C++ Program Design -- Virtual Functions

Junjie Cao @ DLUT

Summer 2016

http://jjcao.github.io/cPlusPlus

most important and powerful aspects of inheritance -- virtual functions.

why we need them

```
class Base{
protected:    int m_nValue;
public:
    Base(int nValue): m_nValue(nValue){ }
    const char* GetName() { return "Base"; }
    int GetValue() { return m_nValue; }
};

class Derived: public Base{
public:
    Derived(int nValue): Base(nValue){}
    const char* GetName() { return "Derived"; }
};
```

```cpp
Derived cDerived(5);
// These are both legal!
Base &rBase = cDerived;
Base *pBase = &cDerived;

cout << "cDerived is a " << cDerived.GetName() << endl;
cout << "rBase is a " << rBase.GetName() << endl;
cout << "pBase is a " << pBase->GetName() << endl;
```

cDerived is a Derived and has value 5
rBase is a Base and has value 5
pBase is a Base and has value 5

This result may not be quite what you were expecting at first!

rBase and pBase are a Base reference and pointer, they can only see members of Base (or any classes that Base inherited)

Use for pointers and references to base classes

- “The above examples seem kind of silly. Why would I set a pointer or reference to the base class of a derived object when I can just use the derived object?”

```
void Report(Cat &cCat){
    cout << cCat.GetName() << " says " << cCat.Speak() << endl;
}
void Report(Dog &cDog){
    cout << cDog.GetName() << " says " << cDog.Speak() << endl;
}
```

- Not too difficult, but consider what would happen if we had 30 different animal types instead of 2.

```
void Report(Animal &rAnimal){
    cout << rAnimal.GetName() << " says " << rAnimal.Speak() << endl;
}
```

Use for pointers and references to base classes

```
Cat acCats[] = { Cat("Fred"), Cat("Tyson"), Cat("Zeke") };
Dog acDogs[] = { Dog("Garbo"), Dog("Pooky"), Dog("Truffle") };
for (int iii=0; iii < 3; iii++)
    cout << acCats[iii].GetName() << " says " << acCats[iii].Speak() << endl;
for (int iii=0; iii < 3; iii++)
    cout << acDogs[iii].GetName() << " says " << acDogs[iii].Speak() << endl;
```

```
Cat cFred("Fred"), cTyson("Tyson"), cZeke("Zeke");
Dog cGarbo("Garbo"), cPooky("Pooky"), cTruffle("Truffle");
Animal *apcAnimals[] = { &cFred, &cGarbo, &cPooky, &cTruffle, &cTyson, &cZeke };
for (int iii=0; iii < 6; iii++)
    cout << apcAnimals[iii]->GetName() << " says " << apcAnimals[iii]->Speak() << endl;
```

Virtual functions and polymorphism

```
class Base{
public: virtual const char* GetName() { return "Base"; }
};

class Derived: public Base{
public: virtual const char* GetName() { return "Derived"; }
};

Derived cDerived;
Base &rBase = cDerived;
cout << "rBase is a " << rBase.GetName() << endl;
```

Virtual functions and polymorphism

```
Derived cDerived;
Base &rBase = cDerived;
cout << "rBase is a " << rBase.GetName() << endl;
```

- Base::GetName() is virtual, which tells the program to go look and see if there are any more-derived versions of the function available.

- Because the Base object: rBase is pointing to is actually a Derived object,
- the program will check every inherited class between Base and Derived
- and use the most-derived version of the function that it finds.

```
class A{public: virtual char* GetName() { return "A"; }};
class B:public A{public:virtual char* GetName() { return "B"; }};
class C:public B{public:virtual char* GetName() { return "C"; }};
class D:public C{public:virtual char* GetName() { return "D"; }};

int main(){
    C cClass;
    A &rBase = cClass;
    cout << "rBase is a " << rBase.GetName() << endl;

    return 0;
}
```

rBase is a C

A more complex example

```
class Animal{
protected:    std::string m_strName;
    // We're making this constructor protected because we don't want people creating Animal objects directly, but we still want derived classes to be able to use it.
    Animal(std::string strName)   : m_strName(strName)   {   }
public:
    std::string GetName() { return m_strName; }
    virtual const char* Speak() { return "???"; }
};

class Cat: public Animal{
public:
    Cat(std::string strName) : Animal(strName) { }
    virtual const char* Speak() { return "Meow"; }
};
```

```
class Dog: public Animal{
public:
    Dog(std::string strName) : Animal(strName) { }
    virtual const char* Speak() { return "Woof"; }
};
```

