http://www.cprogramming.com/tutorial/cpreprocessor.html

http://msdn.microsoft.com/en-us/library/3sxhs2ty%28v=vs.80%29.aspx

Preprocessor tasks:

These tasks are carried out before handing over the control to the compiler. Typically there are three types of Preprocessor directives: Directives, Constants, Conditions, Macros. The preprocessor directives always start with a "#" symbol.

Best example for directive would be #include. You include the source code of another file along with your code. #define is used to define constants in the program. So before compilation where ever the constant name comes it is replaced by its value that the programmer has specified using #define.

Example,

```
#define PI 3.142
void main()
{
        int a = 2;
        int b = a*PI;
}
```

Same as →

```
#define PI 3.142
void main()
{
        int a = 2;
        int b = a*3.142;
}
```

The next useful trick is specifying conditions using : #if,#ifdef,#ifndef,#elif,#endif,#else. All these conditional compilation directives must end with #endif. #if evaluvates the expression next to it and if it's true then it includes the block next to in the source else it doesn't. Remember it is only a compile time condition not a run time condition. To illustrate how this works lets look at the following code

```
int a = 2;
#if 2
        printf("HI");

#endif
```

Will print HI

Not same as →

```
int a = 2;
#if a
        printf("HI");

#endif
```

This won't print HI

Similarly #ifdef is used to check if the constant is defined or not. If its defined the subsequent set of statements are included. #ifndef performs the opposite of #ifdef.

Macros are like functions. They are extremely powerful. Why? Calling time for a function is avoided as the macro code is replaced at the location where call is made. Macros are also defined using #define. For example,

```
#define ADD(a,b) a+b
void main()
{
```

```
        int a = ADD(2,3);
        printf("%d",a);
}
```

This will print 5.

ADD(2,3) would be replaced by 2+3.

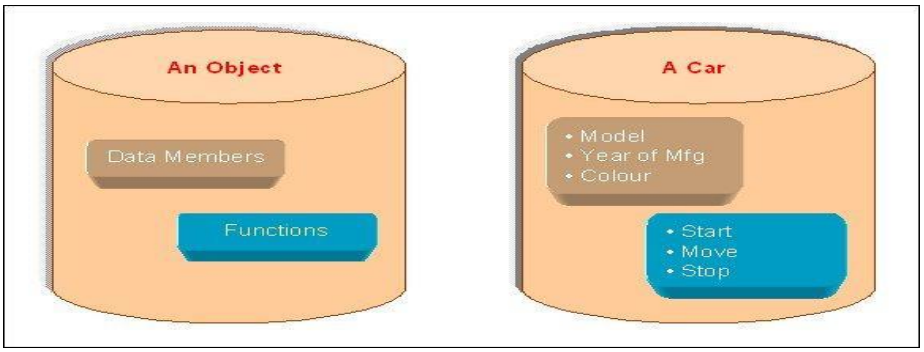
C++:

The prime purpose of C++ programming was to add object orientation to the C programming language, which is in itself one of the most powerful programming languages.

The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see whole world in the form of objects. For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

Objects:

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of a class. Each instance of a class can hold its own relevant data.



An Object is a collection of data members and associated member functions also known as methods.

Classes:

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects. Characteristics of an object are represented in a class as Properties. The actions that can be performed by objects become functions of the class and are referred to as Methods.

For example consider we have a Class of Cars under which Santro Xing, Alto and WaganR represents individual Objects. In this context each Car Object will have its own, Model, Year of Manufacture,

Color, Top Speed, Engine Power etc., which form Properties of the Car class and the associated actions i.e., object functions like Start, Move, and Stop form the Methods of Car Class.

No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

Inheritance:

Inheritance is the process of forming a new class from an existing class or base class. The base class is also known as parent class or super class. The new class that is formed is called derived class. Derived class is also known as a child class or sub class. Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

Data Abstraction:

Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without presenting the details.

Data Encapsulation:

Data Encapsulation combines data and functions into a single unit called class. When using Data Encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class. Data Encapsulation enables the important concept of data hiding possible.

Polymorphism:

Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways.

Overloading:

Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an existing operator or function begins to operate on new data type, or class, it is understood to be overloaded.

Reusability:

This term refers to the ability for multiple programmers to use the same written and debugged existing class of data. This is a time saving device and adds code efficiency to the language. Additionally, the programmer can incorporate new features to the existing class, further developing the application and allowing users to achieve increased performance. This time saving feature optimizes code, helps in gaining secured applications and facilitates easier maintenance on the application.

