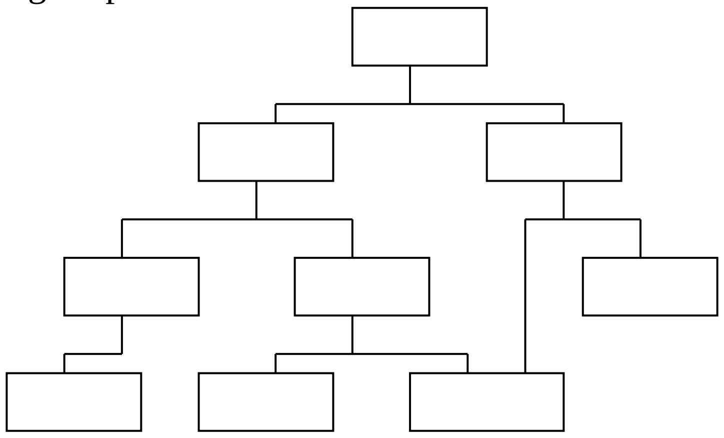
C++ Language

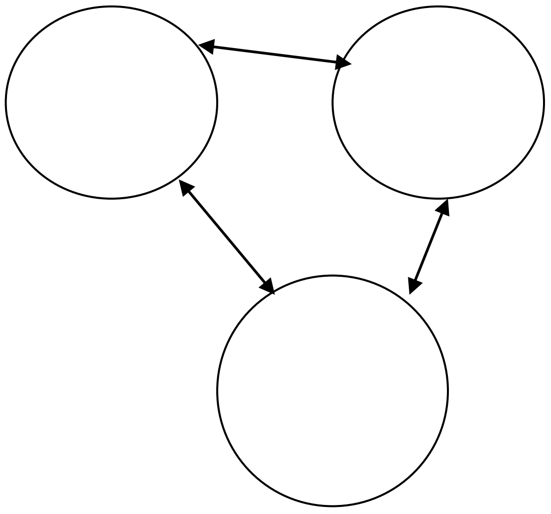
The C++ is object oriented language. The object oriented Language is a new technique in programming and in thinking.

Before the appearance of OO Concept, all the programmer used the STRUCTURED Programming in which the program is divided into functions each perform a specified task. At the end, the program has the following shape



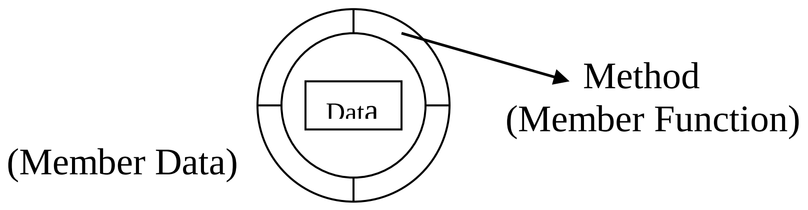
This type of programming is very important but to a certain limit, because it stress on the function and once the project reach a certain size, its complexity becomes too difficult to manage

The Object Oriented Programming takes the best ideas of structured programming and combines them with powerful, new concepts that encourage you to look at the task of programming in a new light.
In the OO technique, allows you to decompose a problem into subgroups of related parts. These subgroups are independent on each other but can deal together to make the program.
Each subgroup has its own DATA and has some METHOD that works with the Data. The program has the following shap



Each Component that contain the Data and the Method is called OBJECT
The Method is a function in the object that deal with Data
The Template that I create from it an Object is called Class.
Any program simulate a real life problem, the deal between object is through Message. The Message activate a method of the object.

So, we can say that the object is composed of 2 things
The data and the method that ENCAPSULATE the data



<u>Object Oriented Technology based on:</u>

Encapsulation:

The Data is encapsulated by the method, this mean that no one can deal with the data directly and the only thing that can work with the data is the method of the same class. So we deal with the data through the method.
This is good because :

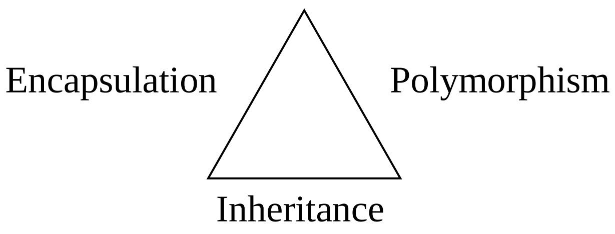
1) no one can change the data value directly, this provides a significant level of protection against accidental modification or incorrect use.
2) All the Member Data being local variable, this is easy to be maintained.

Inheritance:

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of classification. Most knowledge is made manageable by hierarchical classifications.(Ex: Rectangle and Square)

Polymorphism:

Polymorphism allows one name to be used for several related but slightly different purposes. The purpose of polymorphism is to let one name be used to specify a general class of action. Depending upon what type of data it is dealing with, a specific instance of the general case is executed.



Enhancement in C++ than C

- Flexible declaration (the variable is declared anywhere in the program

C Language	C++ Language

• struct MyStruct
{
 	
};
struct MyStruct name_struct;
	• struct MyStruct
	{
	
	};
	MyStruct name_struct;

• char ch;
int n;
n = ch;
	• char ch;
	int n;
	n = ch; //is wrong on C++
	//we must use type
	//casting

• long l;
float f;
f = (float) l;
	• long l;
	float f;
	f = float (l);

• int *ptr;
ptr = (int *),malloc(sizeof(int));
*ptr = 15;
free ptr;
	int *ptr;
	ptr = new(int); } ptr = new int(15);
	*ptr = 15;
	delete (ptr);
	
	To initialize an array:
	ptr = new int[10];

- The input and output instead of scanf and printf, we use:

```
        cin>>x;  //for input
        cout<<"First Name"<<x;  //for output
```

- Default Argument Value:

Sometimes, a function has always the same value as argument, In this case we can use the default argument value, in which we give default value to all or some of the argument, then when calling the function we can not passing these argument.

Example: a program to calculate the volume of a box

```
# include <stdio.h>
int BoxVolume( int l=1, int b=1, int n=1);
void main()
{
        int v1,v2,v3,v4;
        v1 = BoxVolume (10,2,2);
        v2 = BoxVolume (3,2);       //equivalent to BoxVolume(3,2,1)
        v2 = BoxVolume (5);         //equivalent to BoxVolume(5,1,1)
        v2 = BoxVolume ();          //equivalent to BoxVolume(1,1,1)
}
int BoxVolume( int l=1, int b=1, int n=1)
{
        return (l * b *n);
}
```

Note:

*)We can't use the default for the first parameter then putting argument for the other parameter, if we put the first default the other must have the default.

*)It is recommended to arrange the argument in a way that the mostly used default at the end of parameter.

