

Object-oriented Programming

Function Overriding

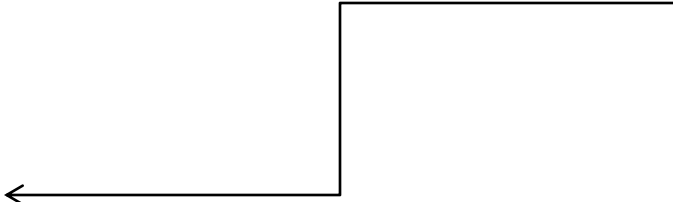
Function Overriding

- A way to support polymorphism in the program is by using **Function Overriding**
- A derived class can define a function with the **same name** and **same type signature** as defined in its base class
- The child “***extends***” the functionality of its base class through the overridden function

Example

```
class Base
{
    public:
    void f()
    { cout << "This is base"; }
};
```

Base class implementation
(overridden function)



```
class Derived
{
    public:
    void f()
    { cout << "This is derived"; }
};
```

Derived class
implementation
(overriding function)



Function Overriding

- The compiler determines which *version* of the function to call based on the type of object used for the call

- For the previous example:

Base ***b***;

Derived ***d***;

b.f(); // base version of f() is called

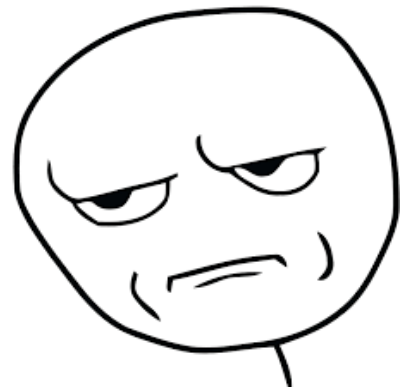
d.f(); // derived version of f() is called

Overloading vs Overriding

- **Function overloading** is use of the same name but different parameter signature; whereas **function overriding** is the use of same name and same type signatures
- **Function overriding** requires an inheritance relationship; whereas **function overloading** does not require an inheritance relationship

Using Base class Pointer

- Recall that a parent class variable can be used to hold reference of its child class
- When the parent and child classes have overridden functions, this can lead to unexpected results



Example

```
class Base
{
    public:
    void f()
    { cout << "This is base"; }
};
```

```
class Derived
{
    public:
    void f()
    { cout << "This is derived"; }
};
```

```
int main()
{
    Derived d;
    Base * b = &d;
    b->f();
}
```

Output:
This is base

Early/Compile-time Binding

- In the previous example, we had a *derived class reference* stored in *base class pointer* but still the compiler called the base class version of the overridden function
- This behavior is called **Early Binding** or **Compile-time Binding**
- The compiler only “*sees*” the “*type*” of object and use it for calling the appropriate function regardless of what type of reference is stored in it

Late or Runtime Binding

- With **Late Binding** or **Runtime Binding**, the compiler checks the reference stored in the base class pointer for calling the appropriate overridden function
- We can enforce runtime binding by declaring the function as *virtual*

Virtual Function

- A **virtual function** allows the compiler to choose the correct overridden function to call
- **Virtual functions** allow late binding or runtime binding
- We can declare a function to be virtual by using the keyword **virtual**

Example

```
class Base
{
    public:
    virtual void f()
    { cout << "This is base"; }
};
```

```
class Derived
{
    public:
    void f()
    { cout << "This is derived"; }
};
```

```
int main()
{
    Derived d;
    Base * b = &d;
    b->f();
}
```

Output:
This is derived

Virtual Function

- If we declare a function as *virtual* in the parent class and override it, the overridden versions in all of the derived classes are also implicitly considered as *virtual*
- However, if we want, we can additionally use the *virtual* keyword with overridden functions in derived classes but that would just be needless