
35 }
```

The definition of member functions can be inside or outside the definition of class. If the member function is defined inside the class definition it can be defined directly, but if its defined outside the class, then we have to use the scope resolution :: operator along with class name :: with function name.

Think and correct the error !!!!! .

C++ language
Lesson -12

static

Static data members

In C++, static is a keyword or modifier that belongs to the type ***not instance***. So instance is not required to access the static members. In C++, static can be ***field***, ***method***, ***constructor***, ***class***.

Static data members are class members that are declared using the static keyword. There is only ***one copy*** of the static data member in the class, even if there are many class objects. This is because all the objects share the static data member.

The syntax of the static data members is given as follows :

```
static data_type data_member_name;
```

Note :

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

Static variables in a Function

When a variable is declared as static, space for it gets allocated for the lifetime of the program. Even if the function is called multiple times, space for the static variable is allocated only once and the value of variable in the previous call gets carried through the next function call.

```
#include <iostream>
using namespace std;
void demo()
{
    // static variable
    static int count = 0;
    cout << count << " ";
    count++;
}

int main()
{
    for (int i=0; i<5; i++)
        demo();
    return 0;
}
```

```
0 1 2 3 4
```

You can see in the above program that the variable count is declared as static. So, its value is carried through the function calls. The variable count is not getting initialized for every time the function is called.

Static data Example

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

using namespace std;
class Student {
   private:
        int code;
        char name[10];
        int marks;
   public:
        static int objectCount;
       Student() {
          objectCount++;
       }
};

int Student::objectCount = 0;

int main(void) {

   Student s1;
   Student s2;
   Student s3;

   cout << "Total objects created = " << Student::objectCount << endl;
   return 0;
}
```

Static variables in a class

As the variables declared as static are initialized only once as they are allocated space in separate static storage so, the static variables in a class are shared by the objects. There can not be multiple copies of same static variables for different objects. Also because of this reason ***static variables can not be initialized using constructors.***

```
#include<iostream>
using namespace std;
class GfG
{
   public:
     static int i;
};
// this program generate  error
int main()
{
  GfG obj1;
  GfG obj2;
  obj1.i =2;
  obj2.i = 3;
  // prints value of i
  cout << obj1.i<<" "<<obj2.i;
}
```

Error

You can see in the above program that we have tried to create multiple copies of the static variable i for multiple objects. But this didn't happen. So, a static variable inside a class should be initialized explicitly by the user using the class name and scope

Static variables in a class

These functions work for the class as whole rather than for a particular object of a class.
It can be called using an object and the direct member access . operator.
But, its more typical to call a static member function by itself, using class name and scope resolution :: operator.

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

class GfG
{
public:
   static int i;
};

int GfG::i = 1;

int main()
{
   GfG obj1;

   // prints value of i
   cout << obj1.i<<endl;
   GfG::i=5;
   GfG obj2;
   cout << obj2.i<<endl;
   cout << obj1.i<<endl;

}
```

Static Member Functions

These functions work for the class as whole rather than for a particular object of a class.
It can be called using an **object** and the **direct member** access . operator.
But, its more typical to call a static member function by itself, using class name and scope resolution :: operator.

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

class MyClass
{
   public:
   static void s_function()
   {
      cout << "static"<<endl;
   }
};

int main()
{
   MyClass::s_function();   // calling member function directly with class name
   MyClass obj;
   obj.s_function();        // calling member function by instance variable
}
```

Note :
These functions **cannot** access ordinary data members and member functions, but only static data members and static member functions.
It doesn't have any "this" keyword which is the reason it cannot access ordinary members.

C++ language
Lesson -13

Operator Overloading

Operator Overloading

C++ provides two kinds of overloading: ***function overloading*** and ***operator overloading***.

Function overloading allows more than one version of a function with the same name to be used, with the appropriate version being called according to the parameter types used by the function.

Operator overloading allows a standard C++ operator to take on a new meaning.

Operator overloading is a compile-time polymorphism in which the **operator** is **overloaded** to provide the special meaning to the user-defined data type.

Operator overloading is used to **overload** or redefines most of the **operators** available in C++. It is used to perform the operation on the user-defined data type.

Operator Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Rules for Operator Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type.

Rules for Operator Overloading:

- ***Existing operators*** can only be overloaded.

- The overloaded operator ***contains at least one*** operand of the ***user-defined data type***.

- We ***cannot*** use ***friend function*** to overload ***certain operators***. However, the ***member function can be*** used to overload those operators (Assignment operator =,function call operator () subscriping operator [],class member access operator ->).

- When ***unary operators*** are overloaded through a ***member function*** take ***no explicit arguments***, but, if they are overloaded by a ***friend function***, takes ***one argument***.

- When ***binary operators*** are overloaded through a ***member function*** takes ***one explicit argument***, and if they are overloaded through a ***friend function*** takes ***two explicit arguments***.

Syntax for C++ Operator Overloading

To overload an operator, we use a special operator function.

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

Here:

- returnType is the return type of the function.
- operator is a keyword.
- symbol is the operator we want to overload. Like: +, <, -, ++, etc.
- arguments is the arguments passed to the function.

Binary Operator Overloading

When binary operators are overloaded through a ***member function*** takes one explicit argument, and if they are overloaded through a ***friend function*** takes two explicit arguments.

```
#include <iostream>
using namespace std;
class Point {
   int x;
   int y;
public:
   void set(int x1,int y1) { x=x1 ;y=y1 ; }
   void set(const Point &p) {x=p.x; y=p.y;}
   void show() {cout<< "\n x=" <<x << " y=" <<y<< "\n";}
   Point(int x1,int y1) { x=x1 ;y=y1 ; }
   Point() { x=0 ;y=0 ; }
   Point operator+(Point p)
   {
      Point temp;

      temp.x=x+p.x;
      temp.y=y+p.y;
      return temp;
   };
};

int main()
{
   Point p1(10,20);
   Point p2(15,30);
   Point p3;

   p3 = p1+p2;
   p3.show();
   return (0);
}
```

Using member function

Using friend function

```
#include <iostream>
using namespace std;
class Point {
   int x;
   int y;
public:
   void set(int x1,int y1) { x=x1 ;y=y1 ; }
   void set(const Point &p) {x=p.x; y=p.y;}
   void show() {cout<< "\n x=" <<x << " y=" <<y<< "\n";}
   Point(int x1,int y1) { x=x1 ;y=y1 ; }
   Point() { x=0 ;y=0 ; }
   friend Point operator+(Point p1,Point p2);

};

Point operator+(Point p1,Point p2)
   {
      Point temp;

      temp.x=p1.x+p2.x;
      temp.y=p1.y+p2.y;
      return temp;
   };

int main()
{
   Point p1(10,20);
   Point p2(15,30);
   Point p3;

   p3 = p1+p2;
   p3.show();
   return (0);
}
```

Unary Operator Overloading

unary operators operate on a single operand like (The increment (++) and decrement (--) operators, The unary minus (-) operator ,The logical not (!) operator)

The unary operators operate on the object for which they were called and normally, this operator appears on the ***left side*** of the object (as **prefix operator)** , as in !obj, -obj, and ++obj

```
#include <iostream>
using namespace std;
class Point {
   int x;
   int y;
public:
   void set(int x1,int y1) { x=x1 ;y=y1 ; }
   void set(const Point &p) {x=p.x; y=p.y;}
   void show() {cout<< "\n x=" <<x << " y=" <<y<< "\n";}
   Point(int x1,int y1) { x=x1 ;y=y1 ; }
   Point() { x=0 ;y=0 ; }
   Point operator-()
   {
      Point temp;

      temp.x=-x;
      temp.y=-y;
      return temp;
   };
   Point operator ++()
   {
      Point temp;

      x++;
      y++;
      return *this;
   }
};

int main()
{
   Point p1(10,20);
   Point p2,p3;

   p2=++p1;
   p2.show();

   p3=-p1;
   p3.show();
   return (0);
};
```

postfix Operator Overloading

The operator symbol for both prefix(++i) and postfix(i++) are the same.