- The C++ support a global variable and a local variable having the same name. To access the global variable in the function having a local variable with the same name we use :: scope operator

```
int n = 5;
void main()
{
        int n = 7;     //local variable
        n = 8;         //access the local variable
        ::n = 9;       //access the global variable
        printf("%d \n %d", n, ::n);
}
```

What Is a Reference?

A reference is an alias; when you create a reference, you initialize it with the name of another object, the target. From that moment on, the reference acts as an alternative name for the target, and anything you do to the reference is really done to the target.

You create a reference by writing the type of the target object, followed by the reference operator (`&`), followed by the name of the reference.

```
#include <iostream.h>
int main()
{
        int  intOne;
        int &rSomeRef = intOne;
        intOne = 5;
        cout << "intOne: " << intOne;
        cout << "rSomeRef: " << rSomeRef;
        rSomeRef = 7;
        cout << "intOne: " << intOne;
        cout << "rSomeRef: " << rSomeRef;
        return 0;
}
```

a local `int` variable, `intOne`, is declared, a reference to an `int`, `rSomeRef`, is declared and initialized to refer to `intOne`.

Note:

1) References must be initialized at the point of definition. If you declare a reference, but don't initialize it, you will get a compile-time error.
2) The address of a reference is the address of the aliased object.

```
  #include <iostream.h>
  int main()
  {
          int  intOne;
          int &rSomeRef = intOne;
          intOne = 5;
          cout << "intOne: " << intOne;
          cout << "rSomeRef: " << rSomeRef;
          cout << "&intOne: "  << &intOne;
          cout << "&rSomeRef: " << &rSomeRef;
          return 0;
  }
  Output: intOne: 5
  rSomeRef: 5
  &intOne:  0x3500
```

&rSomeRef: 0x3500

3) The sizeof() a reference is the sizeof() the object
4) Using constant reference parameter emulates ordinary “Pass by value” Or “Pass by copy” without the copying overhead.
5) A non-constant reference is a value that can be updated
6) Overloads to the index and assignment operators require reference return value
7) references cannot be reassigned

Passing Function Arguments by Reference:

Another way for calling function by reference is by using the reference operator.

```
#include <iostream.h>
void swap(int &x, int &y);
int main()
{
        int x = 5, y = 10;
        cout << "Main. Before swap, x: " << x << " y: " << y << "\n";
        swap(x,y);
        cout << "Main. After swap, x: " << x << " y: " << y << "\n";
        return 0;
}
void swap (int &rx, int &ry)
{
        int temp;
        cout << "Swap. Before swap, rx: " << rx << " ry: " << ry << "\n";
        temp = rx;
        rx = ry;
        ry = temp;
        cout << "Swap. After swap, rx: " << rx << " ry: " << ry << "\n";
}
```

```
Output: Main. Before swap, x:5 y: 10
Swap. Before swap, rx:5 ry:10
Swap. After swap, rx:10 ry:5
Main. After swap, x:10, y:5
```

The Inline function (request)

The in-line function is very important feature not found in C. It is a function that is expanded at the point at which it is called instead of actually being called.

The reason of inline function is efficiency. Every time a function is called, a series of instructions must be executed to set up the function call (pushing arguments onto stack, returning from function). In some cases, many CPU cycles are used to perform these procedures. But, when a

function is expanded in line, no overhead exists and the program speed increases.
If the inline function is large, the program size increase. So, the best inline functions are those that are very small.
i.e. it put the function itself in the program when compiling instead of calling it in the runtime.
The inline is a request not a command to compiler to generate inline code. There are various situations that can prevent the compiler from compiling the request. The following situation won't be generated if the function contain:
Loop, Switch, or goto
If the function not returning a value, and a return statement exists
If it is recursive function
If the function contain static variable

To declare a function inline, just write "inline" before the return type of function

```
inline ret_type FuncName(parameter_list)
{

}
```

<u>Function Overloading:</u>
In C++, two or more functions can share the same name as long as their parameter declarations are different. In this situation, the functions that share the same name are said to be *overloaded.*
<u>Example:</u>

```
#include <iostream.h>
int sqr_it(int I);
double sqr_it(double d);
long sqr_it(long l);

void main()
{
        cout<<sqr_it(10)<<"\n";
        cout<<sqr_it(11.0)<<"\n";
        cout<<sqr_it(9L)<<"\n";
}

int sqr_it(int I)
{
      cout<<"Inside the sqr_it() function that uses an integer argument \n";
      return I * I;
}
double sqr_it(double d)
{
      cout<<”Inside the sqr_it() function that uses a double argument \n”;
      return d * d;
}
long sqr_it(long l)
{
      cout<<”Inside the sqr_it() function that uses a long argument \n”;
      return l * l;
}
```

Example 1: Write a program that work with imaginary number (x+yi)

The imaginary number is composed from two parts a real part and imaginary part

```
struct Complex
{
	float real;
	float img;
};
Complex AddComplex(Complex C1, Complex C2);
Complex SubComplex(Complex C1, Complex C2);
void PrintComplex(Complex C);
Complex AddComplex(Complex C1, Complex C2)
{
	Complex temp;
	temp.real = C1.real + C2.real;
	temp.img = C1.img + C2.img;
	return temp;
}
Complex SubComplex(Complex C1, Complex C2)
{
	Complex temp;
	temp.real = C1.real - C2.real;
	temp.img = C1.img - C2.img;
	return temp;
}
void PrintComplex(Complex C)
{
	cout<<”Real = “<<C.real;
	cout<<Imaginary = “<<C.img;
}
int main()
{
	Complex cpl1, cpl2, cpl3;
	cpl1.real = 10.2;
	cpl1.img = 5.3;
	cpl2.real = 8.1;
	cpl2.img = 3.6;
	cpl3 = AddComplex(cpl1, cpl2);
	PrintComplex(cpl3);
	return 0;
}
```

Till Here, we can assign a value to a real and to the imaginary directly. So, we are away from the Encapsulation.
We will make a function named SetComplex() to set the value of the complex number:

```
void SetComplex(Complex *pC, float r, float i)
{
	pC->real = r;
	pC->img = i;
}
```

and the main function will be modified to:

```
int main()
{
	Complex cpl1, cpl2, cpl3;
	SetComplex(&cpl1, 10.2, 5.3);
	SetComplex(&cpl2, 8.1, 3.6);
	cpl3 = AddComplex(cpl1, cpl2);
	PrintComplex(cpl3);
	return 0;
}
```