Constructors and Destructors:

Constructors are used to initialize components. Destructors are used to destruct these components at the object's death. Why do we use them? To make the object ready and to clean up things when it dies. These are also functions but with no return type. Remember Constructor is always public. Even if the programmer doesn't explicitly specify a constructor, the compiler adds a constructor implicitly. Constructors have the same name as that of the class. There are three types of Constructors:

- Default Constructors : Class-Name(){ }
- Parameterized Constructors: Class-Name(int paramter) { a = parameter; }
- Copy Constructors: Class-Name(Class-Name &object) { //Body of the constructor }

Obviously you are allowed to overload or have more than one type of Constructor thanks to the concept of Function Overloading. So which constructor gets called? That decision is made when the object is created. Consider the following snippet

```
class Cls
{
        Public:
        int a;
        Cls() { a = 2; }                    //First Constructor
        Cls(int param1) { a = param1;}      //Second Constructor
        Cls(Cls obj1) { a = obj1.a; }       //Third Constructor
}
Void main()
{
        Cls obj1; //Calls First constructor
        Cls obj2(2) //Calls Second constructor
        Cls obj3(obj2) // Calls Third Constructor
}
```

Now the house keeping work is done by Destructor. It has the same name as the Class expect that a tilde (~) sign precedes the class name. When the object goes out of scope this function is called automatically and all the cleanup work is performed. As usual, if the programmer doesn't specify a destructor then a destructor is added implicitly.

```
Class Cls
{       Int a;
        ~cls() {  Printf("Destructing");}    }
```

Inheritance:

Inheritance is something like taking characteristics from one entity and using it. Since in the programming world characteristics are defined using Variables and functions, Inheritance can be defined as using the variables and functions of one class in another. The class that initially had the

variables and functions is called the Base class and the class that borrowed it is called as the derived class.
Why is it useful?
1. Code Re-Usability
2. Easier Maintenance

But will the base class just let you use all its characteristics. No!! A golden rule for Inheritance is that Private variables and functions can never be inherited. It won't let its own object use it, let alone some other class. Also the way things are inherited indicate how the characteristics of the base class is present in Derived class. There are three ways of inheriting the characteristics: Public, Private, Protected. The following table summarizes

	Public Characteristics	Protected Characteristics	Private Characteristics
Public Inheritance	Public	Protected	No Access
Protected Inheritance	Protected	Protected	No Access
Private Inheritance	Private	Private	No Access

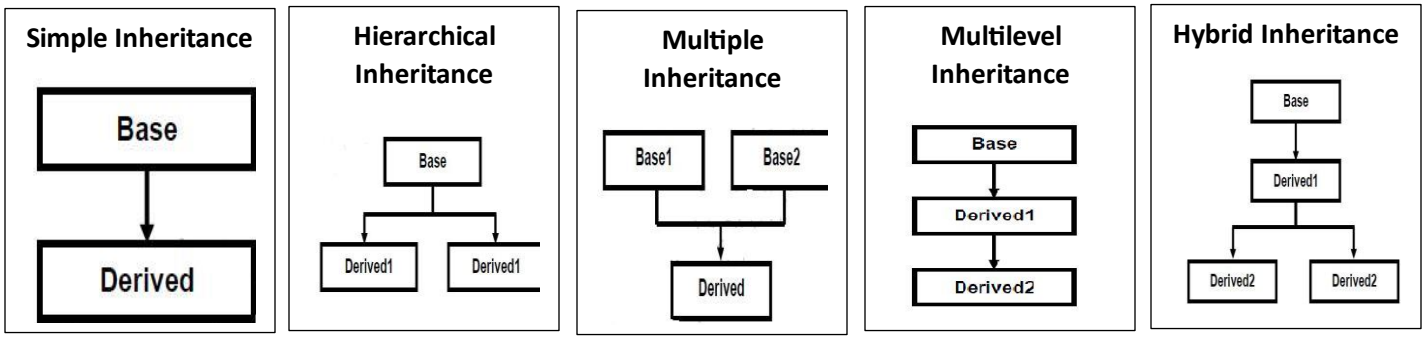
Notice that Public inheritance the access specifier retains. Protected and Private the access in the derived class for all the characteristics is Protected and Private respectively.

Syntax :
Derived Classname : Access specifier Base Class-Name
class Class2 : public Class1, public Class3 { //Class definition}

Consider the following code snippet:

```
class cls1 {        public:   int a;
                    void print1(){
                            printf("Class 1\n");
                    }
};
class cls2: public cls1 {
        public:
        int b;
        void print2()     {
                printf("Class 2\n");
        }
};
Void main() {
        cls2 obj1;
        obj1.print1();
        obj1.print2();                  //This program will print Class 1 and Class 2
}
```

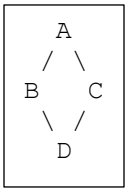
So we have seen that using we can use the same code defined in Class 1 with an object of Class 2.
There are different types of inheritance all highlighted using flow diagrams.