- didn’t make Animal::GetName() virtual. This is because GetName() is never overridden in any of the derived classes, therefore there is no need.

```
void Report(Animal &rAnimal){
    cout << rAnimal.GetName() << " says " << rAnimal.Speak() << endl;
}

int main(){
    Cat cCat("Fred");    Dog cDog("Garbo");
    Report(cCat);    Report(cDog);
}
```

A word of warning

- the signature of the derived class function must exactly match the signature of the base class virtual
- If the derived class function has different parameter types, the program will likely still compile fine, but the virtual function will not resolve as intended.

Use of the virtual keyword

```
class Base{public:
    virtual const char* GetName() { return "Base"; }
};

class Derived: public Base{
public:
    const char* GetName() { return "Derived"; } // note lack of
};

Derived cDerived;
Base &rBase = cDerived;
cout << "rBase is a " << rBase.GetName() << endl;
```

rBase is a Derived

- Only the most base class function needs to be tagged as virtual for all of the derived functions to work virtually.

- However, having the keyword virtual on the derived functions does not hurt,

- and it serves as a useful reminder that the function is a virtual function

- Consequently, it’s generally a good idea to use the virtual keyword for virtualized functions in derived classes even though it’s not strictly necessary.

Return types of virtual functions

- return type of a virtual function and its override must match. Thus, the following will not work

```
class Base{
public:
    virtual int GetValue() { return 5; }
};
class Derived: public Base{
public:
    virtual double GetValue() { return 6.78; }
};
```

```
error C2555: 'Derived::GetValue': overriding virtual function
return type differs and is not covariant from 'Base::GetValue'
```

covariant return types

```
class Base{
public:
    // This version of GetThis() returns a pointer to a Base class
    virtual Base* GetThis() { return this; }
};

class Derived: public Base{
    // because Derived is derived from Base, it's okay to return Derived* instead of Base*
    virtual Derived* GetThis() { return this; }
};
```

some older compilers (eg. Visual Studio 6) do not support covariant return types.

Virtual destructors, virtual assignment, and overriding virtualization

Virtual destructors

- **always** make your destructors virtual if you're dealing with inheritance

```
class Base{public:~Base(){ cout << "Calling ~Base()" << endl;}
};

class Derived: public Base{
    ~Derived() { // note: not virtual
        delete[] m_pnArray;     }
};

Derived *pDerived = new Derived(5);
Base *pBase = pDerived;
delete pBase;
```

Calling ~Base()

Solution: Virtual destructors

```
class Base{public:
      virtual ~Base(){ cout << "Calling ~Base()" << endl;}
};

class Derived: public Base{
     virtual ~Derived() { // note: not virtual
          delete[] m_pnArray;     }
};

Derived *pDerived = new Derived(5);
Base *pBase = pDerived;
delete pBase;
```

Virtual assignment

- virtualizing the assignment operator really opens up a bag full of worms and gets into some advanced topics outside of the scope of this tutorial.

- leave your assignments non-virtual for now, in the interest of simplicity

Overriding virtualization

- Very rarely you may want to override the virtualization of a function

```
int main()
{
    Derived cDerived;
    Base &rBase = cDerived;
    // Calls Base::GetName() instead of the virtualized Derived::GetName()
    cout << rBase.Base::GetName() << endl;
}
```

The downside of virtual functions

- why not just make all functions virtual?

- The answer is because it’s inefficient -- resolving a virtual function call takes longer than a resolving a regular one.

- Furthermore, the compiler also has to allocate an extra pointer for each class object that has one or more virtual functions.

- We’ll talk about this more in the following pages

how virtual functions are implemented

- is not strictly necessary to effectively use virtual functions,
- it is interesting.

- When a C++ program is executed, it executes sequentially,
- beginning at the top of main().
- When a function call is encountered, the point of execution jumps to the beginning of the function being called.

- How does the CPU know to do this?

