# C++ OOPS

## Access modifiers

- The members in c++ are private by default
    - Access modifiers
        - private
        - public
        - default

# Constructor

```
class Employee{
public:
    string name;
    string company;
    int age;
    void Introduce(){   //method
        cout<<"hello " + name;
    }

    Employee(string n,string c,int a){ //constructor
        name = n;
        company = c;
        age = a;
    }
};

int main(){
    Employee object = Employee("sasha","google",33);
    object.Introduce();
}
```

- constructor does not have return type
- they should be same name as class name
- needs to be public

# Encapsulation

```
class Employee{
private:
   string name;
   string company;
   int age;
public:
   void setName(string n){ //setters
      name = n;
   }
   string getName(){   //getters
      return name;
   }
   void Introduce(){   //method
      cout<<"hello " + name;
   }

   Employee(string n,string c,int a){ //constructor
      name = n;
      company = c;
      age = a;
   }
};
int main(){
   Employee object = Employee("sasha","google",33);
   object.Introduce();
   object.setName("frigo");
   cout<<object.getName();
}
```

- **The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users.**
- **To achieve this, you must declare class variables/attributes as private (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public get and set methods.**

# ABSTRACTION!

```
class AbstractEmployee{
   virtual void Askforpromotion()=0;
};
class Employee:AbstractEmployee{
private:
   string name;
   string company;
   int age;
public:
   void setName(string n){ //setters
      name = n;
   }
   string getName(){   //getters
      return name;
   }
   void Introduce(){   //method
      cout<<"hello " + name;
   }

   Employee(string n,string c,int a){ //constructor
      name = n;
      company = c;
      age = a;
   }

   void Askforpromotion(){ //abstract method
      cout<<"you got the promotion";
   }
};
```

**Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.**

- **Helps the user to avoid writing the low level code**
- **Avoids code duplication and increases reusability.**
- **Can change internal implementation of class independently without affecting the user.**
- **Helps to increase security of an application or program as only important details are provided to the user.**

```
class Employee{
protected:
    string name;
    string company;
    int age;
public:
    void setName(string n){ //setters
        name = n;
    }
    string getName(){   //getters
        return name;
    }
    Employee(string n,string c,int a){ //constructor
        name = n;
        company = c;
        age = a;
    }
};
class Developer: public Employee{
public:
      string fav;
      Developer(string n,string c,int a,string fa)
       :Employee(n,c,a)
      {
         fav = fa;
      }
      void hello(){
         cout<<name<<"sleepin"<<endl;
      }
```

# INHERITANCE

- **In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:**
- **derived class (child) - the class that inherits from another class**
- **base class (parent) - the class being inherited from**

# POLYMORPHISM

```
class Employee{
protected:
   string name;
   string company;
   int age;
public:
   void setName(string n){ //setters
      name = n;
   }
   string getName(){   //getters
      return name;
   }
   Employee(string n,string c,int a){ //constructor
      name = n;
      company = c;
      age = a;
   }
   virtual void hello(){
      cout<<name<<"working";
   }
};
```

```
class Developer: public Employee{
public:
     string fav;
     Developer(string n,string c,int a,string fa)
      :Employee(n,c,a)
     {
        fav = fa;
     }
     void fixbug(){
        cout<<"fixed the bug"+name;
     }
     void hello(){
        cout<<name<<"sleepin"<<endl;
     }
};
```

```
class Teacher:public Employee{
   string sub;
   public :
   void prepare(){
      cout<<name<<"is preparing"<<sub<<" ";
   }
   Teacher(string n,string c,int a,string sube)
    :Employee(n,c,a)
   {
     sub = sube;
   }
    void hello(){
         cout<<name<<"coding"<<endl;
     }
};
```

```
int main(){
    Employee object =
Employee("sasha","google",33);
    Developer d =
Developer("saldina","fb",45,"c++");
    Teacher t = Teacher("jack","olf",56,"maths");
    Employee *e= &d;
    Employee *r= &t;
    e->hello();
    r->hello();
}
```

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

# Virtual function

- A virtual function is a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at Run-time.

- **Rules for Virtual Functions**
- **Virtual functions cannot be static.**
- **A virtual function can be a friend function of another class.**
- **Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.**
- **The prototype of virtual functions should be the same in the base as well as derived class.**
- **They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.**
- **A class may have virtual destructor but it cannot have a virtual constructor.**

# DESTRUCTORS

- 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.

## Properties

- 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.
-

```
class String {
private:
 char* s;
 int size;

public:
 String(char*); // constructor
 ~String(); // destructor
};

String::String(char* c)
{
 size = strlen(c);
 s = new char[size + 1];
 strcpy(s, c);
}
String::~String() { delete[] s; }
```

Destructors have same name as the class preceded by a tilde (~)
Destructors don't take any argument and don't return anything