In the last example, we found that the first element of function was a variable to structure or a pointer to structure. All these functions can’t work independently from the structure and the structure can’t work independently from the functions. So, we can put all these function in the structure and don’t use the first element in the function.
The example will be:

```
Struct Complex
{
	float real;
	float img;
	void SetComplex(float r, float i);
	Complex AddComplex(Complex C);
	Complex SubComplex(Complex C);
	void PrintComplex();
};
int main()
{
	Complex cpl1, cpl2, cpl3;
	cpl1.SetComplex(10.2, 5.3);
	cpl2.SetComplex(8.1, 3.6);
	cpl3 = cpl1.AddComplex(cpl2);
	cpl3.PrintComplex();
	return 0;
}
```

```
void Complex::SetComplex(float r, float i)
{
        real = r;
        img = i;
}
Complex Complex::AddComplex(Complex C)
{
        Complex temp;
        temp.real = real + C.real;
        temp.img = img + C.img;
        return temp;
}
Complex Complex::SubComplex(Complex C)
{
        Complex temp;
        temp.real = real - C.real;
        temp.img = img - C.img;
        return temp;
}
void Complex::PrintComplex()
{
        cout<<"\n Real = "<<real;
        cout<<"\n Imaginary = "<<img;
}
```

Even we had make SetComplex() function that set the real and imaginary part of a complex number, we can still access the member data directly. i.e. we can write the following statement in the main() function after declaring the variable of the structure:

```
int main()
{
        Complex cpl1
        cpl1.real = 3.2;
        Return 0;
}
```

To eliminate the accessibility of the member data to the class user, use the keyword *private* and we can also change the keyword struct to the keyword class.

Definition:

private keyword: mean that all the members that are declared after this keyword will be private. This mean that it can be accessed only within methods of the class itself.

Public keyword: mean that all the members that are declared after it will be public. This means that it can be accessed through any object of the class.
The difference between the struct keyword and the public key word is:
By default, all members of struct are public, while all members of class are private by default.

Then the program will be:

```
class Complex
{
        private:
                float real;
                float img;
        public:
                void SetComplex(float r, float i);
                Complex AddComplex(Complex C);
                Complex SubComplex(Complex C);
                void PrintComplex();
};
int main()
{
        Complex cpl1, cpl2, cpl3;
        cpl1.SetComplex(10.2, 5.3);
        cpl2.SetComplex(8.1, 3.6);
        cpl3 = cpl1.AddComplex(cpl2);
        cpl3.PrintComplex();
        return 0;
}
```

then the definition of the member function(method) of the Complex class.

Example 2: Write a class that deal with the stack.
To make a Stack class, we must decide what we need to work with stack.
We need: An array
An indicator
Push function
Pop function

```
class Stack
{
        private:
                int st[10];
                int tos;
        public:
                void push(int n);
                int    pop();
};
void Stack::push(int n)
{
        if (tos==10)
        {
                cout<<"stack is full";
        }
        else
        {
                st[tos] = n;
                tos++;
        }
}
int Stack::pop()
{
        int retval = 0;
        if(tos == 0)
        {
                cout<<"\n Stack is empty";
        }
        else
        {
                tos--;
                retval = st[tos];
        }
        return retval;
}
```

```
int main()
{
        Stack S1;
        Stack S2;
        S1.push(10);
        S1.push(3);
        S1.push(15);
        S2.push(11);
        S2.push(20);
        cout<<S1.pop();
        cout<<S2.pop();
        return 0;

}
```

This program has a small bug, each time I declare an object from this class, I must initialize the tos to zero.
So, I will make a method named initstack() and I will call it each time I declare an object from this class. So, the class will have the following view:

```
class Stack
{
        private:
                int st[10];
                int tos;
        public:
                void initstack()
                {tos = 0;}
                void push(int n);
                int    pop();
};
```

The way we declare the initstack is the second way to declare an Inline function

The main() will be:

```
int main()
{
        Stack S1;
        Stack S2;
        S1.initstack();
        S2.initstack();
        S1.push(10);
        S1.push(3);
        S1.push(15);
        S2.push(11);
        S2.push(20);
        cout<<S1.pop();
        cout<<S2.pop();
        return 0;
}
```

In this example, we declared two object S1, S2 and when we call the initstack() function, it will set the tos with zero.

<u>How the compiler will know that this tos belong to that object?</u>

When calling the initstack(), this is done through an object (ex: S1.initstack();) an implicit parameter containing a pointer to the object created is sent to the function this parameter is called (***this pointer)***

The ***this pointer*** :

Every class member function has a hidden parameter: the `this` pointer which points to the individual object that is used to make the call. Therefore, in each call to `a member function (method)`, the `this` pointer for the object is included as a hidden parameter.

The last example is:

```
#include <iostream.h>
class Stack
{
        private:
                int st[10];
                int tos;
        public:
                void initstack()
                {tos = 0;}
                void push(int n);
                int   pop();
};

int main()
{
        Stack S1;
        Stack S2;          The this pointer is sent implicitly
        S1.initstack();
        S2.initstack();
        S1.push(10);         The this pointer is sent implicitly
        S1.push(3);
        S1.push(  15 );
        S2.push(11);
        S2.push(20);
        cout<<S1.pop( );  ← The this pointer is sent implicitly
        cout<<S2.pop();
        return 0;
}
void Stack::push(int n)
{
        if (this->tos==10)
        {
                cout<<"stack is full";
        }
        else
        {
                this->st[tos] = n;
                this->tos++;
        }
}
int Stack::pop()
{
        int retval = 0;
        if(this->tos == 0)
        {
                cout<<"\n Stack is empty";
        }
        else
        {
                this->tos--;
                retval = this->st[tos];
        }
        return retval;
}
```

The this is send automatically, and contain the object I am using. In the method declaration, we needn't to write the this pointer. But if we write it, it is not wrong.

Each time I create an object from the class, I must call the InitStack once to initialize the tos with zero.

In the C++ Language, when working with classes, there is a function called automatically at the creation of any object. This function is called (**The Constructor)**.