Hence, we need two different function definitions to distinguish between them. This is achieved by passing a ***dummy int parameter*** in the postfix version.

```
#include <iostream>
using namespace std;
class Point
{
   int x;
   int y;
public:
   void set(int x1,int y1) {x=x1 ;y=y1;}
   void set(const Point &p) {x=p.x; y=p.y;}
   void show() {cout<< "\n x="<< x<< " y="<< y<< "\n";}
   Point operator=(Point ob2);
   int operator==(Point ob2);
   Point operator++();          // prifix
   Point operator++(int z);     //postfix
};
```

```
Point Point::operator=(Point ob2)
{
   this->x=ob2.x+5;
   this->y=ob2.y+5;
   return *this;
}

int Point::operator==(Point ob2)
{
   if((x==ob2.x)&&(y==ob2.y))
      return 1;
   else
      return 0;
}

Point Point::operator++()
{
   x++;
   y++;
   return *this;
}

Point Point::operator++(int z)
{
   x+=2;
   y+=2;
   return *this;
}
```

```
int  main()
{
   Point p1,p2;
   Point p3;


   p1.set(10, 20);
   p2.set(20,30);

   p1++;
   ++p2;
   p1.show();
   p2.show();
   if (p1==p2)
      cout << " Equal points "<<endl;
   else
      cout << " not equal points "<<endl;
   p3=p1;
   p3.show();
}
```

C++ language
Lesson -14

Inheritance

Inheritance

Inheritance is that it provides code ***re-usability***. In place of writing the same code, again and again, we can simply inherit features of one class into the other .

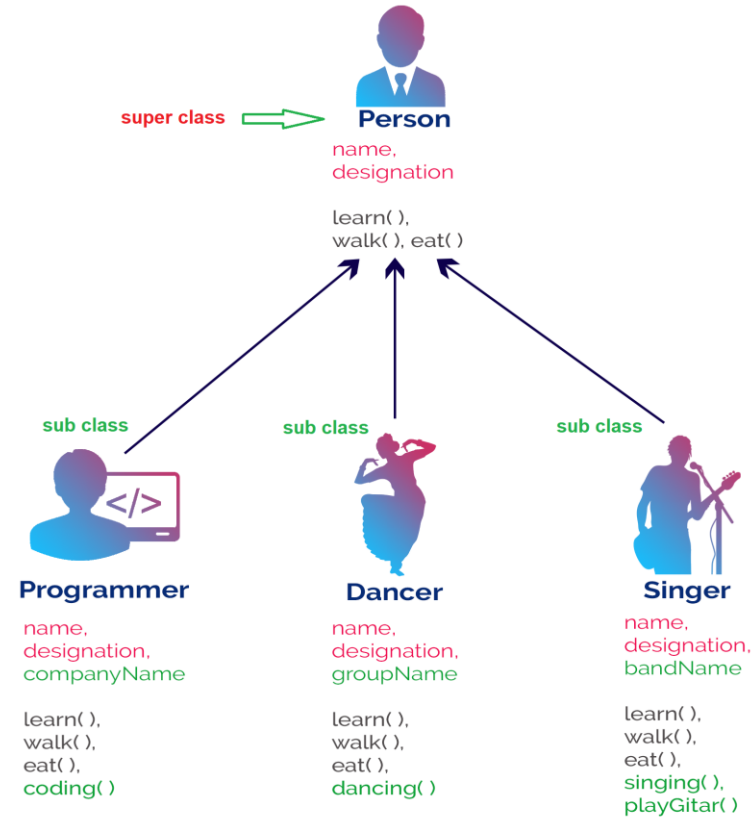
The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class:

The class that inherits properties from another class is called Sub class or Derived Class.

Super Class:

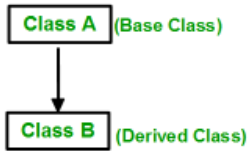
The class whose properties are inherited by sub class is called Base Class or Super class.



Inheritance Type

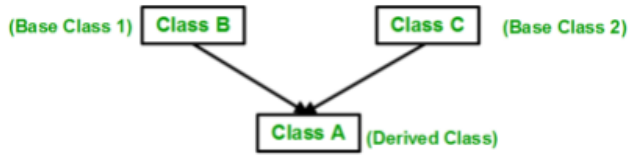
Single Inheritance:

In single inheritance, a class is allowed to inherit from only one class



Multiple Inheritance:

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one sub class is inherited from more than one base classes



inheritance

Class ***inheritance*** is one of the main characteristics of the OOP approach.

If a ***base class*** such is declared, a ***derived class*** may also be declared which takes on all the attributes of base class and adds more. The derived class is said to inherit the base class.

Advantage of C++ Inheritance
Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

- Single inheritance
- Multi Level Inheritance
- Multiple inheritance

Inheritance Basics

Derived Classes

A Derived class is defined as the class derived from the base class.
The Syntax of Derived class:

```
class derived_class_name : visibility-mode base_class_name
        {
        // body of the derived class.
        }
```

derived_class_name: It is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are ***publicly*** inherited or ***privately*** inherited. ***It can be public or private.***

base_class_name: It is the name of the base class.

publicly inherited

When the base class is ***publicly inherited*** by the derived class, ***public members*** of the ***base class*** also become the ***public members*** of the ***derived class***.

Therefore, the ***public members*** of the ***base clas***s are ***accessible by the objects*** of the ***derived class as well as*** by the ***member functions of the base class***.

```
class derived_class_name : public base_class_name
        {
        // body of the derived class.
        }
```

privately inherited

When the base class is ***privately inherited*** by the derived class, ***public members*** of the ***base class*** becomes the ***private members*** of the ***derived class***.

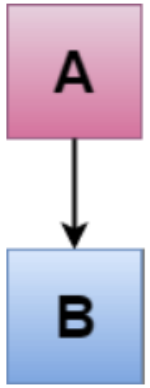
Therefore, the ***public members*** of the ***base class*** are ***not accessible*** by the ***objects of the derived class*** only by the ***member functions of the derived class***.

```
class derived_class_name : private base_class_name
        {
        // body of the derived class.
        }
```

C++ Single Inheritance

Single inheritance:

is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

```
#include <iostream>
using namespace std;
class Animal {
  public:
      int weight;
      void eat() {
          cout<<"Eating..."<<endl;
      }
};
class Dog: public Animal
{
    public:
        void bark(){
          cout<<"Barking...";
          }
};

int main(void) {
   Animal a1;
   Dog d1;


   a1.weight=20;
   cout << a1.weight <<endl;

   a1.eat();
   a1.weight=20;

    d1.weight=15;
    cout << d1.weight <<endl;
    d1.eat();
    d1.bark();

   return 0;
}
```

C++ Single Inheritance

```
using namespace std;
class A
{
   int a = 4;
   int b = 5;
   public:
   int mul()
   {
      int c = a*b;
      return c;
   }
};

class B : private A
{
   public:
   void display()
   {
      int result = mul();
      std::cout <<"Multiplication of a and b is : "<<result<< std::endl;
   }
};
```

```
int main()
{
   B b;
   b.display();

    return 0;
}
```

```
Multiplication of a and b is : 20
```

How to make a Private Member Inheritable

The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a ***third visibility modifier***, i.e., **protected**. The member which is declared as protected will be accessible to all the member functions within the class as well as the class ***immediately derived*** from it.

