

COMP322 - Introduction to C++

Lecture 09 - Inheritance continued

Dan Pomerantz

School of Computer Science

11 March 2012

Recall from last time

- ▶ Inheritance describes the creation of derived classes from base classes.
- ▶ Derived classes inherit all data and functions from their base classes, while certain functions such as constructors are handled specially.
- ▶ Polymorphism is a product of the combination of virtual functions and C++ assignment compatibility rules.
- ▶ We can create abstract types which contain “pure” functions that may have no implementation. We can’t actually create objects from these types, but we can create references or pointers to abstract types.
- ▶ A key goal of inheritance: allow us to link related data together. (e.g. Create an array of Shape*)

Basics of inheritance

- ▶ Inheritance is used to denote an *is-a* relationship.
- ▶ e.g. A square IS a shape.
- ▶ If we have a class Square inherit from Shape then all functions and properties of the Shape object are automatically added to the Square object.
- ▶ `public Shape : public Square`

Static dispatch

If a member function is not `virtual`, the choice of function to call is made at *compile* time:

```
class A {
    int f();
};

class B : public A {
    int f();
};

int main() {
    B b;
    A *pa = &b;
    pa->f(); // Calls A::f() because pa is of type 'A *'
}
```

This is called either “static dispatch” or “static binding”, and it is the default behavior in C++.

Dynamic dispatch

If a member function is `virtual`, the choice of function to call is made at *run* time:

```
class A {
    virtual int f();
};

class B : public A {
    int f();
};

int main() {
    B b;
    A *pa = &b;
    pa->f(); // Calls B::f() because pa points to a 'B *'
}
```

Called either “dynamic dispatch” or “run-time binding”, this is both more useful and less efficient than static dispatch.

Dynamic dispatch internals

- ▶ Dynamic dispatch is implemented by adding a layer of indirection to a function call.
- ▶ Any object with dynamic dispatch contains an automatically-generated pointer to a *vtable*.
- ▶ Objects of a given class generally share a vtable.
- ▶ The vtable contains the addresses of the virtual functions.
- ▶ At runtime, the call to the function is performed by indirecting through the vtable pointer.

Virtual functions and constructors/destructors

Calls to virtual functions in the context of the constructor or destructor always use *static* dispatch.

This is different from the behavior of Java, for example.

```
class A {
public:
    A() { cerr << "A()\n"; f(); } // Always calls A::f()!
    ~A() { cerr << "~A()\n"; g(); } // Always calls A::g()!
    virtual void f() { cerr << "A::f()\n"; g(); } // Depends on context
    virtual void g() { cerr << "A::g()\n"; }
};

class B : public A {
public:
    virtual void g() { cerr << "B::g()\n"; }
};

int main() {
    B b;
    A *pa = &b;
    pa->f(); // Calls A::f(), then B::g()!
}
```

Dynamic dispatch from base classes

The prior example hints at an important point: A base class can invoke virtual functions in a derived class, with no knowledge of the derived classes.

```
class base {
public:
    virtual bool vf1(int x) = 0; // Pure virtual
    void f1(int x) {             // Some generic method
        if (vf1(x)) {
            // ...
        }
    }
};

class derived : public base {
public:
    bool vf1(int x) {
        // do something with 'x'...
        return 1;
    }
};
```


Implementing pure virtual functions

A pure virtual function can provide an implementation which could be used by derived classes.

```
class A {  
public:  
    virtual void f() = 0;  
};  
  
void A::f() {  
    // ...  
}  
  
class B : public A {  
public:  
    void f();  
};  
  
void B::f() {  
    A::f(); // call the base class  
    // do more...  
}
```

The compiler will still refuse to create an object of class A!

Multiple inheritance

Java includes the “interface” construct, which allows one to generically specify a group of functions which must be implemented by a derived class.

C++ accomplishes the same thing through abstract classes and *multiple inheritance*.

Multiple inheritance allows a class to be derived from two or *more* base classes. The derived class inherits all of the data and functions of each base class, which clearly raises the possibility of naming conflicts.

Java interfaces are similar to a C++ abstract class with no data or function implementations. Both provide only the prototypes for functions which must be implemented by the derived class.

Multiple inheritance syntax

The syntax of multiple inheritance is straightforward. Each base class can use private, public, or protected inheritance:

```
class A {           // base class 1
    int x;
public:
    void f();
};

class B {           // base class 2
    int y;
public:
    void g();
};

class C : public A, public B { // C inherits from A & B
    int z;           // Visible only within 'C'
public:
    // ..
};
```

Class C will contain both functions and three variables.

Assignment compatibility

A derived class is assignment compatible with *any* base class.

```
class A {           // base class 1
// ...
};

class B {           // base class 2
// ...
};

class C : public A, public B { // C inherits from A & B
// ...
};

int main() {
    A a;
    B b;
    C c;
    a = c; // OK
    b = c; // OK
    c = a; // Error
    return 0;
}
```

Assignment compatibility with pointers

Assignment compatibility with pointers is maintained similarly. However, the conversion to different base classes may return *different* values.

```
class A {          // base class 1
//...
};

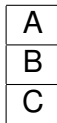
class B {          // base class 2
//...
};

class C : public A, public B { // C is derived from A, B
//...
};

int main() {
    C c;
    A *pa = &c;
    B *pb = &c;
    // In general, (int)pa != (int)pb
    //...
}
```

Assignment compatibility with pointers

Why is this? Consider how the compiler arranges memory for an object of class C. The data for of A, B, and C are concatenated:



When we take the address of such an object, the compiler examines the type of the expression, and returns a pointer to the beginning of the appropriate part of the structure.