The Constructor:

Is a function called automatically when creating the object and must have the following properties:

- Has no return type not even void
- Has the name of the class
- Must be public member function
- Must not be virtual, static or const

```
class Stack
{
        private:
                int st[10];
                int tos;
        public:
                Stack()    //Constructor
                {tos = 0;}
                void push(int n);
                int    pop();
};
```

I may have more than one constructor; they differ only in the argument. When creating object, it use only one of them. The Constructor that has no argument is called: "The Default Constructor"

<u>Example:</u> If we want that the array st[] to be dynamic that the class user will enter it OR if he didn't enter anything it will be array of 10
the class will be as follow:

```
class Stack
{
        private:
                int *st;
                int tos, size;
        public:
                Stack()
                        {
                          tos 0;
                          size = 10;
                          st = new int[size];
                        }

                Stack(int s)
                        {
                          tos 0;
                          size = s;
                          st = new int[size];
                        }
                void push(int n);
        int pop();
};
void Stack::push(int n)
{
    if(tos ==size)
    {
        cout<<"stack is full";
    }
    else
    {
        st[tos] = n;
        tos++;
    }
}
int Stack::pop()
{
    int retval = 0;
    if(tos == 0)
    {
        cout<<"\n Stack is empty";
    }
    else
    {
        tos--;
        retval = st[tos];
    }
    return retval;
}
```

Using the class:

```
Stack S1;        //this will call the default constructor i.e. size = 10
Stack S2(100);  //this call the other constructor that have one parameter
                //(size=100)
```

Before erasing the object, we must free the space that we have allocated So, we need a function that is called before erasing the object immediately.

There is another function that is called before erasing the object from memory it is called (**The Destructor)**

<u>The Destructor:</u>

It is a function called automatically just before erasing the object from the memory it has the following:

- Has no return type
- Has the name of the class proceeding with ~

- Has no parameter
- Must be public
- There is only one destructor

<u>To make the destructor for the Stack class</u>

```
~Stack()
{
        delete st;
}
```

<u>Let us make a program that deal with the complex number</u>

```
class Complex
{
		double real, img;
	public:
		Complex(double r = 0, double I = 0)
		{
			real = r;
			img = I;
		}
		void SetReal(double d) {real = d;}
		void SetImg(double d) {img = d;}
		double GetReal() {return real;}
		double GetImg() {return img;}
		Complex AddComplex(Complex C);
		Complex SubComplex(Complex C);
};

Complex Complex::AddComplex(Complex C)
{
	Complex temp;
	temp.real = real + C.real;
	temp.img = img + C.img;
	return temp;
}
Complex Complex::SubComplex(Complex C)
{
	Complex temp;
	temp.real = real - C.real;
	temp.img = img - C.img;
	return temp;
}

void main()
{
	Complex C1, C2, C3, C4;
	Complex temp;
	temp = C2.AddComplex(C3);
	C1 = temp.AddComplex(C4);
OR
	C1 = C2.AddComplex(C3.AddComplex(C4));
}
```

This is very difficult way. To make this formula easier, use the operator overloading.

Operator Overloading:

To make operator overloading, the operator must exist and we extend its functionality. When using the operator overloading, we must keep its characteristics.

In the last example, instead of using AddComplex() function, we can overload the + operator and instead of using SubComplex() function, we can overload the – operator as follow:

Complex AddComplex(Complex C) ⇨ Complex operator +(Complex C)
Complex SubComplex(Complex C) ⇨ Complex operator –(Complex C)

- If we want to add an integer number to a complex number:

Ex: C3 = C1 + 4

We must make another overloading of the +operator as follow:

In class definition:

```
        Complex operator + (int x);
```

In class implementation:

```
        Complex Complex::operator + (int x)
        {
                Complex temp;
                temp.real = real + x;
                temp.img = img;
                return temp;
        }
```

- To overload the == operator

In class definition:

```
        int operator ==(Complex C);
```

In class implementation:

```
        int Complex::operator==(Complex C)
{
        if((real==C.real)&&(img==C.img))
        {
                return 1;
        }
        return 0;
}
```

- Overloading the = operator

C1 = C2, is valid but it makes a bitwise copy.

The overloading of (=) operator must have a return value to respect the nature of the assignment operator which is multi assignment.

```
Complex& Complex::operator=(Complex C)
{
	real = C.real;
	img = C.img;
	return *this;
}
```

The use of reference is good in performance wise.

<u>Example:</u>

We have added an integer to the complex which has the following syntax:

```
C1 = C2 + 3;
```

But, if we want to add an integer to the complex with the following syntax:

```
C1 = 3 + C2;
```

The first element before the (+) operator (which will be hold in the this) is integer number. So, we can’t overload the (+) operator as a member function.

```
Complex operator + (int n, Complex C)
{
	Complex temp;
	temp.SetReal(n + C.GetReal());
	temp.SetImg(C.GetImg());
	return temp;
}
```

This is very difficult to write, it will be easiest if there is a way that we can write it as follow:

```
temp.real = n + C.real;
temp.img = C.img;
```

Making it as a friend function can do this.

<u>Friend function:</u>

It is a non member function, but has a right to access the private member of the class.

<u>Note:</u>

The friend function violates the concept of encapsulation of C++. So, it is not recommended to use it.

All we have to do, is to define the friend function in the class definition preceded with the keyword friend.

```
friend Complex operator + (int n, Complex C);
```

Class Variable (Static Members)

The class variable is a member variable that is defined at the class level. It is create only once in the memory, regardless the number of object created from that class (i.e. it is shared between all classes).

- It is defined as a class member, while it is initialized after the class definition.
- To deal with the static variable(s) we use static function(s), which are also class methods.
- Since it is shared for all the objects created from that class, and there is only one copy of it in the memory. We don't need to call it using an object (because it is not related to any object). So we call it using the class name.
- Because it is called using the class name not an object, it doesn't have the "this" pointer.

Example:

Let's change the Complex class in order to notify the user with a message when number of objects in the memory exceeds the 5 objects.

Solution

```
class Complex
{
                float real, img;
                static int count;
        public:
                Complex(int n = 0)
                {
                        if(count > 5)
                        {
                                cout<<"Number of object created is : << count;
                        }
                        real = img = n;
                        count ++;
                }
                Complex(int m, int n)
                {
                        if(count > 5)
                        {
                                cout<<"Number of object created is : << count;
                        }
                        real = m;
                        img = n;
                        count ++;
                }
                ~Complex()
                {
                        count--;
                }
                static int GetCount()
                {
                        return count;
                }
};
int Complex :: count = 0;

int main()
{
        Complex cpl1; cpl2;
        .
        .
        cout<< Complex::GetCount();
        return 0;
}
```

Creating Array of Objects:

```
class Complex
{
		float real, img;
	public:
		Complex(int n = 0)
		{
			real = img = n;
		}
		Complex(int m, int n)
		{
			real = m;
			img = n;
		}
};
```

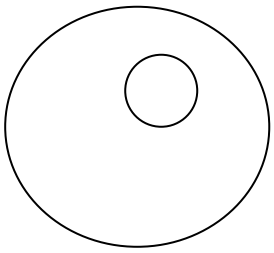
Complex ar[5]; /* this will create an array of 5 objects(call the constructor 5 times) and it will use the Default Constructor */
Complex ar[5] = {3.2, 4.7, 9.11, 2.1, 8.0}; /* this will create an array of 5 objects(call the constructor 5 times) and it will use the Constructor that take one float value as argument */
Complex ar[5] = {Complex(7.1, 3.4), Complex (2.9, 5.2), …}; /* this will create an array of 5 objects(call the constructor 5 times) and it will use the Constructor suitable to the argument between {} */

