COMP322 - Introduction to C++

Winter 2011

Lecture 08 - Inheritance continued

School of Computer Science

McGill University

March 8, 2011

Last Time

- Single Inheritance
- Polymorphism: Static Binding vs Dynamic Binding
- Abstract Classes

Public inheritance

- The use of the `public` keyword in the declaration of the derived class is the norm, but it must be present.
- One can specify `private` or `protected` inheritance, which hides inherited members.
- If you omit the `public` keyword, inheritance is `private`.

```
class A {
public:  void f();
};

class B: A { // B inherits A privately
public:  void g();
};

int main() {
  A a;
  B b;
  a.f(); // OK
  b.g(); // OK
  b.f(); // Illegal
}
```

Static binding

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 “static binding”, and it is the default behavior in C++.

Dynamic binding

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 binding", this is both more useful and less efficient than static binding.

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;
  //...
}
```

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

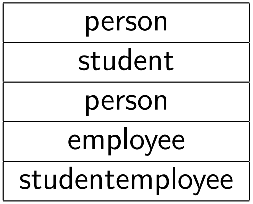
An difficult situation arises when one class inherits from two others, both of which share a base class:

```
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`:



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
}
```

Generic Programming

- Generic Programming is a programming paradigm for developing efficient, reusable software libraries

Introduction to Generic Programming

- Algorithms are written independently of data
  - Data types are filled in during execution
- Number of algorithms are data-type independent:
  - sorting
  - searching
  - finding n-th largest
  - swapping, etc
- Write once, use many times philosophy
  - dramatically reduces lines-of-code
  - makes code much easier to maintain

Templates

- In C++, templates = generic programming
- A template is a routine in which some parameters are qualified by a type variable.
- The template is then specialized for each combination of argument types that occur in practice
- We can define
    - Function Templates
    - Class Templates
- In Java, generic programming is implemented using inheritance: All the collections API classes are written in terms of the Object class, and since all objects in Java are instances of this class, these collections are usable by all class types.

Function Templates

- Special functions that can operate with generic types
- A template parameter is a special kind of parameter used to pass a type as an argument
  - Just like regular function parameters, but pass types to a function
- Declaration Format:

```
template <class identifier> function_declaration;
```

Function Templates - Example

```
template <class T> T getMax(T a, T b){
  return (a>b?a:b);
}

int main(){
  int i=5,j=6,k;
  long l=10,m=5,n;

  k = getMax<int>(i,j);
  n = getMax<long>(l,m);
  cout<< k << endl;
  cout<< n <<endl;

  return 0;
}
```

Note: You cannot mix int and long types: There is one template type only in definition.

Function Templates - Multiple template types

Define template types after the template keyword

```
template <class T1, class T2> T getMax(T1 a, T2 b){
  return (a>b?a:b);
}
```

Class Templates

- Can have members that use template parameters as types

  ```
  template <class T> class myClass{
    private:
    T *storage;
    //...
  };
  ```

- More Next Time