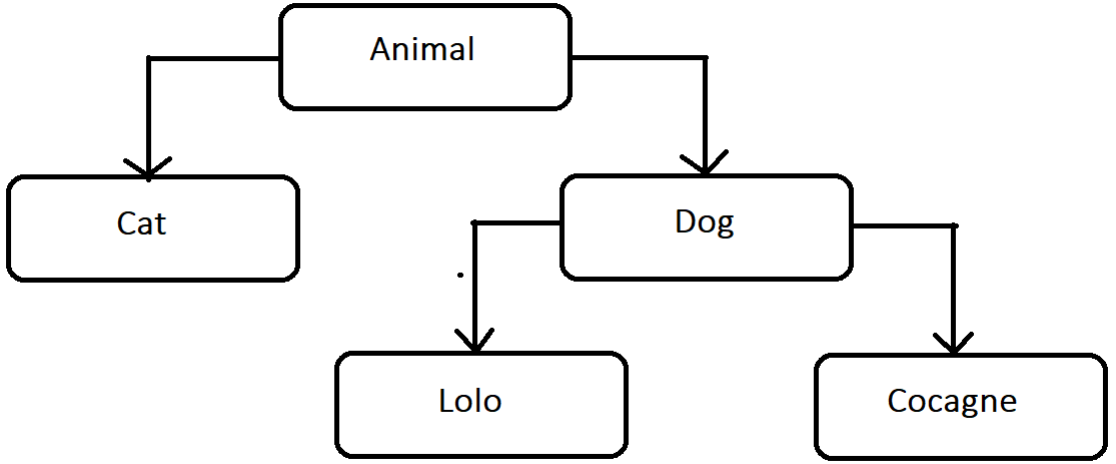
Visibility of Inherited Members

Base class visibility	Derived class visibility		
	Public	**Private**	**Protected**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Multi Level Inheritance

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++.

Inheritance is transitive so the last derived class acquires all the members of all its base classes.



Multilevel inheritance is a process of deriving a class from another derived class.

Multi Level Inheritance

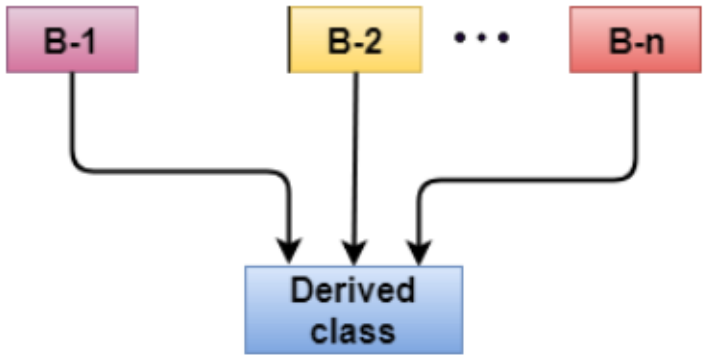
```
#include <iostream>
using namespace std;
 class Animal {
  public:
   void eat() {
      cout<<"Eating..."<<endl;
   }
};
   class Dog: public Animal
   {
      public:
       void bark(){
          cout<<"Barking..."<<endl;
          }
   };
class Cat: public Animal
   {
      public:
       void meow(){
          cout<<"Meow..."<<endl;
          }
   };
```

```
class Lolo: public Dog
   {
      public:
     void play() {
        cout<<"Play..."<<endl;
        }
   };

class Cocagne: public Dog
   {
      public:
      void attack() {
       cout<<"Attack..."<<endl;
       }
   };
```

```
int main(void) {
    Lolo d1;
    Cocagne d2;

     cout << " Lolo dog is " <<endl;
    d1.eat();
    d1.bark();
    d1.play();

    cout << " Cocagne dog is " <<endl;
    d2.eat();
    d2.bark();
    d2.attack();
    return 0;
}
```

C++ Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.

C++ Multiple Inheritance

```
#include <iostream>
using namespace std;
class A
{
   protected:
      int a;
   public:
      void set_a(int n)  { a = n; }
};

class B
{
   protected:
      int b;
   public:
      void set_b(int n)  { b = n; }
};
```

```
class C : public A,public B
{
   public:
    void display()
    {
        std::cout << "The value of a is : " <<a<< std::endl;
        std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b;
    }
};

int main()
{
   C c;
   c.set_a(10);
   c.set_b(20);
   c.display();
    return 0;
}
```

Ambiguity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

```
#include <iostream>
using namespace std;
class A
{
   public:
   void display()
   {
      std::cout << "Class A" << std::endl;
   }
};
class B
{
   public:
   void display()
   {
      std::cout << "Class B" << std::endl;
   }
};
class C : public A, public B
{
   public:
   void view()
   {
      display();
   }
};
int main()
{
   C c;
   c.view();
   return 0;
}
```

The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
class C : public A, public B
{
   public:
   void view()
   {
      A :: display();      // Calling the display() function of class A.
      B :: display();      // Calling the display() function of class B.
   }
};
```

Mechanism Of constructor and Distractor

```cpp
#include <iostream>
using namespace std;
class base
{
public:
base() {cout<< "Constructing base class\n";}
~base() {cout<< "Destructing base class\n";}
};

class derived: public base
{
public:
derived() {cout <<"Constructing derived class\n";}
~derived() { cout << "Destructing derived class\n";}
};

int main()
{
derived ob;
}
```

```
Constructing base class
Constructing derived class
Destructing derived class
Destructing base class
```

Mechanism Of constructor and Distractor

```
#include <iostream>
using namespace std;
class base
{
   int i;
public:
   base(int n) {cout<< "Constructing base class\n";
                i = n;
}
~base() {cout<< "Destructing base class\n";}
void showi() {cout<< "i= "<< i<< '\n';}
};

class derived: public base
{
   int j;
public:
   derived(int n,int m):base(m) {
      cout <<"Constructing derived class\n";
         j=n; }
~derived() {cout<< "Destructing derived class\n";}
void showj() { cout << "j= " <<j << "\n";}
};

int  main()
{
   derived ob(20,30);
   ob.showi();
   ob.showj();
}
```

```
Constructing base class
Constructing derived class
i= 30
j= 20
Destructing derived class
Destructing base class
```

C++ Function Overriding

If derived class defines same function as defined in its base class, it is known as function ***overriding*** in C++.

It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

To overriding method must be the ***same return*** type ***same function*** name and ***same parameters***

```
Eating bread...
```

```
#include <iostream>
using namespace std;
class Animal {
   public:
   void eat(){
   cout<<"Eating...";
      }
};
class Dog: public Animal
{
    public:
    void eat()
      {
        cout<<"Eating bread...";
      }
};
int main(void) {
  Dog d = Dog();
  d.eat();
  return 0;
}
```

Access Overridden Function in Base

To access the overridden function of the base class, we use the scope resolution operator ::.

We can also access the overridden function by using a pointer of the base class to point to an object of the derived class and then calling the function from that pointer.

```
class Base {
    public:
      void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
      }
};

int main() {
    Derived derived1, derived2;

    derived1.print();

    derived2.Base::print();

    return 0;
}
```

Access overridden function using object of derived class in C++

```
class Base {
    public:
      void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
          Base::print();
      }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

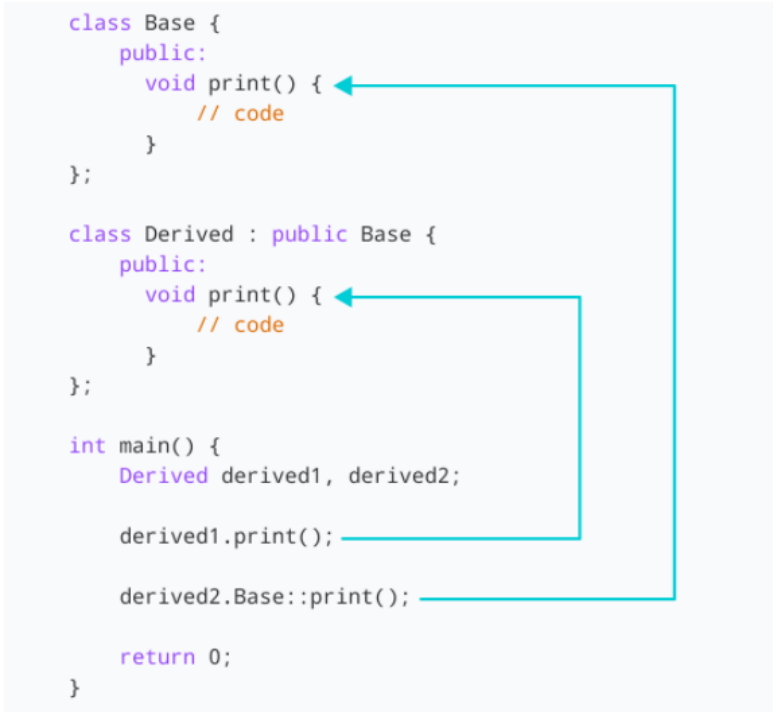
Access overridden function inside derived class in C++