Complex *pC;
pC = new Complex(7.2, 3.9); /*this will create one object using dynamic allocation using the constructor that take 2 arguments*/
pC = new Complex[10]; /* this will create an array of objects using dynamic allocation . this is done by the Default Constructor (Must Have Default constructor) */
delete []pC; /*delete all object(s) (call the destructor for all objects) before de-allocating the pointer */

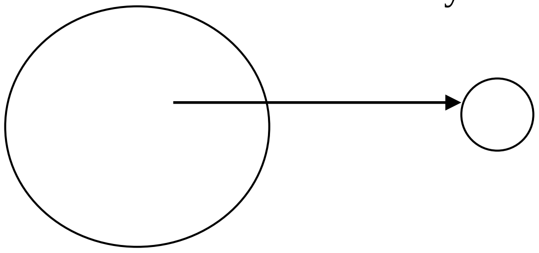
Relations between classes

There are three types of relations between classes:

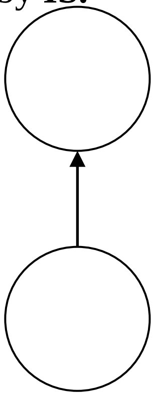
1. **Aggregation:**
   In which a class have an object of another class as a member variable in it. This is named as “Embeded object”, and is defined by the relation **HAS.**



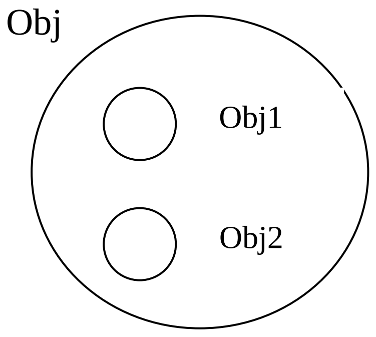
2. **Association:**
   In which a class have a pointer as a member variable that points to object of another class. This relation is defined by **KNOWS.**



3. **Inheritance:**
   In which a class inherits some or all members of another class. This relation is defined by **IS.**



First: Aggregation



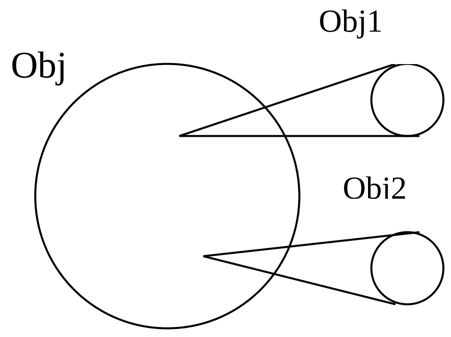
Here, the object Obj1 and Obj2 must be created before creating Obj. and this is known using the following statement:
Obj HAS obj1, obj2

Obj1, Obj2 can be member of obj only if:
*) Obj doesn't have a constructor
OR
*) Obj has a default constructor
OR
*)Obj constructor specifies an initialization for that member

this is called LAYERED CLASS
Note:
Obj can access public member of obj1, obj2

```
Class Point
{
  public:
        int x;
        int y;
        Point(int l, int m)
        {
                x=l;
                y=m;
        }
};
```

Second: Association



Here, we create the object Obj. and when need, (at run time)
This is known using the following statement
Obj KNOWS obj1, obj2

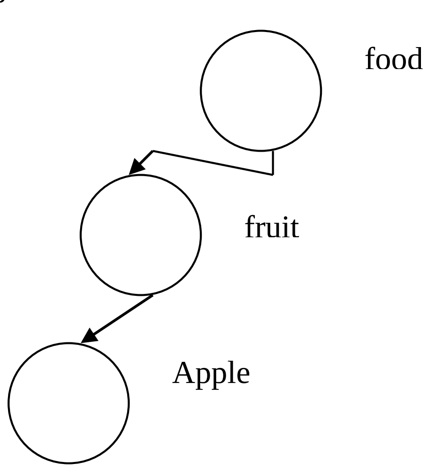
This is more flexible
Obj1, obj2 may, or may not be accessible

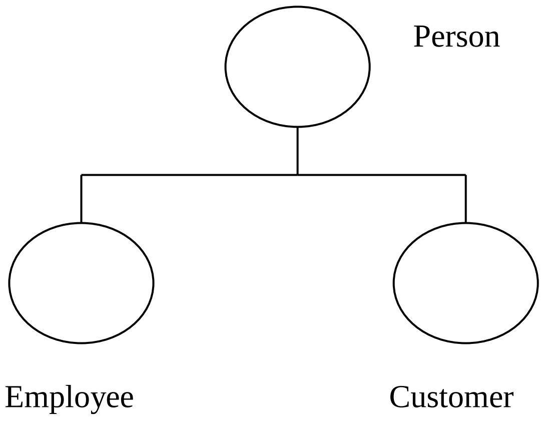
```
class Rectangle
{
        private:
                int length, width;
                Point P1, P2;
        public:
        Rectangle(int a, int b, int c,
int d):P1(a,b), P2(c,d)
{
        length = P2.x – P1.x;
        width = P2.y – P1.y;
}
};
```

```
class Rectangle
{
        private:
                int length, width;
                Point *P1, *P2;
        public:
                Rectangle(Point *pa,
Point *pb)
{
        P1 = pa;
        P2 = pb;
        length = P2->x – P1->x;
        width = P2->y – P1->y;
}
};
```

Inheritance

Inheritance is supported by allowing one class to inherit the properties of another object.





