C++

Inheritance: virtual functions

Example

```
class B {
 public:
  B() { }
  void f() { cout << “B::f()” << endl; }
};

class D1 : public B {
  public:
    D1() { }
    void f() { cout << “D1::f()” << endl; }
};

Class D2 : public B {
  public:
   D2() { }
   void f() { cout << “D2::f()” << endl; }
}

void g (B* b) { b->f(); }
void g (B& b) { b.f(); }
```

- We would like g() to print D1::f() for a D1 object and D2::f() for a D2 object
- However as seen in previous lesson, B::f() will be printed

Static binding

- When the compiler generates the function call, it looks at the static type of the argument (in the previous example: B)
  - Method B::f() is called
- This is called **static binding**
- The method to be called is decided by a static analysis at compile time
- Static binding is used in languages like: C, C++, Fortran, Pascal

Dynamic binding

- The other possibility is **dynamic binding**
- With dynamic binding, the method to be called is selected dynamically (during runtime) based on the (runtime) type of the object
- Dynamic binding is used in languages like Java, python, ruby, obj-c)

Polymorphism

- When you have a pointer to an object, the object may be of a derived class
- This is called (dynamic) **polymorphism**
- Poly = several – morphos = form (from Greek)
- Polymorphism allows us to work with objects without knowing their precise type at compile time
- For example, in g(B* b) the runtime type of b is not known at compile time and not necessary known by the developer who wrote the code for g()

In C++

- In C++, dynamic binding is achieved by using the keyword **virtual** (for a method)
- A virtual function is a method declared with the virtual keyword in the class definition (if the method definition is outside the class, do not put virtual)

```
class B {
 public:
  virtual void f();
};

void B::f() {
 cout << “B::f()” << endl;
}
```

Fixing the first example

```
class B {
 public:
  B() { }
  virtual void f() { cout << “B::f()” << endl; }
};

class D1 : public B {
  public:
    D1() { }
    virtual void f() { cout << “D1::f()” << endl; }
};

Class D2 : public B {
  public:
   D2() { }
   virtual void f() { cout << “D2::f()” << endl; }
}

void g (B* b) { b->f(); }
void g (B& b) { b.f(); }
```

```
D1 d1;
D2 d2;

g(&d1); // prints D1::f()
g(&d2); // prints D2::f()
g(d1); // prints D1::f()
g(d2); // prints D2::f()
```

Virtual functions

- A method declared virtual in a base class is automatically virtual in the derived class (directly and indirectly derived classes)
  - In theory, you do not have to repeat the virtual keyword
  - In practice, it is good style to use the virtual keyword if a method is virtual (it improves readability)

Virtual functions

- There is a small performance and memory overhead for using virtual functions, because the compiler is adding a layer of indirection
  - The compiler creates one pointer per object with at least one virtual function (the “v-pointer”)
  - The compiler creates a table in a class with at least one virtual function (the “v-table”) with one pointer per virtual method

Overriding

- When a derived class defines a method with the same name (and signature) as a base class method, we say that it **overrides** the base class method
- !! Overriding happens only if the method is virtual !! Otherwise it is name hiding and source of problems
- Overriding is necessary if the base class is not sufficient for the derived classes:
  - Derived classes can further specialize a virtual method by overriding it

Overriding

- When overriding a base class method, it is possible to:
  - Augment the method in the base class
  - Redefine it entirely
- If you want to augment the method in the base class, you need to explicitly call it in the code of the overriden method in the derived class

Example

```
class B {
 public:
  B() { }
  virtual void f() { cout << “B::f()” << endl; }
};

class D : public B {
  public:
    D() { }
    virtual void f() {
         B::f(); // or this->B::f();
         cout << “D::f()” << endl; }
};
```

```
D d;

g(&d); // prints:
          // B::f()
          // D::f()
```

Virtual vs non-virtual

- When designing a base class, you need to decide if a method will be virtual or not
- This decision can be made by asking the question:
  - If this method should have exactly the same behavior for all derived classes then it should be non-virtual
  - If this method needs to be specialized depending on the object, then it should be virtual
- This decision has to be taken when you design base class

Overriding vs. overloading

- Do not mix overriding and overloading. This are two different concepts
- Overloading:
  - Allow to use same name for functions or methods with different arguments
  - Decision of the function or method to call is done by the compiler at compile time (no dynamic binding involved)
- Overriding:
  - Allow to specialize the behavior of an existing method by providing a different implementation in the derived classes
  - Overriding is only possible with inheritance and dynamic binding
  - Decision of the method to call done at runtime based on the dynamic type of the object (dynamic binding)

Virtual destructor

- You need a virtual destructor when someone will delete a derived class object via a base class pointer
- Example: Base* p = new Derived; delete p;
- A simple rule is to have a virtual destructor whenever your class has at least one virtual function
- Note: if the base class destructor is virtual, the derived classes destructors will be virtual automatically

Example

```
class B {
 public:
  B() { cout<< “B()” << endl; }
  ~B() { cout<< “~B()” << endl; }
};
class D : public B {
 public:
  D() { cout<< “D()” << endl; }
  ~D() { cout<< “~D()” << endl; }
};

int main() { B *b = new D; delete b; }
```

Result:

B()
D()
~B()

Example

```
class B {
 public:
  B() { cout<< “B()” << endl; }
  virtual ~B() { cout<< “~B()” << endl; }
};
class D: public B {
 public:
  D() { cout<< “D()” << endl; }
  ~D() { cout<< “~D()” << endl; }
};

int main() { B *b = new D; delete b; }
```

Result:

B()
D()
~D()
~B()

Virtual constructor

- Be careful that C++ does not support the concept of **virtual constructor**
- It means: you can not write **virtual Base() {}**, where Base is the name of the class.
  - You will get a compile time error
- You can get around that by using some tricks though

Calling virtual functions in constructor (or destructor)

- You should never rely on virtual functions called in constructor (or destructor)
- The result is probably not going to do what you think (be careful if you are used to Java)
- The reason is: during base class construction of a derived class object, the type of the object is that of **the base class**

Example

```
class B {
 public:
  B() { f(); }
  virtual void f() { cout << “B::f()” << endl; }
};

class D : public B {
  public:
    D() { }
    virtual void f() { cout << “D::f()” << endl; }
};

int main() {
 D d;
}
```

Result will be:

B::f()

Pure virtual functions

- There are virtual methods for which an implementation would make no sense because they are too abstract
- Such method serve only to provide a common interface to the methods in the derived class
- Such a method is called a **pure virtual (member) function**
- A method declared as a pure virtual function has body (no associated code)

Pure virtual functions

- The syntax for a pure virtual functions is:

virtual *type name*(*type args*, …) = 0;

```
class B {
 public:
   virtual void f() = 0;
};

class D : public B {
  public:
    D() { }
    virtual void f() { cout << “D::f()” << endl; }
};
```

Abstract base classes

- Abstract base classes (ABC) are classes where at least one method is a pure virtual function
- If a derived class does not implement the pure virtual methods, then it is also an ABC
- ABC in C++ serve a purpose similar to interface in Java
- You can not create an instance of an ABC

ABC - 2

- An ABC can not be used as an argument type in a call by value
- It can not be used as a return type (if returned by value)
- It can not be used as the type of an explicit conversion
- Pointers and references to ABC can be declared

Interface

- A base class contains what is common to several classes
- If pure virtual functions are used, the class is an abstract class and it is used as an interface common to the derived classes
- In the case of pure virtual function, no code is inherited, but instead we inherit an interface that can be reused by all derived classes
- Interfaces are really important in object oriented programming: they correspond to abstract concepts (separated from an implementation)