The following rules govern the order in which the calls are made to the constructor and destructor during inheritance: (source : http://stackoverflow.com/questions/7539282/order-of-calling-constructors-destructors-in-inheritance-c)

- Construction always starts from the base class. If there are multiple base classes then, it starts from the left most base. (side note: If there is a virtual inheritance then it's given higher preference).
- Then it comes the turn for member fields. They are initialized in the order they are declared
- At the last the class itself is constructed
- The order of destructor is exactly reverse

Diamond Problem:



Consider the inheritance shown here. In the Derived class D there will be multiple instances of 'A', one through B and the other through C. To avoid this we use the keyword virtual

```
class A { public: void func{} }
class B : public virtual A {}
class C : public virtual A {}
class D : public B, public C {}
```

This is called virtual inheritance. The base class A is called the virtual base class.

Source: http://www.learncpp.com/cpp-tutorial/118-virtual-base-classes/

Virtual Function:

Virtual function is an implementation that can be overridden or used as is in the derived class.

```
class Base
{
public:
   virtual void Print() { printf("Base"); }
};
```

```
class Derived: public Base
{
public:
   virtual void Print() { printf("Derived"); }

};
Void main()
{
        Derived obj1;
        Obj1.print(); //This will print Derived.
}
```

Abstract classes:

Abstract classes are classes that are solely used for the purpose of inheritance. Objects cannot be created for abstract classes. Now a class is an abstract class if it contains atleast one **Pure Virtual function.** A pure virtual function is declared as follows.

Virtual void func() = 0; //Notice the =0 part here. This defines an abstract class.

Now the difference between a normal virtual function and a pure virtual function is that, derived class that derives from an abstract must override the Pure virtual function. Else the derived class will also become abstract class. This is just the application of the rule that defines Abstract classes.

Scope resolution operators:

' :: ' is defined as scope resolution operators. Say you use a local and global variable with the same name 'a'. Now if you use 'a' in your code it refers to local variable. What if you want to refer to the global variable. Scope resolution operator helps you do that. It also helps in defining a member function outside the class.

Friend Classes and functions:

Friend classes and functions are functions and classes defined outside the class but are allowed to access the private members of the Class. The syntax for the friend class is as follows:

```
class A
{
        Private: int a;
        Int b;
        Friend class B;
}
Class B
{
        Public:
        Void func()
        {
                Cout<<b;
        }
```

```
}
```
Friend function takes an object of the class as an argument.
```
Class A
{
        Private:
        int b;
        Public:
        A()
        {
                b= 2;
        }
        Friend void print(A obj);
}
void print(A obj1)
{
        Cout<<obj1.b;
}
```

Inline Functions:

Inline functions are created by specifying the inline keyword. Whenever an inline function is called then the function code is copied on to the location where the function call is made.

```
inline void hello()
{
 cout<<"hello";
}
Void main()
{
        Hello();
}
```

Data Structures

Stacks and Queues

Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a stack, the element deleted from the set is the one most recently inserted: the stack implements a last-in, first-out, or LIFO, policy. Similarly, in a queue, the element deleted is always the one that has been in the set for the longest time: the queue implements a first-in, first-out, or FIFO, policy.

The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP.

```
class Stack
{
        int top;
        public:
        bool is_stack_empty();
        void push(int x);
        int pop();
}
class Queue
{
        int front, rear;
        public:
        bool is_queue_empty();
        void enqueue(int x);
        int dequeue();
}
```

Linked Lists: A linked list is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object.

Each element of a doubly linked list L is an object with an attribute key and two other pointer attributes: next and prev. The object may also contain other satellite data. Given an element x in the list, x.next points to its successor in the linked list, and x.prev points to its predecessor. If x.prev = NIL, the element x has no predecessor and is therefore the first element, or head, of the list. If x.next = NIL, the element x has no successor and is therefore the last element, or tail, of the list. An attribute L.head points to the first element of the list. If L.head D NIL, the list is empty

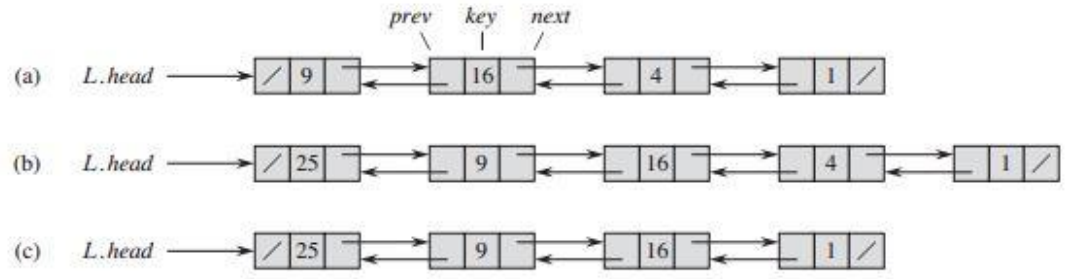


Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. **(b)** Following the execution of LIST-INSERT(L, x), where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.