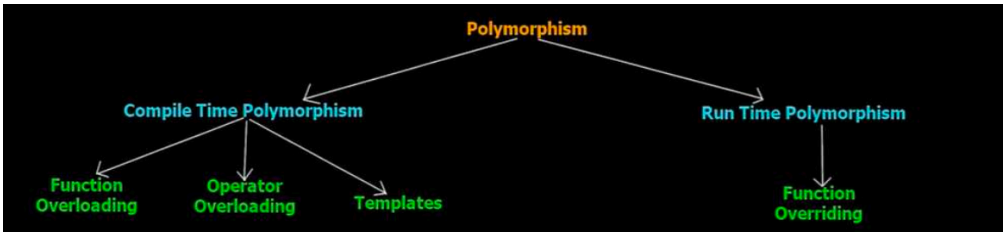
4 - Polymorphism

- Having many forms.
- **Types:**
  - **Compile time Polymorphism**
  - Function Overloading
  - Operator Overloading
  - Template
  - **Runtime Polymorphism**
  - Function Overloading

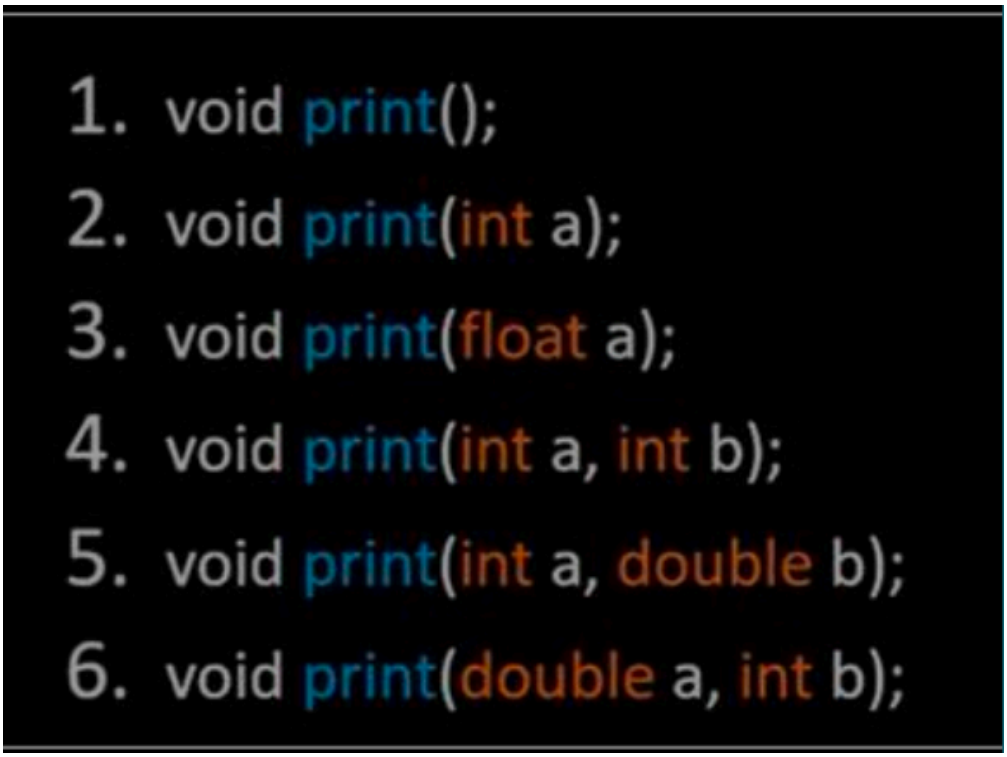


Function Overloading

have more than one function with same name but with different parameters.

Overloaded Functions are differentiated by checking:

- Number of arguments.
- Type and sequence of the arguments.
  Not by return type of the function.



Operator Overloading

A type of polymorphism in which an operator is overloaded to give use defined meaning to it. Operator that are not overloaded are follows:

- scope operator (::)
- sizeof
- member selector (.)
- member pointer selector (*)
- ternary operator (? :)

Binary operator Overloading

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

class Complex{
private :
    int real, img;
public :
    Complex(int r = 0, int i = 0){
        real = r;
        img = i;
    }
    Complex operator + (Complex &obj){
        Complex res;
```

