C++ Programming I

Basics of Object-Oriented Programming
Polymorphism

C++ Programming
April 19, 2018

Dr. P. Arnold
Bern University of Applied Sciences

Agenda

- **Polymorphism**
  - Virtual Functions
  - Virtual Destructors
  - Virtual Function Table

Agenda

- **Polymorphism**
  - Virtual Functions
  - Virtual Destructors
  - Virtual Function Table
- **Abstract Classes**

Agenda

- **Polymorphism**
  - Virtual Functions
  - Virtual Destructors
  - Virtual Function Table
- **Abstract Classes**
- **Interface Classes**

Polymorphism

Basics of Polymorphism
Intro

- “Poly” is Greek for many, and “morph” means form, thus many forms
- Objects of different types are treated similarly
- There are different forms of Polymorphism:
  1. Compile-Time Polymorphism
     - Function Overloading
  2. Run-Time Polymorphism
     - Virtual Functions
- The following slides are based on the examples given by:
  `http://www.learncpp.com/cpp-tutorial/`
  `121-pointers-and-references-to-the-base-class-of-derived-objects/`

Basics of Polymorphism

An Example using Inheritance

```cpp
#include <string>
#include <iostream>

class Animal
{
protected:
    // Protected Constructor
    Animal(std::string name) : m_name(name){}
    std::string m_name;

public:
    std::string getName() { return m_name; }
    std::string speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string name): Animal(name){}
    std::string speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string name): Animal(name){}
    std::string speak() { return "Woof"; }
};
```

Basics of Polymorphism

Pointers and References to Derived Objects

- We can set pointers or references to derived objects
- What is the output of the following code snippet?

```cpp
int main()
{
    Cat cat("Fred");
    std::cout << "Cat is named " << cat.getName() << ", and it says " << cat.speak() << std::endl;

    Dog dog("Garbo");
    std::cout << "Dog is named " << dog.getName() << ", and it says " << dog.speak() << std::endl;

    Animal *pAnimal = &cat;
    std::cout << "AnimalPtr is named " << pAnimal->getName() << ", and it says " << pAnimal->speak() << std::endl;

    pAnimal = &dog;
    std::cout << "AnimalPtr is named " << pAnimal->getName() << ", and it says " << pAnimal->speak() << std::endl;

    return 0;
}
```

Basics of Polymorphism

Polymorphic Behaviour

```
Output:

Cat is named Fred, and it says Meow
Dog is named Garbo, and it says Woof
AnimalPtr is named Fred, and it says ???
AnimalPtr is named Garbo, and it says ???
```

- Not what we want!
- Because `AnimalPtr` is an `Animal` pointer, it can only see the Animal portion of the class. Consequently, `AnimalPtr->speak()` calls `Animal::speak()` rather than the `Dog::Speak()` or `Cat::speak()` function.
- But why should we use Pointers and References to Base class? Instead we could work directly with the derived classes!

Basics of Polymorphism

Need of Pointer and References to Base Class I

- We want a function to print animal's name and sound. Without pointers to base class we would have to do function overloading for every animal!

```
// Overload functions fo reach animal! There are a lot
void report(Cat& cat)
{
    std::cout << cat.getName() << " says " << cat.speak();
}

void report(Dog& dog)
{
    std::cout << dog.getName() << " says " << dog.speak();
}
```

Basics of Polymorphism

Need of Pointer and References to Base Class I

- We want a function to print animal's name and sound. Without pointers to base class we would have to do function overloading for every animal!

```
// Overload functions fo reach animal! There are a lot
void report(Cat& cat)
{
    std::cout << cat.getName() << " says " << cat.speak();
}

void report(Dog& dog)
{
    std::cout << dog.getName() << " says " << dog.speak();
}
```

- However, because Cat and Dog are derived from Animal, Cat and Dog are Animals. Therefore, it makes sense that we should be able to do something like this:

```
// It would be nice to write one function only!
void report(Animal& Animal)
{
    std::cout << Animal.getName() << " says " << Animal.speak()
        << '\n';
}
```

- The problem: `Animal::speak()` will be called!