Access Overridden Function in Base

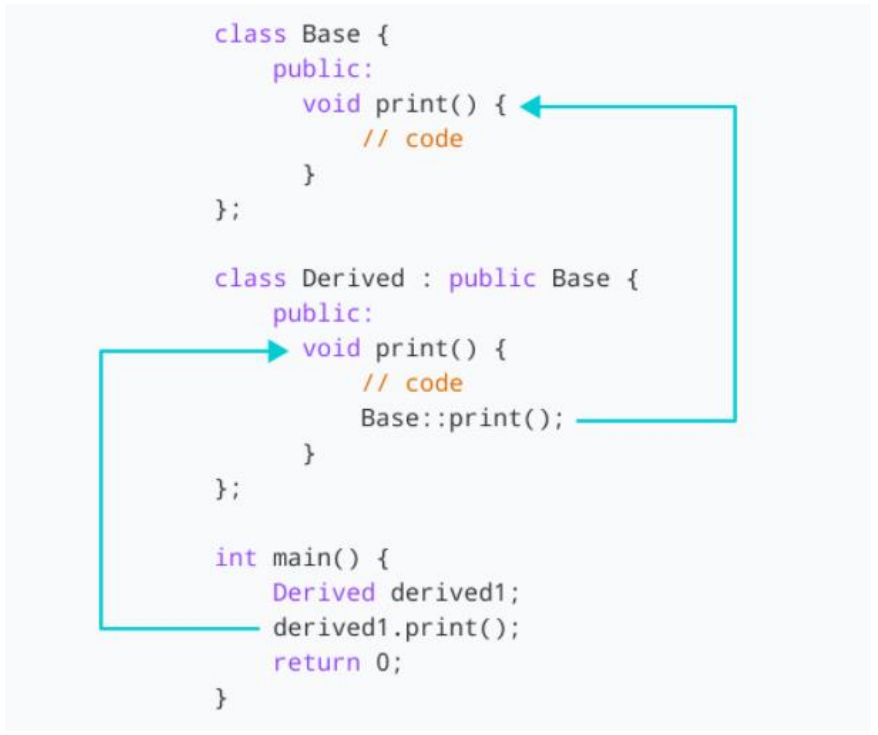
To access the overridden function of the base class, we use the scope resolution operator ::.

We can also access the overridden function by using a pointer of the base class to point to an object of the derived class and then calling the function from that pointer.

To access the overridden function of the base class, You can also do that by creating the child class object in such a way that the reference of parent class points to it.



Access overridden function using object of derived class in C++



Access overridden function inside derived class in C++

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

class Base {
  public:
   void print() {
      cout << "Base Function" << endl;
   }
};

class Derived : public Base {
  public:
   void print() {
      cout << "Derived Function" << endl;

   }
};

int main() {
   Base derived1= Derived();
   derived1.print();
   return 0;
}
```

Call Overridden Function Using Pointer

If you created a pointer of Base type named. This pointer points to the Derived object .

When we call the print() function using this pointer, it calls the overridden function from Base This is because even though the pointer points to a Derived object, it is actually of Base type. So, it calls the member function of Base.

In order to override the Base function instead of accessing it, we need to use ***virtual functions in the Base class.***

```
#include <iostream>
using namespace std;
class Base {
  public:
   void print() {
      cout << "Base Function" << endl;
   }
};
class Derived : public Base {
  public:
   void print() {
      cout << "Derived Function" << endl;
   }
};
int main() {
   Derived derived1;
   Base* ptr = &derived1;
   ptr->print();
   return 0;
}
```

Polymorphism

term "***Polymorphism***" is the combination of "poly" + "morphs" which means many forms.

It is a greek word.

There are two types of polymorphism in C++:

1) **Compile time polymorphism:** It is achieved by function overloading and operator overloading which is also known as static binding or early binding.

2) Runtime polymorphism: It is achieved by method overriding which is also known as dynamic binding or late binding.

```
#include <iostream>
using namespace std;
class Animal {
      public:
   virtual void speak()
     {
        cout<<"~~~~~~~~~"<<endl;
     }
};
class Dog: public Animal     {
    public:
    void speak()
     {
       cout<<"Haw Haw Haw"<<endl;
     }
  };
class Cat: public Animal {
   public:
    void speak()
     {
       cout<<"Miao Miao Miao"<<endl;
     }
};
   int main(void) {
     Dog d ;
     Cat c ;
     cout << "===Polymorphism===="<<endl;
     Animal * ptr;
     ptr=&d;
     ptr->speak();
     ptr=&c;
     ptr->speak();
     return 0;
   }
```

pure virtual function

A ***pure virtual function*** is a virtual function in C++ for which we need not to write any function definition and only we have to declare it. It is declared by assigning 0 in the declaration.

```
1  #include <iostream>
2  using namespace std;
3  class Animal {
4        public:
5     virtual void speak()=0;
6  };
7  class Dog: public Animal     {
8      public:
9
10    };
11 class Cat: public Animal {
12    public:
13     void speak()
14       {
15          cout<<"Miao Miao Miao"<<endl;
16       }
17 };
18    int main(void) {
19       Dog d ;
20       Cat c ;
21       cout << "===Polymorphism====="<<endl;
22       Animal * ptr;
23       ptr=&d;
24       ptr->speak();
25       ptr=&c;
26       ptr->speak();
27       return 0;
28    }
29
```

abstract class

An **abstract class** is a **class** that is designed to be specifically used as a base **class**.

An **abstract class** contains at least one pure virtual function.

A **class** derived from an **abstract** base **class** will also be **abstract** unless you override each pure virtual function in the derived **class.**

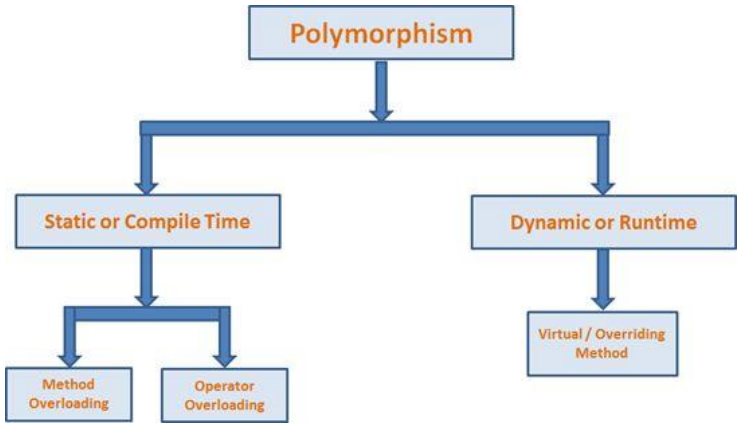
```
#include <iostream>
using namespace std;
// Base class
class Shape {
  public:
    // pure virtual function providing interface framework.
    virtual int getArea() = 0;
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }
  protected:
    int width;
    int height;
};
// Derived classes
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);
    }
};
class Triangle: public Shape {
  public:
    int getArea() {
      return (width * height)/2;
    }
};
```

```
int main(void) {
  Rectangle Rect;
  Triangle  Tri;