Binding

- When a program is compiled, the compiler converts each statement in your C++ program into one or more lines of machine language.
- Each line of machine language is given its own unique sequential address.
- Thus, each function has a unique machine language address.
- **Binding** refers to the process that is used to convert identifiers (such as variable and function names) into machine language addresses.
- Although binding is used for both variables and functions, in this lesson we’re going to focus on function binding.

Early binding (also called static binding)

```cpp
void PrintValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    PrintValue(5); // This is a direct function call
    return 0;
}
```

late binding (or dynamic binding)

- Sometimes, it is not possible to know which function will be called until runtime (when the program is run)

```
int Add(int nX, int nY){return nX + nY;}
int Subtract(int nX, int nY){return nX - nY;}
// Create a function pointer named pFcn (yes, the syntax is ugly)
int (*pFcn)(int, int);
// Set pFcn to point to the function the user chose
switch (nOperation)     {
    case 0: pFcn = Add; break;
    case 1: pFcn = Subtract; break;
}
cout << "The answer is: " << pFcn(nX, nY) << endl;
```

virtual table

- To implement virtual functions, C++ uses a special form of late binding known as the virtual table.

```
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() {};
};
```

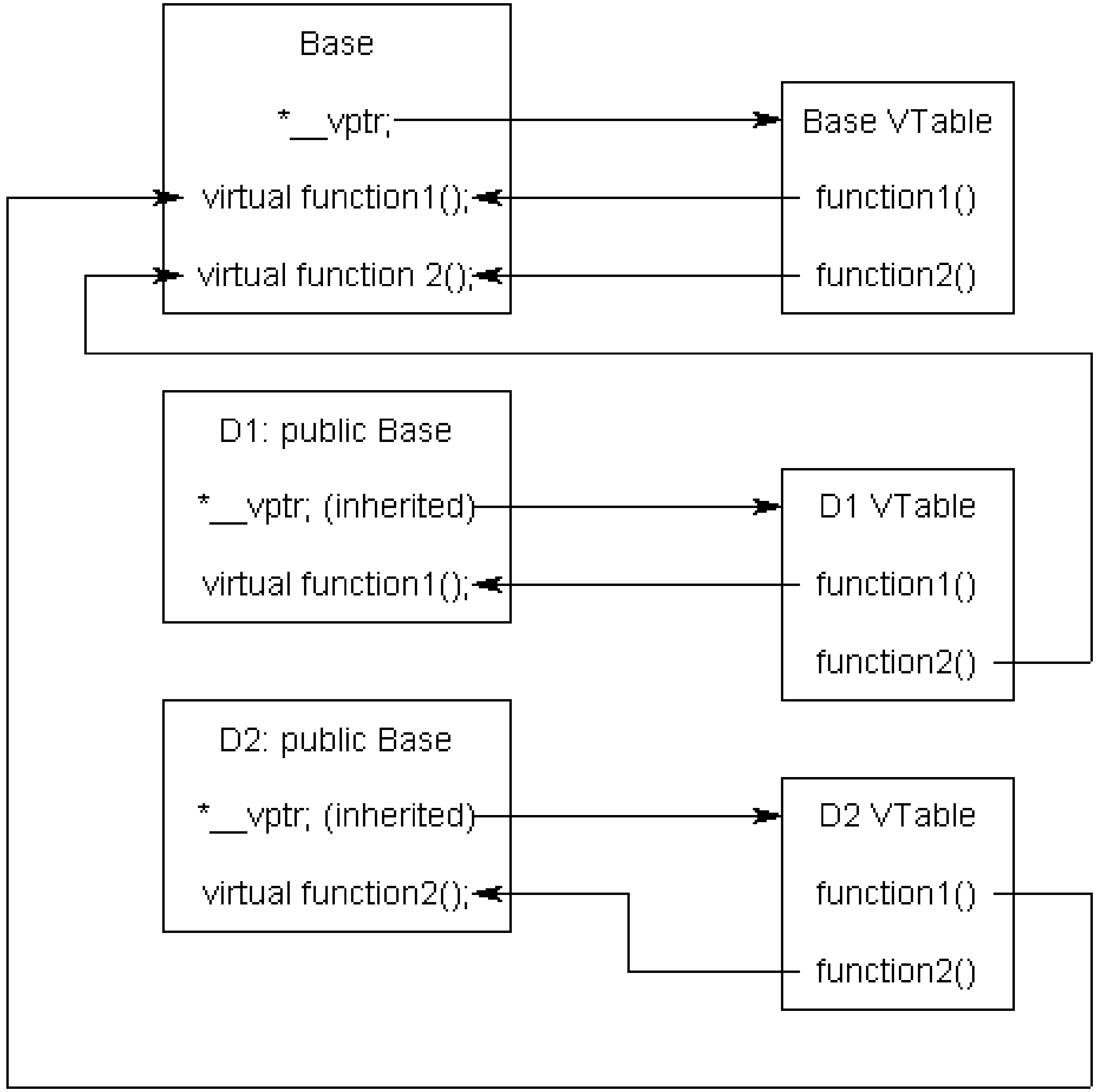
```
class Base{public:
    FunctionPointer *__vptr;
    virtual void function1() {};
    virtual void function2() {};
};

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

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

class D1:virtual void function1()

class D2:virtual void function2()



- Calling a virtual function is slower than calling a non-virtual :
  - First, we have to use the *__vptr to get to the appropriate virtual table.
  - Second, we have to index the virtual table to find the correct function to call.
  - Only then can we call the function.

- As a result, we have to do 3 operations to find the function to call, as opposed to one operation for a direct function call.

- However, with modern computers, this added time is usually fairly insignificant.

Pure virtual functions, abstract base classes, and interface classes

pure virtual function (or abstract function)