Basics of Polymorphism

Need of Pointer and References to Base Class II

- Imagine we have multiple dogs and cats and want to use them in a array

```
int main()
{
    Cat cats[] = { Cat("Fred"), Cat("Misty"), Cat("Zeke") };
    Dog dogs[] = { Dog("Garbo"), Dog("Pooky"), Dog("Truffle") };

    for (int i=0; i < 3; ++i)
    {
        cout << cats[i].getName() << " says " << cats[i].speak();
    }

    for (int i=0; i < 3; ++i)
    {
        cout << dogs[i].getName() << " says " << dogs[i].speak();
    }

    return 0;
}
```

- Now, consider what would happen if you had 30 different types of animals. You'd need 30 arrays, one for each type of animal!

Basics of Polymorphism

Need of Pointer and References to Base Class II

- However, because Cat and Dog are derived from Animal, Cat and Dog are Animals. Therefore, it makes sense that we should be able to do something like this:

```cpp
int main()
{
    Cat fred("Fred"), misty("Misty"), zeke("Zeke");
    Dog garbo("Garbo"), pooky("Pooky"), truffle("Truffle");

    // Set up an array of pointers to animals, and set those
    // pointers to our Cat and Dog objects
    Animal *animals[] = { &fred, &garbo, &misty, &pooky,
         &truffle, &zeke };
    for (int i=0; i < 6; ++i)
    {
        cout << animals[i]->getName() << " says " <<
             animals[i]->speak();
    }
    return 0;
}
```

- While this compiles and executes, unfortunately the fact that each element of array "animals" is a pointer to an Animal means that `animals[i]->speak()` will call `Animal::speak()` instead of the derived class version of `speak()` that we want.

Polymorphic Behavior

Virtual Functions

- The way out of the problem are virtual functions!
- A **virtual function** is a special type of function that, when called, resolves to the most-derived version of the function that exists between the base and derived class.
- This capability is known as **Polymorphism**.
- The previous examples implemented using virtual function is shown on the next slide

Polymorphic Behavior

The keyword `virtual`

```cpp
class Animal
{
protected:
    std::string m_name;
    Animal(std::string name) : m_name(name) {}

public:
    std::string getName() { return m_name; }
    virtual std::string speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string name) : Animal(name) {}
    virtual std::string speak() { return "Meow"; } // virtual
};

class Dog: public Animal
{
public:
    Dog(std::string name) : Animal(name) {}
    virtual std::string speak() { return "Woof"; } // virtual
};

void report(Animal &animal)
{
    std::cout << animal.getName() << " says " << animal.speak();
}
```

Polymorphic Behavior

The keyword `virtual`

```
int main()
{
    Cat cat("Fred");
    Dog dog("Garbo");

    report(cat);
    report(dog);
}
```

- Output?

Polymorphic Behavior

The keyword `virtual`

```cpp
int main()
{
    Cat cat("Fred");
    Dog dog("Garbo");

    report(cat);
    report(dog);
}
```

- Output?

```
Output:

Fred says Meow
Garbo says Woof
```

- It works!

Polymorphic Behavior

The keyword `virtual`

- Similarly, the following example works

```
Cat fred("Fred"), misty("Misty"), zeke("Zeke");
Dog garbo("Garbo"), pooky("Pooky"), truffle("Truffle");

// Set up an array of pointers to animals, and set those
// pointers to our Cat and Dog objects
Animal *animals[] = { &fred, &garbo, &misty, &pooky, &truffle,
     &zeke };
for (int i=0; i < 6; ++i)
{
    cout << animals[i]->getName() << " says " <<
         animals[i]->speak() << endl;
}
```

- Produces the output:

Polymorphic Behavior

The keyword `virtual`

- Similarly, the following example works

```
Cat fred("Fred"), misty("Misty"), zeke("Zeke");
Dog garbo("Garbo"), pooky("Pooky"), truffle("Truffle");

// Set up an array of pointers to animals, and set those
// pointers to our Cat and Dog objects
Animal *animals[] = { &fred, &garbo, &misty, &pooky, &truffle,
    &zeke };
for (int i=0; i < 6; ++i)
{
    cout << animals[i]->getName() << " says " <<
        animals[i]->speak() << endl;
}
```

- Produces the output:

```
Output:

Fred says Meow
Garbo says Woof
Misty says Meow
Pooky says Woof
Truffle says Woof
Zeke says Meow
```