Sources of ambiguity

Multiple inheritance can introduce ambiguities and conflicts:

```
class A {  
public:  
    void f();  
};  
  
class B {  
public:  
    void f();  
};  
  
class C : public A, public B {  
public:  
    void g() { f(); } // Which f() do I invoke?  
};
```

This can be resolved by qualifying the call to `f()` as `A::f()`, for example.

Diamond inheritance

An difficult situation arises when one class inherits from two others, both of which share a base class:

```
class person {
    string name;
public:
    void getName();
};

class student : public person {
    int year; // U0, U1, U2, etc..
public:
    void getYear();
};

class employee : public person {
    int year; // years of seniority
public:
    void getYear();
};

class studentemployee : public student, public employee {
};
```


The diamond inheritance problem

The problem is even worse than we might imagine at first glance. Our `getYear()` method is clearly ambiguous.

So is the `getName()` method! Our `student` and `employee` classes both inherited from `person`. Internally, C++ represents derived class as a concatenation of the base and derived classes, so our `studentemployee` class contains two copies of `person`:

| |
|-----------------|
| person |
| student |
| person |
| employee |
| studentemployee |

The diamond inheritance problem

If we use `getName()` in our `studentemployee` class, it could refer to either `person`. This can be overcome with scope resolution:

```
class studentemployee : public student, public employee {
public:
    void f() {
        string s = employee::getName();
    }
};
```

Assignment compatibility is now broken:

```
int main() {
    person p1;
    studentemployee se1;
    p1 = se1; // Which person does the compiler use??
}
```

Again, there is a workaround:

```
p1 = (employee) se1;
```

There are two solutions for diamond inheritance

- ▶ Avoid this situation at all costs!
- ▶ Use virtual inheritance:

```
class person {
    string name;
public:
    void getName();
};
```

```
class student : virtual public person {
    int year; // U0, U1, U2, etc..
public:
    void getYearOfStudy();
};
```

```
class employee : virtual public person {
    int year; // years of seniority
public:
    void getYearsOfService(); // avoid function name conflict
};
```

```
class studentemployee : public student, public employee {
};
```

What is virtual inheritance?

- ▶ Virtual inheritance ensures that a single copy of the common base class is maintained in all derived classes.
- ▶ As with virtual functions and dynamic dispatch, the compiler adds a layer of indirection to accesses to the virtual base class.
- ▶ Virtual inheritance must be anticipated and applied *above* the point where any two classes with a common base class are joined.

```
class A {};  
class B: virtual public A {};  
class C: virtual public A {};  
class D: public B, public C {};
```

- ▶ Rules for virtual inheritance are more complex than we can cover here.

Advanced type casting

The complexity of C++ inheritance has inspired a number of additional type conversion operators.

`dynamic_cast<type>(expression)` - safely converts pointers and references among polymorphic types, with runtime checks.

```
class A { /* ... */ };
class B { /* ... */ };
class C: public A, public B { /* ... */ };

int main() {
    A *pa1 = new A;
    A *pa2 = new C;
    B *pb;
    C *pc;

    pc = dynamic_cast<C *>(pa1); // Returns NULL
    pc = dynamic_cast<C *>(pa2); // OK
    pb = dynamic_cast<B *>(pa2); // OK
}
```

Advanced type casting

`static_cast<type>(expression)` - converts among related classes with static checks. Unlike dynamic cast, it cannot consider the runtime type, it only considers the compile time type.

```
int main() {
    A *pa1 = new A;
    A *pa2 = new C;
    B *pb;
    C *pc;

    pc = static_cast<C *>(pa1); // OK, but dangerous
    pc = static_cast<C *>(pa2); // OK
    pb = static_cast<B *>(pc);   // OK
    pb = static_cast<B *>(pa2); // Compiler error
}
```

Advanced type casting

`reinterpret_cast<type>(expression)` - converts among any pointer types, with no checks or adjustments. This can lead to extremely dangerous situations, as we can convert among completely unrelated types!

```
int main() {
    A *pa1 = new A;
    B *pb;
    C *pc;

    pc = reinterpret_cast<C *>(pa1); // Legal but dangerous
    pb = reinterpret_cast<B *>(pa1); // Legal but dangerous
}
```

A few non-original comments

When designing class 'C', given class 'B', consider which of the following relationships applies:

- ▶ 'C' **is a** 'B' - if it makes sense to think of class 'C' as a specialization of 'B', then 'C' can be implemented as a derived class of 'B'.
 - ▶ In a course system, a 'Student' or 'Professor' is a specialization of 'Person'
- ▶ 'C' **has a** 'B' - on the other hand, 'C' may naturally contain a 'B', but they aren't the same kind of thing.
 - ▶ A 'Person' probably has a 'Address'.
- ▶ 'C' **is implemented as** 'B' - 'C' relies on the services of 'B' in an inessential way. This is one case where private inheritance makes sense.
 - ▶ A stack 'C' may be implemented as a linked list 'B'.