```
class Base
{
public:
    const char* SayHi() { return "Hi"; }
    virtual const char* GetName() { return "Base"; }
    virtual int GetValue() = 0; // a pure virtual function
};
```

pure virtual function

two main consequences:

1. any class with one or more pure virtual functions becomes an **abstract base class**, which means that it can **not be instantiated**!

```
int main(){
    Base cBase; // pretend this was legal
    cBase.GetValue(); // what would this do?
}
```

2. any derived class must **define a body** for this function, or that derived class will be considered an abstract base class as well.

Example

```
class Animal{
protected:
    Animal(std::string strName)  : m_strName(strName) { }
public:
    virtual const char* Speak() { return "???"; }
};
```

```
class Dog: public Animal{
public:
    Dog(std::string strName): Animal(strName) { }
    virtual const char* Speak() { return "Woof"; }
};
```

```
class Cat: public Animal{
public:
    Cat(std::string strName): Animal(strName) { }
    virtual const char* Speak() { return "Meow"; }
};
```

- prevented people from allocating objects of type Animal by making the constructor protected.
- It is still possible to create derived classes that do not redefine Speak()

```
class Cow: public Animal{
public:
    Cow(std::string strName) : Animal(strName) {}
    // We forgot to redefine Speak
};

int main(){
    Cow cCow("Betsy");
    cout << cCow.GetName() << " says " << cCow.Speak() << endl;
}
```

A better solution to this problem

```
class Animal{
virtual const char* Speak() = 0; // pure virtual function
};

int main(){
    Cow cCow("Betsy");
    cout << cCow.GetName() << " says " << cCow.Speak() << endl;
}
```

- C:\Test.cpp(141) : error C2259: 'Cow' : cannot instantiate abstract class due to following members: C:Test.cpp(128) : see declaration of 'Cow'
- C:\Test.cpp(141) : warning C4259: 'const char *__thiscall Animal::Speak(void)' : pure virtual function was not defined

Interface classes

- An **interface class** is a class that has no members variables, and where all of the functions are pure virtual!
- In other words, the class is purely a definition, and has no actual implementation.

```
class IErrorLog
{
    virtual bool OpenLog(const char *strFilename) = 0;
    virtual bool CloseLog() = 0;

    virtual bool WriteError(const char *strErrorMessage) = 0;
};
```

FileErrorLog v.s. ScreenErrorLog

```
double MySqrt(double dValue, FileErrorLog &cLog) {
    if (dValue < 0.0)    {
        cLog.WriteError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
        return dValue;
}
```

forces callers of MySqrt() to use a FileErrorLog, which may or may not be what they want.

A much better way

```
double MySqrt(double dValue,  IErrorLog &cLog){
    if (dValue < 0.0)     {
        cLog.WriteError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
        return dValue;
}
```