- The signature of the derived class function must exactly match the signature of the base class virtual function

Polymorphic Behavior

Good to Know

- **Don't** call virtual functions from the Constructor or destructor the base class! Remember that the base class is constructed before the derived class, *i.e.* that means the derived version of the function does not yet exist and the base function is called.
- Resolving a virtual function call takes longer than resolving a regular one (virtual function tables). Use `virtual` only when needed!
- The **signature** of the derived class function must exactly match the signature of the base class virtual function! Otherwise the base class function will be used
- To avoid unintended signature matching errors the *override* specifier comes handy! Check the next example:
- Use **virtual destructors** when dealing with Inheritance!

Polymorphic Behavior

`override` Specifier

```
class A
{
public:
    virtual const std::string getName1(int x) { return "A"; }
    virtual const std::string getName2(int x) { return "A"; }
};

class B : public A
{
public:
    virtual const std::string getName1(short int x)
    { // note: parameter is a short int
        return "B";
    }
    virtual const std::string getName2(int x) const
    { // note: function is const
        return "B";
    }
};

int main()
{
    B b;
    A &BaseRef = b;
    std::cout << BaseRef.getName1(1) << std::endl;
    std::cout << BaseRef.getName2(2) << std::endl;

    return 0;
}
```

Polymorphic Behavior

`override` Specifier

```cpp
class A
{
public:
    virtual const std::string getName1(int x) { return "A"; }
    virtual const std::string getName2(int x) { return "A"; }
    virtual const std::string getName3(int x) { return "A"; }
};

class B : public A
{
public:
    virtual const std::string getName1(short int x) override {
        return "B"; } // compile error, not an override

    virtual const std::string getName2(int x) const override {
        return "B"; } // compile error, not an override

    virtual const std::string getName3(int x) override {
        return "B";  } // okay,  is an override
};
```

- There is no performance penalty for using the override specifier.

Tipp

Apply the override specifier to every intended override function you write.

Polymorphic Behavior

`final` Specifier

- Use the `final` specifier to prohibit overriding a function

```cpp
class A
{
public:
    virtual const std::string getName() { return "A"; }
};

class B : public A
{
public:
    // final specifier makes this function no longer overridable
    virtual const std::string getName() override final
    {
        return "B";
    } // okay, overrides A::getName()
};

class C : public B
{
public:
    virtual const std::string getName() override
    {
        return "C";
    } // compile error: overrides B::getName(), which is final
};
```

Virtual Destructors

Why we need Virtual Destructors?

- Similarly, as for other virtual function, calling a function using a pointer of type `Base*` that actually points to `derived*` will call the Base's class function if not marked as `virtual`
- The same holds for the destructor
- Check next example

Virtual Destructors

Why we need Virtual Destructors?

```cpp
#include <iostream>
class Base
{
public:
    ~Base() // note: not virtual
    {
        std::cout << "Calling ~Base()" << std::endl;
    }
};

class Derived: public Base
{
private:
    int* m_array;

public:
    Derived(int length)
    {
        m_array = new int[length];
    }

    ~Derived() // note: not virtual
    {
        std::cout << "Calling ~Derived()" << std::endl;
        delete[] m_array;
    }
};
```

Virtual Destructors

Why we need Virtual Destructors?

```
int main()
{
    Derived *derived = new Derived(5);
    Base *base = derived ;
    delete base;

    return 0;
}
```

- Output? What constructors and destructors are called?

Virtual Destructors

Why we need Virtual Destructors?

```
int main()
{
    Derived *derived = new Derived(5);
    Base *base = derived ;
    delete base;

    return 0;
}
```

- Output? What constructors and destructors are called?

```
Output:

Calling ~Base()
```

- Hoppla! Again a memory leak!

Virtual Destructors

Why we need Virtual Destructors?

```cpp
#include <iostream>
class Base
{
public:
    virtual ~Base() // note: virtual
    {
        std::cout << "Calling ~Base()" << std::endl;
    }
};

class Derived: public Base
{
private:
    int* m_array;

public:
    Derived(int length)
    {
        m_array = new int[length];
    }

    virtual ~Derived() // note: virtual
    {
        std::cout << "Calling ~Derived()" << std::endl;
        delete[] m_array;
    }
};
```

Virtual Destructors

Why we need Virtual Destructors?

```cpp
int main()
{
    Derived *derived = new Derived(5);
    Base *base = derived;
    delete base;

    return 0;
}
```

- Output?

