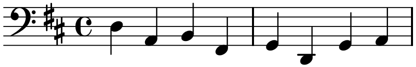
Polymorphism

Polymorphism

polymorphism (noun):
the condition of occurring in several different forms

Polymorphism

polymorphism (noun):
the condition of occurring in several different forms



Method overloading

Two methods of a class can share the same name, so long as the number of parameters or parameter types are different.

Method overloading

```
class Shape
{
    ...
    void init(int h, int w)
    {
        height = h;
        width = w;
    }
    void init(int hw)
    {
        height = hw;
        width = hw;
    }
}
```

```
int main()
{
    Box b;
    b.init(10);
    b.draw();
    b.init(5,5);
    b.draw();
}
```

Method overloading

```
class Shape
{
    ...
    void init(int h, int w)
    {
        height = h;
        width = w;
    }
    void init(int hw)
    {
        height = hw;
        width = hw;
    }
}
```

```
int main()
{
    Box b;
    b.init(10);
    b.draw();
    b.init(5,5);
    b.draw();
}
```

Method overriding

A subclass can *override* a method of a superclass.

The new method has the same name and parameters as the overridden method but can have a different implementation.

Method overriding

```
class TextBox : public Box
{
    string text;
public:
    void setText(string s)
    {
        text = s;
    }
    void draw()                         // Overridden
    {
        for(int i=0; i<height; i++)
        {    if(i==height/2)
                cout << text << endl;
            else
                cout << string(width, '*') << endl;
        }
    }
};
```

Method overriding

```cpp
class Box : public Shape
{
    ...
    void draw()
    { ... }
};

class TextBox : public Box
{
    ...
    void draw()
    { ... }
};
```

```cpp
int main()
{
    Box b;
    b.init(5);
    b.draw();
    TextBox tb;
    tb.init(5);
    tb.setText("Hello");
    tb.draw();
}
```

Method overriding

```
class Box : public Shape
{
    ...
    void draw()
    { ... }
};

class TextBox : public Box
{
    ...
    void draw()
    { ... }
};
```

```
int main()
{
    Box b;
    b.init(5);
    b.draw();
    TextBox tb;
    tb.init(5);
    tb.setText("Hello");
    tb.draw();
}
```

Dynamic binding

By default, C++ will determine which method implementation is used at compile time based on the object's type.

However, a more specific choice could be made at run time because more information is known.

Dynamic binding

Declaring a method as `virtual` tells the compiler to use “dynamic binding” (on the method in this class and any of its subclasses) and make the choice at run time.

```
class Box : public Shape { ... virtual void draw() { ... } };
class TextBox : public Box { ... void draw() { ... } };

void DrawBox(Box& b)
{
    b.draw();
}
```

Which `draw` method will be called in `DrawBox`?

Dynamic binding

Declaring a method as `virtual` tells the compiler to use “dynamic binding” (on the method in this class and any of its subclasses) and make the choice at run time.

with Box reference

```
class Box : public Shape { ... virtual void draw() { ... } };
class TextBox : public Box { ... void draw() { ... } };

void DrawBox(Box& b)
{
    b.draw();
}
```

with TextBox reference

Which `draw` method will be called in `DrawBox`?

Depends on the type of object passed to `DrawBox`!

Dynamic binding

```
int main()
{
    Box b;
    b.init(5);
    DrawBox(b);

}
```

DrawBox will call the draw method of Box.

```
int main()
{
    TextBox tb;
    tb.init(5);
    tb.setText("Hello");
    DrawBox(tb);
}
```

DrawBox will call the draw method of TextBox.

Summary

We discussed three kinds of polymorphism:

1. Method overloading:
   Methods/functions that have the same name but different parameters.

Summary

We discussed three kinds of polymorphism:

1. Method overloading:
   Methods/functions that have the same name but different parameters.

2. Method overriding:
   Methods that have the same name and same parameters, but belong to a superclass and subclass.

Summary

We discussed three kinds of polymorphism:

1. Method overloading:
   Methods/functions that have the same name but different parameters.
2. Method overriding:
   Methods that have the same name and same parameters, but belong to a superclass and subclass.
3. Dynamic binding:
   Method overriding of a function that has been declared `virtual`.

Pure Virtual Functions and Abstract Classes

- A virtual function with *no definition* is pure.
- A class with at least one pvf is abstract.
- An abstract class cannot be instantiated.
  - But pointers and references of abstract type can be created and used.
- If a derived class does not implement all pvf, then it also becomes pure.
- In C++, a virtual function with no definition and a pvf can be different. A pvf can be specified even with a default definition.

Example

```
class A {
public:
	virtual void fun() = 0;
};
class B: public A {
public:
	void fun() { cout << “in B\n”; }
};
int main() {
	A *a = new B;
	a->fun();
	return 0;
}
```

A::fun is a pure virtual function.
A is an abstract class.

Virtual function	Pure virtual function
A virtual function is a member function in a base class that can be redefined in a derived class.	A pure virtual function is a member function in a base class whose declaration is provided in a base class and implemented in a derived class.
The classes which are containing virtual functions are not abstract classes.	The classes which are containing pure virtual function are the abstract classes.
In case of a virtual function, definition of a function is provided in the base class.	In case of a pure virtual function, definition of a function is not provided in the base class.
The base class that contains a virtual function can be instantiated.	The base class that contains a pure virtual function becomes an abstract class, and that cannot be instantiated.
If the derived class will not redefine the virtual function of the base class, then there will be no effect on the compilation.	If the derived class does not define the pure virtual function; it will not throw any error but the derived class becomes an abstract class.
All the derived classes may or may not redefine the virtual function.	All the derived classes must define the pure virtual function.

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.

Virtual Destructor in C++

A **destructor in C++** is a member function of a class used to free the space occupied by or delete an object of the class that goes out of scope. A destructor has the same name as the name of the constructor function in a class, but the destructor uses a tilde **(~)** sign before its function name.

Virtual Destructor

A virtual destructor is used to free up the memory space allocated by the derived class object or instance while deleting instances of the derived class using a base class pointer object. A base or parent class destructor use the **virtual** keyword that ensures both base class and the derived class destructor will be called at run time, but it called the derived class first and then base class to release the space occupied by both destructors.

Why we use virtual destructor in C++?

When an object in the class goes out of scope or the execution of the main() function is about to end, a destructor is automatically called into the program to free up the space occupied by the class' destructor function. When a pointer object of the base class is deleted that points to the derived class, only the parent class destructor is called due to the early bind by the compiler. In this way, it skips calling the derived class' destructor, which leads to memory leaks issue in the program. And when we use virtual keyword preceded by the destructor tilde (~) sign inside the base class, it guarantees that first the derived class' destructor is called. Then the base class' destructor is called to release the space occupied by both destructors in the inheritance class.