  Rect.setWidth(5);
  Rect.setHeight(7);
  // Print the area of the object.
  cout << "Total Rectangle area: " << Rect.getArea() << endl;
  Tri.setWidth(5);
  Tri.setHeight(7);
  // Print the area of the object.
  cout << "Total Triangle area: " << Tri.getArea() << endl;
  return 0;
}
```

Polymorphism

It describes the concept that different classes can be used with the same interface. Each of these classes can provide its own implementation of the interface.

With Polymorphism, a message is sent to multiple class objects, and every object responds appropriately according to the properties of the class. Consider the operation of addition for two numbers the operation will generate a sum if the operands are string then the operation will produce the concatenation of the two string .



Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

12 + 13 ⇨ 25
"youssef"+" "+"shawky" ⇨ "youssef shawky"

C++ language
Lesson -15

Relations

OOP relationships

One of the advantages of Object-Oriented programming language is code reuse. This reusability is possible due to the relationship b/w the classes.

Object oriented programming generally support 4 types of relationships that are: ***inheritance*** , ***association***, ***composition*** and aggregation. All these relationship is based on "is a" ***relationship***, "has-a" relationship and "part-of" relationship.

Inheritance

Inheritance is “IS-A” type of relationship. “IS-A” relationship is a totally based on Inheritance.

Inheritance is a parent-child relationship where we create a new class by using existing class code. It is just like saying that “A is type of B”. For example is “Dog is an Animal”, “Employee is a Person”.

For better understanding let us take a real world scenario:

•HOD is a staff member of college.
•All teachers are staff member of college.
•HOD and teachers has id card to enter into college.
•HOD has a staff that work according the instruction of him.
•HOD has responsibility to undertake the works of teacher to cover the course in fixed time period.

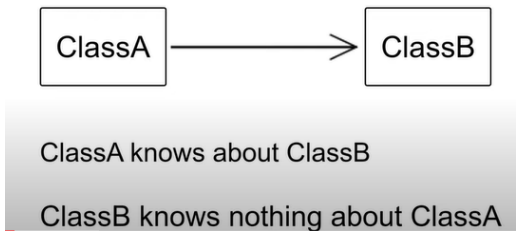
Let us take first two assumptions , “HOD is a staff member of college” and “All teachers are staff member of college”. For this assumption we can create a “StaffMember” parent class and inherit this parent class in “HOD” and “Teacher” class.

Association

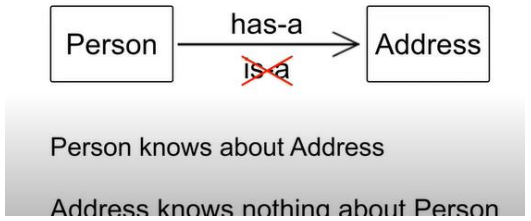
Association is a “has-a” type relationship. Association establish the relationship b/w two classes using through their objects. Association relationship can be one to one, One to many, many to one and many to many. For example suppose we have two classes then these two classes are said to be “has-a” relationship if both of these entities share each other’s object for some work and at the same time they can exists without each others dependency or both have their own life time.

As an example, imagine the relationship between a doctor and a patient. A doctor can be associated with multiple patients. At the same time, one patient can visit multiple doctors for treatment or consultation. Each of these objects has its own life cycle and there is no “owner” or parent. The objects that are part of the association relationship can be created and destroyed independently.

Association (Unary)

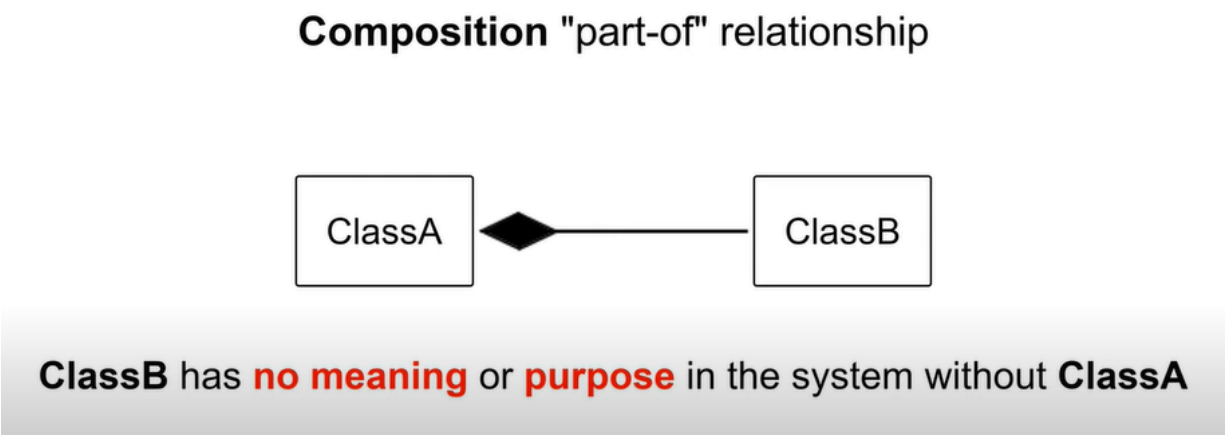


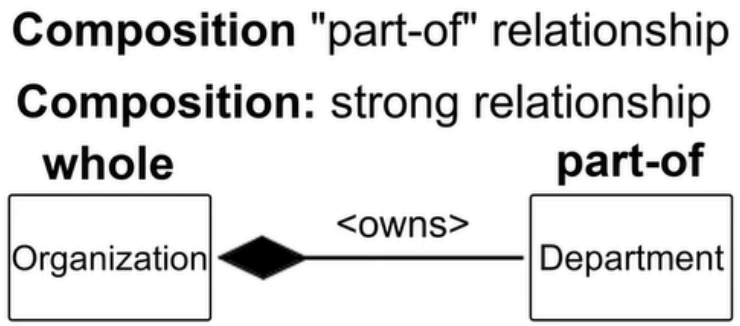
Association (Unary)



Composition

Composition is a "part-of" relationship. Simply composition means mean use of instance variables that are references to other objects. In composition relationship both entities are interdependent of each other for example “engine is part of car”, “heart is part of body”.



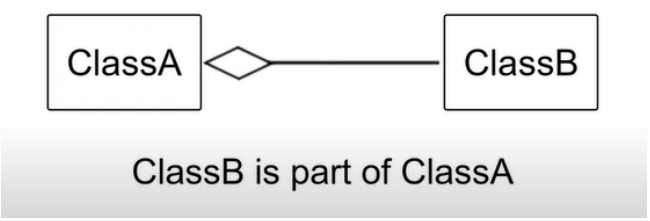


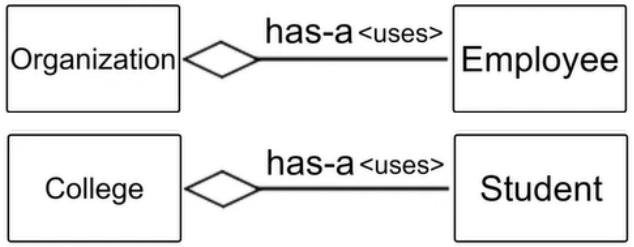
In both **aggregation** and **composition** object of one class "owns" object of another class.

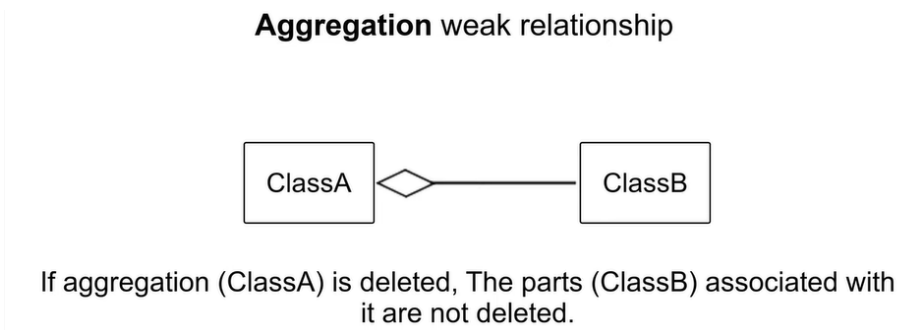
Aggregation

Aggregation is based is on "has-a" relationship. Aggregation is a special form of association. In association there is not any classes (entity) work as owner but in aggregation one entity work as owner. In aggregation both entities meet for some work and then get separated. Aggregation is a one way association.

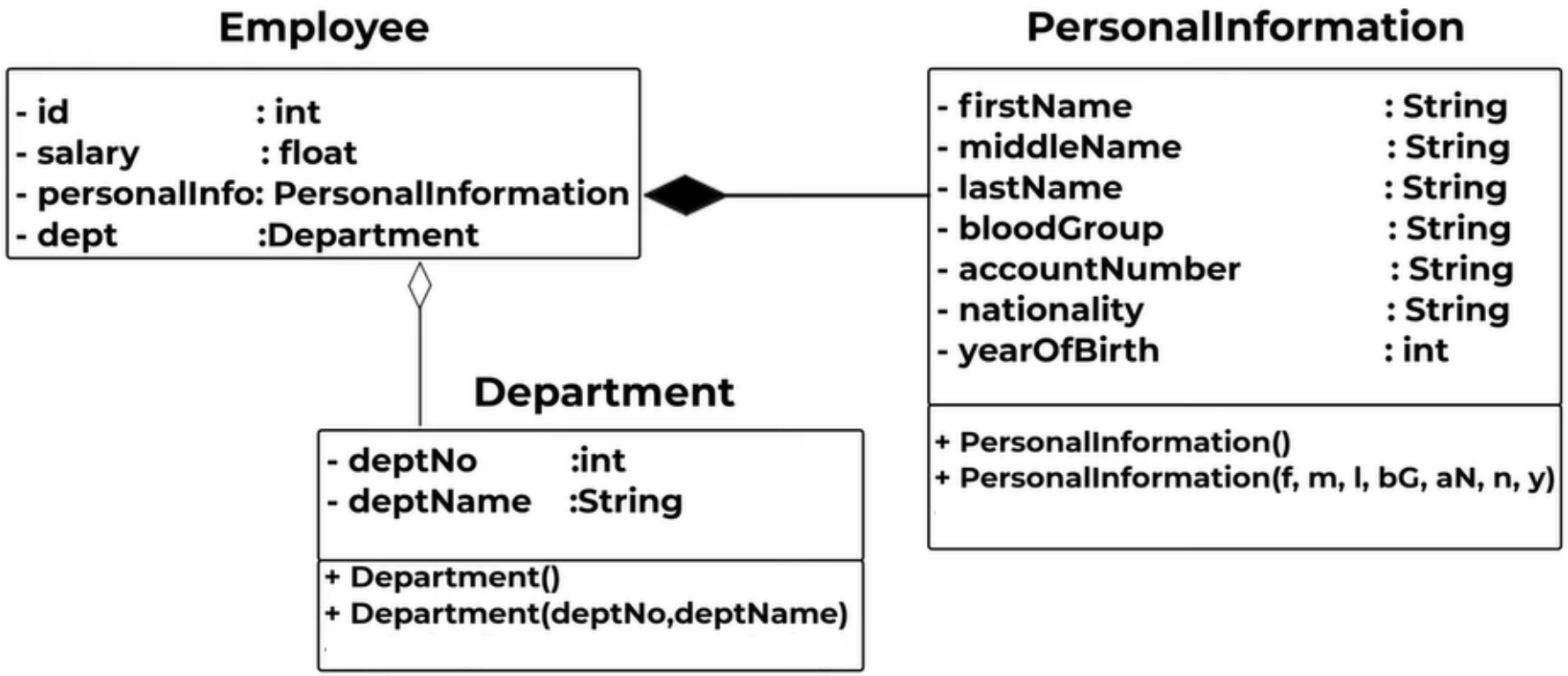
Let us take an example of “Student” and “address”. Each student must have an address so relationship b/w Student class and Address class will be “Has-A” type relationship but vice versa is not true(it is not necessary that each address contain by any student). So Student work as owner entity. This will be a aggregation relationship.







Aggregation or Composition



Aggregation Example

