CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 14 – Polymorphism (3)

Announcement

- Reading assignment
  - Ch. 13

Object Slicing

- Happens when a derived class object is assigned to a base class object
- We can avoid above unexpected behavior with the use of pointers or references

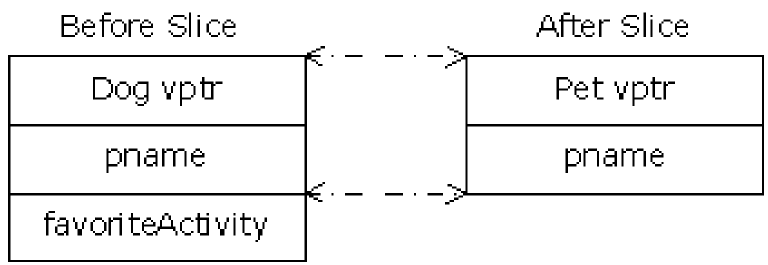
Object Slicing – passing by values

```cpp
////: C15:ObjectSlicing.cpp

class Pet {
  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
};

class Dog : public Pet {
  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
    return Pet::name() + " likes to " +
      favoriteActivity;
  }
};

void describe(Pet p) { // Slices the object
  cout << p.description() << endl;
}
```

Output:

This is Alfred

This is Fluffy




```cpp
int main() {
  Pet p("Alfred");
  Dog d("Fluffy“, “sleep”);
  describe(p);
  describe(d);
}
```

Overriding virtual methods

```
class Base {
 public:
  virtual int f(){ cout << "Base::f()\n"; return 1; }
  virtual void f(string){}
  virtual void g(){}
};

class Derived1 : public Base {
 public:
  void g(){} // ok, only one match for overriding
};

class Derived2 : public Base {
 public:
  int f(){ cout << "Derived2::f()\n"; return 2; } // ok, overriding int Base::f()
};

class Derived3 : public Base {
 public:
  void f(){ cout << "Derived3::f()\n";} // ERROR: return type of Base::f() is different
};

class Derived4 : public Base {
 public:
  int f(int){ cout << "Derived4::f()\n";  return 4; } // Here we are NOT OVERRIDING !
};
```

Overriding virtual methods

- Case where overriding with different return type is ok:

```
class PetFood {
  public:
    virtual string foodType() const = 0;
};

class CatFood : public PetFood {
  public:
    string foodType() const { return "Birds"; }
};

class Pet {
  public:
    virtual string type() const = 0;
    virtual PetFood* eats() = 0;
};

class Cat : public Pet {
  private:
    CatFood cf;
  public:
    string type() const { return "Cat"; }
    CatFood* eats() { return &cf; } // Ok to return CatFood, because it derives from PetFood
};
```

Constructors and destructors

- When an object containing virtual functions is created, its vptr must be initialized to point to the proper VTABLE.
  - This must be done before there’s any possibility of calling a virtual function.
- Default Constructor
  - If you do not provide a default constructor for a class, the compiler will create one for you only for ensuring that the vptr of the object is correctly assigned.
    (no member initialization code is generated)

Constructors and destructors

- Order of Constructor Calls
  - The base class constructor is always called first, before the derived class constructor is called

- Calling virtual methods from a Constructor
  - What happens?
    - Only the local version of the method is called!!
    - the virtual mechanism doesn’t work within the constructor.
    - even if it is virtual, the overridden version is not called. One reason is because the derived class is not yet initialized.

Constructors and destructors

- Virtual Destructors
  - must be used to ensure all classes in a derivation hierarchy are properly destroyed.
  - when delete is called for an object, all its virtual destructors are called starting from the derived class.
  - Forgetting to make a destructor **virtual** can introduce memory leak.

Constructors and destructors

```
// C15:VirtualDestructors.cpp - behavior of virtual vs. non-virtual destructor
#include <iostream>
using namespace std;

class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};
```

Output:

```
~Base1()
~Derived2()
~Base2()
```

```
int main() {
  Base1* bp = new Derived1; // Upcast
  delete bp;
  Base2* b2p = new Derived2; // Upcast
  delete b2p;
}
```

Constructors and destructors

- Calling virtual methods from a Destructor
  - What happens?
    - Only the local version of the method is called !!
    - even if it is virtual, the overridden version is not called. The reason is because, due to the order of destructor calls, the derived classes “are already destroyed”!

Constructors and destructors

- Good rule to follow
  - whenever an object is supposed to derive another object, make its destructor virtual
    - this will ensure correct destruction of all objects in a derivation hierarchy

Wake up!

- https://youtu.be/FM8sIVzvOlw

Operator overloading

- Operators can also be declared virtual and can be overloaded
  - Ex:

```
class Math {
public:
  virtual Math& operator*(Math& rv) = 0;
  virtual Math& multiply(Matrix*) = 0;
  virtual ~Math() {}
};
```

  - Overloaded virtual operators are not commonly used, avoid using them.
  - But simple overloading of operators is very useful, we will cover it later in detail (ch 12)

Downcasting

- It is possible, but it requires an explicit type cast, example:

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){} };
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast ok
  // We know it is a cat, so we can just cast it to Cat*:
  Cat* d2 = (Cat*)(b);
}
```

C-like casts like this can be dangerous since there is no check if the cast is reasonable

Downcasting

- But recall the C++ casting keywords:
  - **static_cast**

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){} };
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast ok
  // We know it is a cat, so we can just cast it to Cat*:
  Cat* d2 = static_cast<Cat*>(b);
}
```

A static_cast will make the compiler test if the two types are on the same hierarchy (better but not really safe)

Downcasting

- **dynamic_cast** provides safer casts:

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){} };
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast
  // Try to cast it to Dog*:
  Dog* d1 = dynamic_cast<Dog*>(b);
  // Try to cast it to Cat*:
  Cat* d2 = dynamic_cast<Cat*>(b);
  cout << "d1 = " << (long)d1 << endl;
  cout << "d2 = " << (long)d2 << endl;
}
```

Output:

d1 = 0

d2 = 7409616

dynamic_cast:
you must be working with
a true polymorphic hierarchy
(one with virtual functions)

A dynamic_cast will return 0 if
the casting is not correct,
so you can check that to
guarantee a safe cast!

Downcasting

- Another way of using run-time type information (RTTI) is with **typeid**.
  - Need to include<typeinfo>

- Again, as with **dynamic_cast**, **typeid** requires a polymorphic object
  - otherwise the local(static) type is returned.

Downcasting

```
class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
     Circle c;
     Shape* s = &c; // Upcast: normal and OK
     s = static_cast<Shape*>(&c); // More explicit but unnecessary

     Circle* cp = 0;
     Square* sp = 0;
     Shape* shapePnter = 0;

     // Example of using typeid():
     if (typeid(s) == typeid(cp)) // C++ RTTI
       cout << "It's a circle!" << endl;
     if (typeid(s) == typeid(sp))
       cout << "It's a square!" << endl;
     if (typeid(s) == typeid(shapePnter))
       cout << "It's a shape!" << endl;
}
```

Static navigation is ONLY an efficiency hack; dynamic_cast is always safer. However:

```
Other* op = static_cast<Other*>(s);
```

Conveniently gives an error message, while

```
Other* op2 = (Other*)s;
```

does not.

Downcasting

- Summary
  - C-like casts are fast but not safe
  - **static_cast** is a bit faster than **dynamic_cast** but will only prevent you from casting out of the hierarchy
  - **dynamic_casts** and **typeid** use RTTI to check for safe casts.
    - but recall they require polymorphic objects (with virtual functions) to be used