Virtual Destructors

Why we need Virtual Destructors?

```
int main()
{
    Derived *derived = new Derived(5);
    Base *base = derived;
    delete base;

    return 0;
}
```

- Output?

```
Output:

Calling ~Derived()
Calling ~Base()
```

- It works!

Virtual Function Table

A Lookup Table of Functions

- To ensure correct function calling using inheritance, C++ uses a special form of late binding known as the **virtual table**.
- First, every class that uses virtual functions (or is derived from a class that uses virtual functions) is given its own virtual table.
- Second, the compiler also adds a **hidden pointer** to the base class, further named `*__vptr`.
- It's simply a static array that the compiler sets up at compile time, containing one entry for each virtual function holding a function pointer that points to the most-derived function accessible by that class.
- It's most simply explained by an example

Virtual Function Table

By an Example

```
class Base
{
public:
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    virtual void function1() {};
};

class D2: public Base
{
public:
    virtual void function2() {};
};
```

Virtual Function Table

By an Example

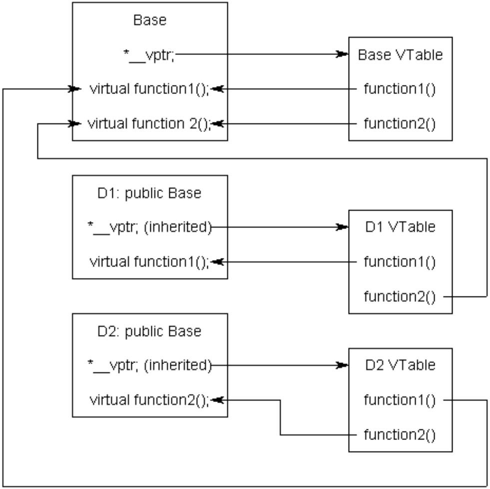
```
class Base
{
public:
    FunctionPointer *__vptr;
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    virtual void function1() {};
};

class D2: public Base
{
public:
    virtual void function2() {};
};
```

- Example of how the hidden pointer is added to the base class
- `*__vptr` is inherited by the derived classes

Virtual Function Table

By an Example

- The `*__vptr` in each class points to the virtual table for that class.
- The entries in the virtual table point to the most-derived version of the function objects of that class



Virtual Function Table

By an Example

- Creating an object of type `D1`

```cpp
int main()
{
    D1 d1;
}
```

- Because `d1` is a `D1` object, `d1` has its `*__vptr` set to the D1 virtual table.

Virtual Function Table

By an Example

- Set a base pointer to `D1`

```
int main()
{
    D1 d1;
    Base *dPtr = &d1;
}
```

- `dPtr->__vptr` points to the D1 virtual table, although `dPtr` is of type Base

Virtual Function Table

By an Example

- Set a base pointer to `D1`

```
int main()
{
    D1 d1;
    Base *dPtr = &d1;
}
```

- `dPtr->__vptr` points to the D1 virtual table, although `dPtr` is of type Base
- Calling `dPtr->function1()`?

```
int main()
{
    D1 d1;
    Base *dPtr = &d1;
    dPtr->function1();
    dPtr->function2();
}
```

- `function1()` is a virtual function
- The program uses `dPtr->__vptr` to get to `D1`'s virtual table
- `dPtr->function1()` resolves to `D1::function1()`
- `dPtr->function2()` resolves to `Base::function2()`

Abstract Classes

Abstract Base Classes
and Pure Virtual Functions

- A base class that cannot be instantiated is called an abstract base class.
- The only purpose of abstract base classes is to inherit from.
- C++ allows you to create an abstract base class using **pure virtual functions**.
- A virtual method is said to be pure virtual when it has a declaration as shown in the following:

```
class AbstractBase
{
public:
    virtual void doSomething() = 0;
};
```