```
class Point
{
 private:
                int x ;
                int y ;
 public:
                Point()
                {
                                x = 0 ;
                                y = 0 ;
                }
                Point(int m, int n)
                {
                                x = m ;
                                y = n ;
                }
};
```

```
class Line
{
 private:
                Point start;
                Point end;
 public:
                Line() : start(), end()
                {               }
                Line(int x1, int y1, int x2, int y2) : start(x1,y1), end(x2,y2)
                {               }
};
```

```
class Circle
{
 private :
                Point center;
                int   radius;
 public :
                Circle() : center()
                {
                                radius = 0 ;
                }
                Circle(int m, int n, int r) : center(m,n)
                {
                                radius = r ;
                }
};
```

```
class Rect
{
 private:
                Point ul;
                Point lr;
 public:
                Rect() : ul(), lr()
                {               }
                Rect(int x1, int y1, int x2, int y2) : ul(x1,y1), lr(x2,y2)
                {               }
};
```

C++ language
Lesson -16

File Handling In C++

File Handling In C++

Files are used to store data in a storage device permanently.

A stream is an abstraction that represents a device on which operations of input and output are performed.
A stream can be represented as a source or destination of characters of indefinite length depending on its usage.

In C++ we have a set of file handling methods. These include ***ifstream, ofstream, and fstream***.

These classes, designed to manage the disk files, are declared in fstream and therefore we must include fstream and therefore we must include this file in any program that uses files.

File Handling In C++

- ***ofstream***: This Stream class signifies the output file stream and is applied to create files for writing information to files

- ***ifstream***: This Stream class signifies the input file stream and is applied for reading information from files

- ***fstream***: This Stream class can be used for both read and write from/to files.

ifstream is input file stream which allows you to read the contents of a file.
ofstream is output file stream which allows you to write contents to a file.
fstream allows both reading from and writing to files by default. However, you can have an fstream behave like an ifstream or ofstream by passing in the ios::open_mode flag.

File Handling In C++

All the above three classes are derived from fstreambase and from the corresponding iostream class and they are designed specifically to manage disk files.

C++ provides us with the following operations in File Handling:

Creating a file:	open()
Reading data:	read()
Writing new data:	write()
Closing a file:	close()

Opening a File

We can open a file using any one of the following methods:
1. First is bypassing the file name in constructor at the time of object creation.

2. Second is using the open() function.

void open(const char* file_name,ios::openmode mode);

fstream new_file;
new_file.open("newfile.txt", ios::out);

In the above example, new_file is an object of type fstream, as we know fstream is a class so we need to create an object of this class to use its member functions.

Default Open Modes :
ifstream ios::in
ofstream ios::out
fstream ios::in | ios::out

Modes	Description
in	Opens the file to read(default for ifstream)
out	Opens the file to write(default for ofstream)
binary	Opens the file in binary mode
app	Opens the file and appends all the outputs at the end
ate	Opens the file and moves the control to the end of the file
trunc	Removes the data in the existing file
nocreate	Opens the file only if it already exists
noreplace	Opens the file only if it does not already exist

close() function

As soon as the program terminates, the memory is erased and frees up the memory allocated and closes the files which are opened. But it is better to use the close() function to close the opened files after the use of the file.

write and read

Using a stream insertion operator << we can write information to a file and using stream extraction operator >> we can easily read information from a file.

ofstream

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{

    ofstream out;

    out.open("d:\\output.txt");
    out << "hello youssef"<<endl;
    out.close();
}
```

ifstream

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ifstream in;
    int x,y,z;
    int sum;

    in.open("d:\\data.txt");
    if (!in)
                    cout << "can\'t open file data.txt"<<endl;
    else
   {
    in >> x>>y>>z;
                sum = x+y+z;
    cout << "sum of "<<x<<" and "<<y<< " and "<<z<<" = "<< sum;
    in.close();
   }
}
```

ifstream to read stream of characters

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ifstream in;
    char ch;

    in.open("d:\\messages.txt");
    if (!in)
                    cout << "can\'t open file messages.txt"<<endl;
    else
   {
    while (!in.eof())
    {
        in.get(ch);
        cout << ch;
    }
    in.close();
   }
}
```

ifstream to read stream of strings

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ifstream in;
    string item;
    int count=0;

    in.open("d:\\messages.txt");
    if (!in)
                    cout << "can\'t open file messages.txt"<<endl;
    else
   {
    while (!in.eof())
    {
        in >> item;
        cout << item<<endl;
        count++;
    }
    cout<< "number of items = "<< count<<endl;
    in.close();
   }
}
```

ifstream to read lines

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
   ifstream in;
   char item[80];

   int count=0;

   in.open("d:\\messages.txt");
   if (!in)
                  cout << "can\'t open file messages.txt"<<endl;
   else
  {
   while (!in.eof())
   {
      in.getline(item,80) ;
      cout << item<<endl;
      count++;
   }
   cout<< "number of items = "<< count<<endl;
   in.close();
  }
}
```