Apple IS A Fruit
Fruit IS A Food

<u>Example:</u>

```
class Base
{
        protected:
                int A;
        public:
                Base(){A = 0;}
                Base(int n){A = n;}
                void SetA(int n){A = n;}
                int GetA(){return A;}
};

class Derived : public Base
{
        protected:
                int B;
        public:
                Derived(){B = 0;}
                Derived(int n) : Base(n)
                {B = n;}
                Derived(int n, int m) : Base(n)
                {B = m;}
                void SetB(int n){B = n;}
                int GetB(){return B;}
                int Product()
```

```
                    {return B * GetA();}
};                                    because A is private member(protected)
```

before creating the derived object, it creates the base class.

```
void main()
{
          Derived obj;
          obj.SetA(10);
          obj.SetB(5);
          Base obj2;
          int p = obj.GetA() * obj.GetB();
OR
          int p = obj.Product();
          Derived obj3(5);
          Derived obj4(10,5);
}
```

<u>member can be:</u>
public: accessed by all class user
private: accessed by member function and friend function of the class
protected: accessed by the member of derivate class.

```
class Derived1 : public Derived
{
                    int c;
          public:
                    Derived1(){c=0;}
                    Derived1(int n):Derived(n){c = n;}
                    Derived1(int n, int m):Derived(n){c = m;}
                    Derived1(int l, int m, int n):Derived(l,m){c = n;}
                    void SetC(int n){c = n;}
                    int GetC(){return c;}
                    int product() //this is called function overriding
                    {
                              return (c*b*a);
                    }                             must be protected
};

void main()
{
          Derived1 ob;
          cout<<ob.product(); //derived1
          cout<<ob.Derived::product(); //derived
}
```

Now, let’s have the following example:

```
class Base
{
        public:
                int x, y;
                Base()
                {
                        x = y = 0;
                }
                Base(int l, int m)
                {
                        x = l;
                        y = m;
                }
                int Product()
                {
                        return x * y;
                }
};

class Derived1 : public Base
{
        public:
                int a;
                Derived1()
                {
                        a = 0;
                }
                Derived1(int l, int m, int n) : Base(m,n)
                {
                        a = l;
                }
                int Product()
                {
                        return x * y * a;
                }
};
```

```
class Derived2 : public Derived1
{
        public:
                int b;
                Derived2()
                {
                        b = 0;
                }
                Derived2(int l, int m, int n, int o) : Derived1(m, n, o)
                {
                        b = l;
                }
                int Product()
                {
                        return x * y * a * b;
                }
};
```

When creating an object from the Derived2:

```
Derived2 obj(10, 20, 30, 40);
.
.
.
obj.Product();          //this will call the last one (x * y * a * b)
        To call another one, use the scope operator(::)
obj.Derived1::Product();  //this will call the one of Derived1
obj.Base::Product();        //this will call the one of the Base

Derived2 *p2;
p2 = &obj;
p2->Product();        //this will call the last one (x * y * a * b)
p2->Derived1::Product();//this will call the one of Derived1
p2->Base::Product();       //this will call the one of the Base
```

Here, we have another rule:
If we have a pointer or a reference to the Base class, we can assign an object of the derived class to it. But the pointer to the Base class can access only the member of the Base class.

i.e.:

```
Derived1 *p1;
p1 = &obj;            //this statement is true
p1 -> Product();     //this will call the function of Derived1
```

```
p1 -> Base::Product();

Base *p;
p = &obj;
p->Product();        //this will call the one of the Base
```

Note:
If we have a pointer to the base class, we can't access the member of the derived class unless we use the virtual function.

Virtual function:
Is a function that is declared as virtual in a base class and redefined in one or more derived classes. However, when a virtual function is redefined by a derived class, the keyword virtual need not to be repeated (although it is not an error to do so)

In the last example, if we need the pointer p or p1 to access the Product() function of Derived2, we must make it virtual function. By just adding the virtual keyword before the function Product() return type in the class Base, and class Derived1.
The virtual function in derived class required to be accessed by pointer to base class.
This offer:

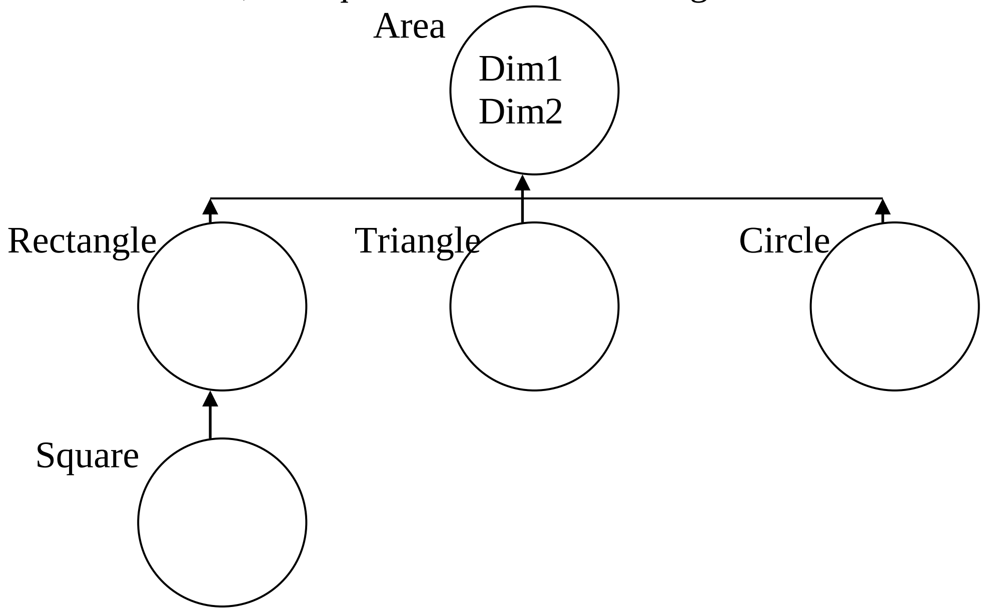
1) Class hierarchy open ended
2) Dynamic binding

Note:
The pointer to base class is used with virtual function.

Example:

Write a program to calculate the area of circle, rectangle, square and triangle.

Any area is calculated using the following equation:
Area = const. * Dim1 * Dim2
And the different will be in the constant. So we can make a generic class Area that has Dim1 and Dim2 as member variables and classes circle, rectangle, and triangle that inherit it. also, the square is a kind of rectangle so it will inherit it as follow:



Solution

```
class Area
{
	protected:
		int dim1;
		int dim2;
	public:
		Area(int x)
		{
			dim1 = dim2 = x;
		}
		Area(int x, int y)
		{
			dim1 = x;
			dim2 = y;
		}
```

```
                void GetDim(int &d1, int &d2)
                {
                        d1 = dim1;
                        d2 = dim2;
                }
};

class Triangle : public Area
{
        public:
                Triangle(int h, int b) : Area(h,b)
                {}
                float CalcArea()
                {
                        return 0.5 * dim1 * dim2;
                }
};

class Circle : public Area
{
        public:
                Circle(int r) : Area(r)
                {}
                float CalcArea()
                {
                        return dim1 * dim2 * 22 / 7;
                }
};

class Rectangle : public Area
{
        public:
                Rectangle(int l, int b) : Area(l,b)
                {}
                float CalcArea()
                {
                        return dim1 * dim2;
                }
};
```

```
class Square : public Rectangle
{
        public:
                Square(int l) : Rectangle(l,l)
                {}
};
```