Abstract Base Classes

and Pure Virtual Functions

- A base class that cannot be instantiated is called an abstract base class.
- The only purpose of abstract base classes is to inherit from.
- C++ allows you to create an abstract base class using **pure virtual functions**.
- A virtual method is said to be pure virtual when it has a declaration as shown in the following:

```
class AbstractBase
{
public:
    virtual void doSomething() = 0;
};
```

- The derived class **is forced** to implement the virtual function of the base class!

```
class Derived: public AbstractBase
{
public:
    void doSomething() // pure virtual fn. must be implemented!
    {
        cout << "Implemented virtual function" << endl;
    }
};
```

Abstract Classes

Remember the example from before

```
class Animal
{
protected:
    std::string m_name;
    Animal(std::string name) : m_name(name) {}

public:
    std::string getName() { return m_name; }
    virtual std::string speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string name) : Animal(name) {}
    virtual std::string speak() { return "Meow"; } // virtual
};

class Dog: public Animal
{
public:
    Dog(std::string name) : Animal(name) {}
    virtual std::string speak() { return "Woof"; } // virtual
};

void report(Animal &animal)
{
    std::cout << animal.getName() << " says " << animal.speak();
}
```

Abstract Classes

Remember the example from before

- It is still possible to create derived classes that do not redefine function `speak()`!

```cpp
#include <iostream>
class Cow: public Animal
{
public:
    Cow(std::string name): Animal(name)
    {
    }

    // We forgot to redefine speak
};

int main()
{
    Cow cow("Betsy");
    std::cout << cow.getName() << " says " << cow.speak() <<
        '\n';
}

// Output
Betsy says ???
```

Abstract Classes
The Better Approach

- A better solution to this problem is to use a pure virtual function!
- We redefine the base class as follows:

```cpp
#include <string>
class Animal // This Animal is an abstract base class
{
protected:
    std::string m_name;

public:
    Animal(std::string name): m_name(name)
    {
    }

    std::string getName() { return m_name; }
    virtual std::string speak() = 0;
    // speak is now a pure virtual function
};
```

- `speak()` is now a **pure virtual function**.
- `Animal` is now an **abstract base class** and can not be instantiated.
- There would be a compiler error when not defining `Cow::speak()`.

Abstract Classes

The Better Approach

- This fixes the code from before:

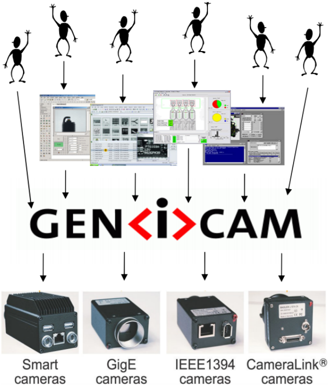
```cpp
#include <iostream>
class Cow: public Animal
{
public:
    Cow(std::string name) : Animal(name)
    {
    }

    virtual std::string speak() { return "Moo"; }

};

int main()
{
    Cow cow("Betsy");
    std::cout << cow.getName() << " says " << cow.speak() << '\n';
}

// Output
Betsy says Moo
```

- `speak()` is now a **pure virtual function**.
- `Animal` is now an **abstract base class** and can not be instantiated.
- There would be a compiler error when not defining `Cow::speak()`.

Interface Classes

Interface Class
Example GenICam

- GenICam: **GEN**eric programming **I**nterface for **CAM**eras
- Realised with a couple of abstract interfaces and multiple use of inheritance



GenICam can connect the Customer..

- …to all cameras
- …through all libraries
- …giving access to all smart features

GenICam can support…

- …any interface technology
- …products from any vendor
- …products with different register layout

GenICam is easy to integrate for…

- …customers
- …camera vendors
- …software library vendors
- …frame grabber / driver vendors

Interface Class

Example GenICam

- An **interface class** is a class that has no member variables and all of the functions are pure virtual!
- An **interface class** is a pure definition without implementation

```
class ICameraDevice
{
public:
    virtual bool getDeviceID( deviceHandle& handle ) = 0;
    virtual bool connectDevice( deviceHandle handle ) = 0;
    virtual bool closeDevice( deviceHandle handle ) = 0;
};
```

- Each camera will inherit from the `ICameraDevice` and has to provide the implementations of the interface
- Easy to extend, *i.e.* USB-Cam, Ethernet-Cam, Virtual-Camera, etc.

Thank You

Questions

???