```
        res.real = real + obj.real;
        res.img = img + obj.img;
        return res;
    }
    void print(){
        if (img >= 0) cout << real << " + i" << img << '\n';
        else cout << real << " - i" << abs(img) << '\n';
    }
};


int main(){
    Complex c1(10, 5), c2(40, - 20);
    Complex c3;
    c3 = c1 + c2;
    c3.print();
}
```

Unary Operator Overloading

- to use `cnt++`, write this: `void operator++(int)`

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

class Counter{
    private:
        int counter;
    public :
    Counter(){
        counter = 0;
    }
    int get_count(){
        return counter;
    }
    void operator++(){
        counter++;
    }
};


int main(){
    Counter cnt;
    ++cnt;
}
```

Function Overriding

If we inherit a class into the derived class and provide a definition for one of the base class's function again insider the derived class, then that function is said to be **Overridden** and this mechanism is called **Function Overriding**.

- If function does not exists in derived class then base class function is called.

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

class A{
    public :
        void print(){
            cout << "INSIDE A\n";
        }
        void show(){
            cout << "INSIDE A\n";
        }
};

class B : public A{
    public :
        void print(){
            cout << "INSIDE B\n";
        }
};

int main(){
    cout << "For print : \n";
    A a;
    a.print();
    B b;
    b.print();

    cout << "\nFor show : \n";
    a.show();
    b.show();
}
```

Output:

```
For print :
INSIDE A
INSIDE B
```

```
For show :
INSIDE A
INSIDE A
```

Virtual Function & Polymorphism

Polymorphism means same action but different reaction.

- Runtime
- should not be static.
- can be declared as friend for another class.
- can be accessed by using pointer object.

Without using Virtual:

- In the case of a normal function, the call is determined by the **pointer type**, not the **object type**.

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

class A{
    public :
        void print(){
            cout << "Inside print A\n";
        }
        void show(){
            cout << "Inside print A\n";
        }
};

class B : public A{
    public :
        void print(){
            cout << "Inside print B\n";
        }
        void show(){
            cout << "Inside print B\n";
        }
};

int main(){
    A *pa;
    B b;
```

```
    pa = &b;
    pa->print();
    pa->show();
}
```

Output:

```
Inside print A
Inside print A
```

- `pa` is an `A*` pointer pointing to a `B` object.
- `print()` and `show()` are normal functions.
- For normal functions, calls are resolved by **pointer type**, not object type.
- So `pa->print()` and `pa->show()` call `A`'s versions.

Using Virtual:

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

class A{
    public :
        virtual void print(){
            cout << "Inside print A\n";
        }
        void show(){
            cout << "Inside print A\n";
        }
};

class B : public A{
    public :
        void print(){
            cout << "Inside print B\n";
        }
        void show(){
            cout << "Inside print B\n";
        }
};

int main(){
    A *pa;
    B b;
    pa = &b;
```

```
    pa->print();
    pa->show();
}
```

Output:

```
Inside print B
Inside print A
```

- `print()` is now a virtual function.
- For virtual functions, calls are resolved by **object type**, not pointer type.
- `pa` is an `A*` pointer pointing to a `B` object → so `pa->print()` calls `B`'s version.
- But `Show()` is still a normal function, so `pa->show()` calls `A`'s version.

Pure Virtual Function & Abstract Class

- A **pure virtual function** is a function in a base class that **must be overridden** in derived classes. It has **no definition in the base class**, only a declaration.
- A class that has **at least one pure virtual function** is called an **abstract class**. Object cannot be created.

If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.

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

class Shape{
    public :
        virtual void get_area() = 0;
};

class Circle : public Shape{
    public :
        void get_area(){
            cout << "r = ";
            int r;
            cin >> r;
            cout << "Area = " << (3.14 * r * r) << '\n';
        }
};

class Rectangle : public Shape{
```

```
    public :
        void get_area(){
            cout << "Lenght = ";
            int l;
            cin >> l;
            cout << "Breadth = ";
            int b;
            cin >> b;
            cout << "Area = " << (l * b) << '\n';
        }
};

int main(){
    Circle c;
    c.get_area();
    Rectangle r;
    r.get_area();
}
```

Output:

```
r = 10
Area = 314
Lenght = 40
Breadth = 20
Area = 800
```

Practice:

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

class Animal{
public:
    virtual void eat() = 0;
};

class Dog : public Animal{
public:
    void eat(){
        cout << "Dog food" << '\n';
    }
};

class Cat : public Animal{
```

```
public:
    void eat(){
        cout << "Cat food" << '\n';
    }
};

int main(){
    Dog d;
    d.eat();
    Cat c;
    c.eat();
}
```

Output:

```
Dog food
Cat food
```