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
   ifstream in;
   string line;

   int count=0;

   in.open("d:\\messages.txt");
   if (!in)
                  cout << "can\'t open file messages.txt"<<endl;
   else
  {
   while (!in.eof())
   {
      getline(in,line) ;
      cout << line<<endl;
      count++;
   }
   cout<< "number of items = "<< count<<endl;
   in.close();
  }
}
```

ifstream to read stream of integers

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ifstream in;
    int item;
    int sum=0;
    int count=0;

    in.open("d:\\numbers.txt");
    if (!in)
                        cout << "can\'t open file numbers.txt"<<endl;
    else
   {
    while (!in.eof())
    {
        in >> item;
        cout << item<<endl;
        sum +=item;
        count++;
    }
    cout<< "number of items = "<< count<<endl;
    cout<< "total ="<<sum;
    in.close();
   }
}
```

read and write object to file

```
file_obj.write((char*)&obj, sizeof(obj));

file_obj.read((char*)&obj, sizeof(obj));
```

read and write object to file

```
#include <iostream>
#include <fstream>
using namespace std;
// Class to define the properties
class Employee {
public:
   string Name;
   int Employee_ID;
   int Salary;
};
```

```
int main(){
   Employee Emp_1;
   Emp_1.Name="John";
   Emp_1.Employee_ID=2121;
   Emp_1.Salary=11000;
   //Wriring this data to Employee.txt
   ofstream file1;
   file1.open("Employee.txt", ios::app);
   file1.write((char*)&Emp_1,sizeof(Emp_1));
   file1.close();
   //Reading data from EMployee.txt
   ifstream file2;
   file2.open("Employee.txt",ios::in);
   file2.seekg(0);
   file2.read((char*)&Emp_1,sizeof(Emp_1));
   printf("\nName :%s",Emp_1.Name);
   printf("\nEmployee ID :%d",Emp_1.Employee_ID);
   printf("\nSalary :%d",Emp_1.Salary);
   file2.close();
   return 0;
}
```

overloaded stream insertion /extraction operators

C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<.

The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.

Here, it is important to make operator overloading function a friend of the class because it would be called without creating an object.

```
friend ostream &operator<<( ostream &output, const Complex &D )
{
   output << D.r<< std::showpos << D.i<<"i"<<endl;
   return output;
}
```

```
friend istream &operator>>( istream  &input, Complex &D ) {
   input >> D.r >> D.i;
   return input;
}
```

C++ language
Lesson -17

Standard Template Library (STL)

Standard Template Library (STL)

According to ***Bjarne Stroustrup***: “The C++ standard library containers were designed to meet **two criteria**: to provide the ***maximum freedom*** in the design of an individual container, while at the same time allowing containers ***to present a common interface*** to users. This allows optimal efficiency in the implementation of containers and enables users to write code that is independent of the particular container used.”

What is Class Templates ?

Class Templates Like function templates, class templates are useful when a class defines something that is independent of the data type. Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

```
#include <iostream>
#include <string>
using namespace std;
#define SIZE 5
template <class T>
class Stack
 {
private:
   int top;
   T st[SIZE];
public:
   Stack();
   void push(T k);
   T pop();
   T topElement();
   bool isFull();
   bool isEmpty();
};
```

```
template <class T>
Stack<T>::Stack() { top = -1; }

template <class T>
void Stack<T>::push(T k)
{
   if (isFull()) {
      cout << "Stack is full\n";
   }
   cout << "Inserted element " << k << endl;
   top = top + 1;
   st[top] = k;
}

template <class T>
bool Stack<T>::isEmpty()
{
   if (top == -1)
      return 1;
   else
      return 0;
}
```

```
template <class T>
bool Stack<T>::isFull()
{
   if (top == (SIZE - 1))
      return 1;
   else
      return 0;
}
template <class T>
T Stack<T>::pop()
{
   T popped_element = st[top];
   top--;
  return popped_element;
}
template <class T>
T Stack<T>::topElement()
{
   T top_element = st[top];
   return top_element;
}
```

```
int main()
{
   Stack<int> integer_stack;
   Stack<string> string_stack;
   integer_stack.push(10);
   integer_stack.push(20);
   integer_stack.push(30);
   string_stack.push("Youssef");
   string_stack.push("Mohab");
   string_stack.push("Namir");
   cout << integer_stack.pop() << " is removed from stack" << endl;
   cout << string_stack.pop() << " is removed from stack " << endl;
   cout << "Top element is " << integer_stack.topElement()<< endl;
   cout << "Top element is " << string_stack.topElement()  << endl;
   return 0;
}
```

Standard Template Library (STL)

The Standard Template Library (STL) is a set of C++ template classes to provide common programming ***data structures*** and ***functions*** such as lists, stacks, arrays, etc. It is a library of ***container*** classes, ***algorithms***, and ***iterators***. It is a generalized library and so, its components are parameterized.

•algorithm defines a collection of functions especially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of the containers.

Containers or container classes store objects and data.

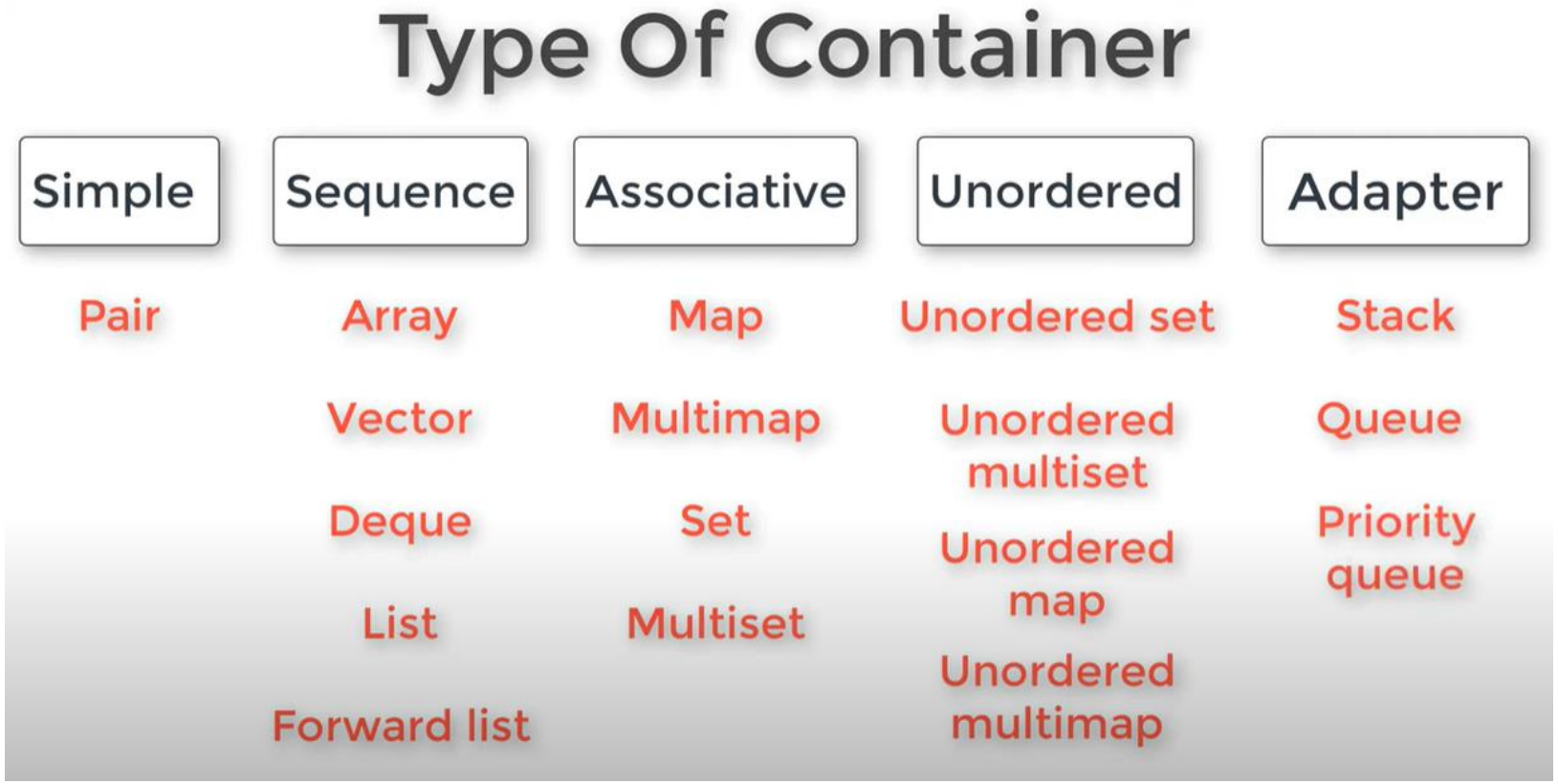
The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed.

iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

containers

The a container is, a container manages a collection of objects. I have learned that C++ contains four types of containers:

- Sequential Containers
- Associative Containers
- Adapters Containers
- Unordered Containers



Sequence Containers

There are five sequence containers offered by C++ Standard Template Library. They are: array, vector, deque, forward_list and list.

The container classes array, vector, and deque are implemented by using an array data structure. And the container classes, list and forward_list, are implemented using a linked list data structure.

The basic difference between these two types of data structures is that arrays are static in nature. This means the size of the container is fixed and set during compilation, although there is a provision to increase the size on requirement. The linked-list implementation is purely dynamic in nature.

Vectors

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.

Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.

In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array.

Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

Vectors

size() – Returns the number of elements in the vector.

max_size() – Returns the maximum number of elements that the vector can hold.

capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.

resize(n) – Resizes the container so that it contains ‘n’ elements.

empty() – Returns whether the container is empty.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
   vector<int> v1;
   vector<int>::iterator it;

   for (int i = 1; i <= 5; i++)
      v1.push_back(i*10);

   for (int i=0;i<v1.size();i++)
      cout << v1[i]<<"\t";
   cout<<endl;

   cout << "Size : " << v1.size();
   cout << "\nCapacity : " << v1.capacity();
   cout << "\nMax_Size : " << v1.max_size();

   v1.resize(3);
   cout << "\nSize : " << v1.size();
   if (v1.empty() == false)
      cout << "\nVector is not empty";
   else
      cout << "\nVector is empty";

   cout << "\nVector elements are: ";
   for (it = v1.begin(); it != v1.end(); it++)
      cout << *it << " ";

   return 0;
}
```

Vectors iterators

begin() – Returns an iterator pointing to the first element in the vector

end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector

rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element

rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

cbegin() – Returns a constant iterator pointing to the first element in the vector.

cend() – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   vector<int> v1;

   for (int i = 1; i <= 5; i++)
      v1.push_back(i);

   cout << "Output of begin and end: ";
   for (auto i = v1.begin(); i != v1.end(); ++i)
      cout << *i << " ";

   cout << "\nOutput of cbegin and cend: ";
   for (auto i = v1.cbegin(); i != v1.cend(); ++i)
      cout << *i << " ";

   cout << "\nOutput of rbegin and rend: ";
   for (auto ir = v1.rbegin(); ir != v1.rend(); ++ir)
      cout << *ir << " ";

   cout << "\nOutput of crbegin and crend : ";
   for (auto ir = v1.crbegin(); ir != v1.crend(); ++ir)
      cout << *ir << " ";

   return 0;
}
```

Vectors Modifiers

assign() – It assigns new value to the vector elements by replacing old ones

push_back() – It push the elements into a vector from the back

pop_back() – It is used to pop or remove elements from a vector from the back.

insert() – It inserts new elements before the element at the specified position

erase() – It is used to remove elements from a container from the specified position or range.

swap() – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.

clear() – It is used to remove all the elements of the vector container

emplace() – It extends the container by inserting new element at position

emplace_back() – It is used to insert a new element into the vector container, the new element is added to the end of the vector

.

```
#include <bits/stdc++.h>
#include <vector>
using namespace std;
int main()
{
   vector<int> v;
   v.assign(5, 10);
   cout << "The vector elements are: ";
   for (int i = 0; i < v.size(); i++)
      cout << v[i] << " ";
   v.push_back(15);
   int n = v.size();
   cout << "\nThe last element is: " << v[n - 1];
   v.pop_back();
   cout << "\nThe vector elements are: ";
   for (int i = 0; i < v.size(); i++)
      cout << v[i] << " ";
   v.insert(v.begin(), 5);
   cout << "\nThe first element is: " << v[0];
   v.erase(v.begin());
   cout << "\nThe first element is: " << v[0];
   v.emplace(v.begin(), 5);
   cout << "\nThe first element is: " << v[0];
   v.emplace_back(20);
   n = v.size();
   cout << "\nThe last element is: " << v[n - 1];
   v.clear();
   cout << "\nVector size after erase(): " << v.size();
   vector<int> v1, v2;
   v1.push_back(1);
   v1.push_back(2);
   v2.push_back(3);
   v2.push_back(4);

   cout << "\n\nVector 1: ";
   for (int i = 0; i < v1.size(); i++)
      cout << v1[i] << " ";

   cout << "\nVector 2: ";
   for (int i = 0; i < v2.size(); i++)
      cout << v2[i] << " ";
   v1.swap(v2);
   cout << "\nAfter Swap \nVector 1: ";
   for (int i = 0; i < v1.size(); i++)
      cout << v1[i] << " ";
   cout << "\nVector 2: ";
   for (int i = 0; i < v2.size(); i++)
      cout << v2[i] << " ";
}
```

Queue in Standard Template Library (STL)

Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front. Queues use an encapsulated object of deque or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

The functions supported by queue are

empty() – Returns whether the queue is empty.
size() – Returns the size of the queue.
queue::swap() in C++ STL: Exchange the contents of two queues but the queues must be of same type, although sizes may differ.
queue::emplace() in C++ STL: Insert a new element into the queue container, the new element is added to the end of the queue.
queue::front() and queue::back() in C++ STL– front() function returns a reference to the first element of the queue. back() function returns a reference to the last element of the queue.
push(g) and pop() – push() function adds the element ‘g’ at the end of the queue. pop() function deletes the first element of the queue.

functions supported by queue

```
int main()
{
	// declaring an empty queue
	queue<int> Q;

	//inserting elements
	Q.push(10);
	Q.push(20);
	Q.push(30);
	cout<<"Queue size before printing the elements: "<<Q.size()<<endl;
	cout<<"Queue element are..."<<endl;
	while(!Q.empty()){
		cout<<" "<<Q.front();
		Q.pop();
	}

	cout<<endl;
	cout<<"Queue size after printing the elements: "<<Q.size()<<endl;

	return 0;
}
```

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

void showq(queue<int> q)
{
   queue<int> tq = q;
   while (!tq.empty()) {
      cout << '\t' << tq.front();
      tq.pop();
   }
   cout << '\n';
}
int main()
{
   queue<int> q1;

   q1.push(10);
   q1.push(20);
   q1.push(30);
   cout << "The queue q1 is : ";
   showq(q1);
   cout << "\nq1.size() : " << q1.size();
   cout << "\nq1.front() : " << q1.front();
   cout << "\nq1.back() : " << q1.back();
   cout << "\nq1.pop() : ";
   q1.pop();
   showq(q1);
   return 0;
}
```