Write a class that calculate the area of different shape (Rectangle, Circle, Triangle and Square)

```
class GeoShape
{
                Circle          *pCr;
                Rectangle       *pRect;
                Square          *pSqr;
                Triangle        *pTr;
    public:
        GeoShape(Circle *p1, Rectangle *p2, Square *p3, Triangle *p4)
        {
                pCr    = p1;
                pRect = p2;
                pSqr   = p3;
                pTr    = p4;
        }
        float CalcArea()
        {
                return (pCr->CalcArea() + pRect->CalcArea() +
                        pSqr->CalcArea() + pTr->CalcArea());
        }
};

void main()
{
        Circle C(10);
        Rectangle R(5,2);
        Square S(7);
        Triangle T(3,12);

        GeoShape obj(&C, &R, &S, &T);
        cout<<”Total Area = “<<obj.CalcArea();
}
```

In the last example of area, we must calculate the area of triangle, circle, rectangle and square. We can’t change any of them.
So, we can do the following:
*) We will add a function named CalcArea() in the area class. It has no importance more than it let us use the virtual function in derived class:
*) Add the keyword “virtual” before the CalcArea() function as follow:

```
	virtual float CalcArea()
	{
		return 0;
	}
```

*) Change the GeoShape class to the following:

```
class GeoShape
{
		Area		*pShape[4];
	public:
		GeoShape(Area *p1, Area *p2, Area *p3, Area *p4)
		{
			pShape[0] = p1;
			pShape[1] = p2;
			pShape[2] = p3;
			pShape[3] = p4;
		}
		float CalcArea()
		{
			int Ar = 0;
			for(int I = 0 ; I < 4 ; I++)
			{
				Ar += pShape[i]->CalcArea();
			}
			return Ar;
		}
};
```

The function CalcArea() in the derived classes Triangle, Rectangle, Circle may not have the keyword virtual. But, we write it in case it will be inherited(i.e. open ended hierarchy)

The main function will be:

```
void main()
{
	Circle C1(10), C2(5);
	Rectangle R(10, 20);
	Square S(17);
```

```
        GeoShape obj(&C1, &C2, &C3, &C4);
        cout<<obj.CalcArea();
}
```
Here, we calculated the sum of 4 area but their types are defined at runtime.

The Abstract class

It is a class that we can't create object from it (is not used alone). Its presence to make another class that inherits it.
Example: in the last example, the class Area() we don't create any object from it and we won't.

The class to be abstract, it must contain at least one pure virtual function, its syntax:
virtual float CalcArea() = 0;
and there is no definition.

Inherited class of abstract class ***MUST*** contain the implementation of the pure virtual with the same name and argument, otherwise, it will be considered as an abstract class.

Multiple Inheritance

It is possible for one class to inherit the attributes of two or more classes. To accomplish this, use a comma separated

Example:

```
class Base1
{
        protected:
                int x;
        public:
                Base1(int n=0)
                {
                        x = n;
                }
                void SetX(int n)
                {
                        x = n;
                }
                int GetX()
                {
                        return x;
                }
};

class Base2
{
        protected:
                int y;
        public:
                Base1(int n=0)
                {
                        y = n;
                }
                void SetY(int n)
                {
                        y = n;
                }
                int GetY()
                {
                        return y;
                }
};
```

```
class Derived : public Base1, public Base2
{
        public:
                Derived(int a. int b):Base1(a), Base2(b)
                {}
                int Sum()
                {
                        return x+y;
                }
                int Product()
                {
                        return x*y;
                }
};

void main()
{
        Derived obj(5, 6);
        Obj.SetX = 10; //access base1
        Obj.SetY = 20; //access base2
        cout<<"Sum = "<<Obj.Sum();
        Derived obj2(10,2);
        cout<<obj2.Sum();
}
```

Note:
The two parent classes are called in the way they are written in the line of declaring the derived class. The way of calling the constructor of the derived class has no effect on the constructing of the parent.

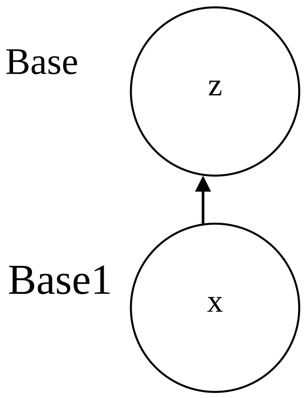
Now, let's have the following example:

```
class Base
{
        protected:
                int z;
};
class Base1:public Base
{
        protected:
                int x;
};
```

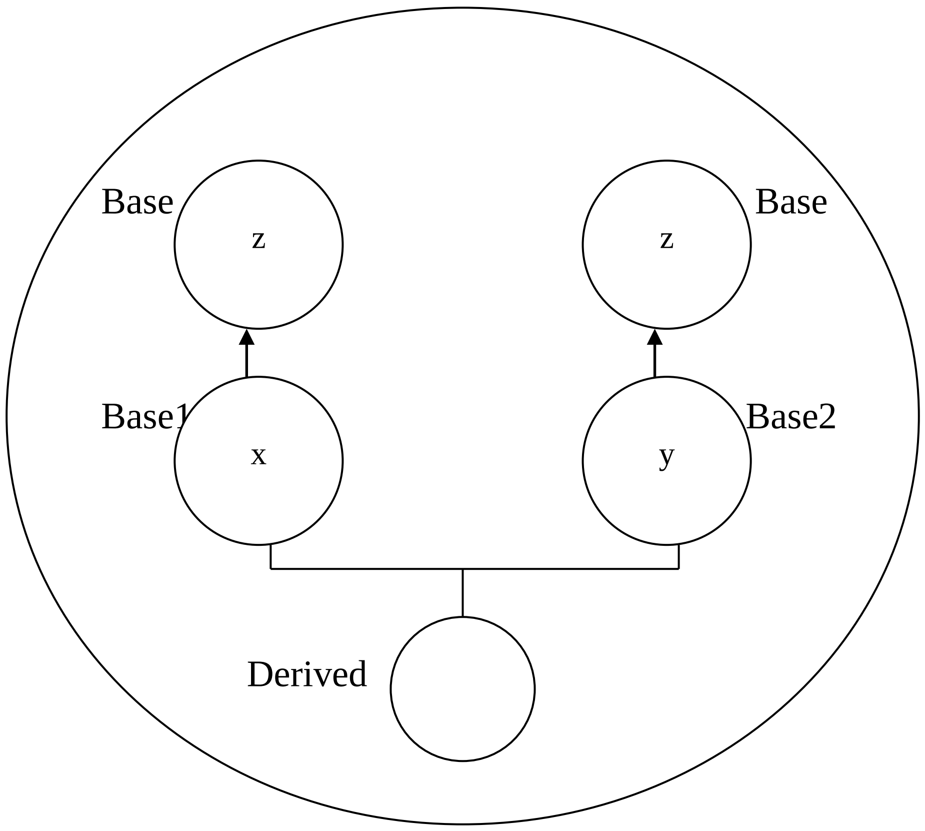
```
class Base2:public Base
{
        protected:
                int y;
};

class Derived: public Base1, public Base2
{
        int Sum()
        {
                return x+y+z;
        }
};
```

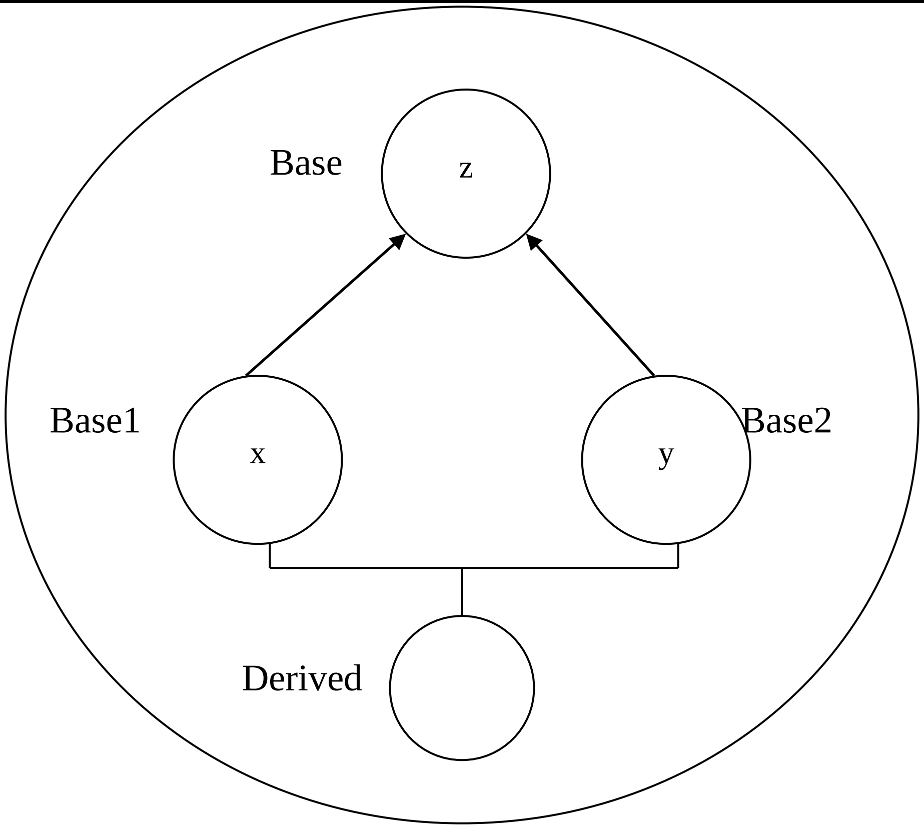
Here, we will have a problem when creating an object from the derived class:
At the beginning, when creating an object from the class Derived, it will call the constructor of the Derived. But, Before executing the Constructor, it will found that the Derived class inherit the Base1 class. So, it calls its constructor first, but it found that it inherit the class Base. It constructs it then constructs Base1 then return to construct Derived.



And before it construct the Derived. It found that it inherits Base2 which inherits the class Base.
So it constructs the class Base, then the class Base2 then the class Derived. And we will have the following figure:

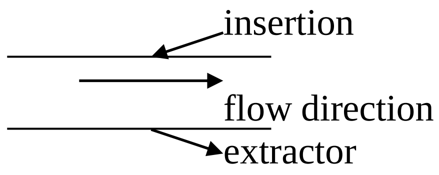


And the object from the Derived class will contain 2 instance from the class Base, 1 instance from the class Base1, and one instance from the class Base2.
If we try to access the member z, there will be an error. Because, there are two z in the Derived object.
To solve this problem, the Base1 and Base2 classes must inherit the Base class as virtual class, this do the following: before constructing an instance of a class in an object, it search if there is already an instance or not. If there is an instance in the object, it don't create a new one. It, there is no instance, it create it. and the figure will be:

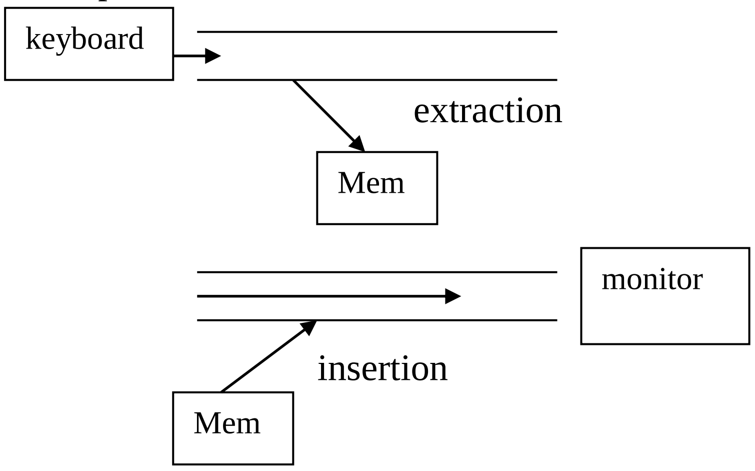

Base
z
Base1
x
y
Base2
Derived

Streams

Stream is the way for input and output.
The data has a flow, so we can insert in this flow or extract from this flow.



This is handled through the classes istream, ostream and iostream.
The C++ contain several predefined streams that are opened automatically when the C++ program begins to execute. They are cin is the stream associated with the standard input, and cout is the stream associated with the standard output.



There are also two operator:
<< insrtion operator because it inserts characters into stream (cout)
>>extraction operator because it extracts character from stream (cin)
The insertion operator and extraction operator are overloaded in the iostream.h to perform stream i/o on any C++ built in type.

C++ allows a better way of performing I/O operations on classes by overloading the << and >>

```
class three_d
{
        int x, y, z;
    public:
        three_d(int a, int b, int c)
        {
            x = a;
            y = b;
            z = c;
        }
```

```
                friend ostream& operator<<(ostream &out, three_d obj)
                {
                        out<<obj.x<<”,”;
                        out<<obj.y<<”,”;
                        out<<obj.z<<”\n”;
                        return out;
                }
                friend istream& operator>>(istream &in, three_d &obj)
                {
                        cout<<”Enter x, y, z values : “;
                        in>>obj.x>>obj.y>>obj.z;
                        return in;
                }
};

void main(void)
{
        three_d a(1, 2, 3), b(3, 4, 5), c(5, 6, 7);
        cout<<a<<b<<c;
        cin>>a;
}
```