CSE 374: Lecture 25

Inheritance

Question Time! Sky's the limit... As on.

Tc 0

Powered by Poll Everywhere

```cpp
namespace student374 {

  class Student {
    std::string name;
    int marks;
  public:
    void getName();
    void getMarks();
    void displayInfo();
  };

  void displayGreeting();

}

namespace teacher374 {
  class Teacher {
    std::string title;
  public:
    Teacher();  // default constructor
    Teacher(std::string name); // parameterized constructor
    Teacher(const Teacher &origt);  // copy constructor

    void getTitle();
    void displayInfo();
    friend std::ostream& operator<<(std::ostream& os, const Teacher& t1);

  };

  void displayGreeting();
}
```

Class layout

A namespace name is mynamespace

```
namespace mynamespace {

    class Rectangle {

        private:
            int width;
            int height;

        public:
            Rectangle();
            Rectangle(int, int);

        public:
            int getArea() {
                return  width * height ;
            }
            int getWidth()  {
                return width;
            }
            int getHeight()   {
                return height;
            }
    };
}
```

Rectangle class in the mynamespace

Field

Rectangle has two field width and height

Constructor

The default Constructor (No parameter)

Constructor with 2 parameters width and height

Method:

Method to calculate the area of this rectangle

Method returns the width of this rectangle

Method returns the height of this rectangle

Access Modifiers

Private: Only accessible inside a class

Protected: Accessible inside a class and all derived classes

Public: Accessible anywhere in a program

Default level of access is PRIVATE

Friends…. Functions or classes that are declared 'friend' in a class can see private and protected members.

Class Constructors (4 types)

- A *default constructor* takes zero arguments. If you don't define any constructors for your class, the compiler will generate one of these constructors for you.
- A *copy constructor* takes a single parameter which is a *const reference* (const T&) to another object of the same type, and initializes the fields of the new object with a COPY of the fields in the referenced object.
- *User-defined constructors* initialize fields and take whatever arguments you like.
- *Conversion constructors* are constructors that take a single argument. For our string example this is like:

```
String(const char* raw);
String s = "foo";
```

Implicit constructors & destructors

Conversion constructors are implicit: automatically applied when a constructor is called with one argument.

If you want a single argument constructor that is not implicit, must use

```
explicit String(const
char* raw);
```

Destructors are used by ‘delete’ to clean up when freeing memory.

```
Virtual ~String();
```

You do not call destructors explicitly

Stack v. Heap

Java: cannot stack-allocate an object (only a pointer to one; all objects are dynamically allocated on the heap - all variables are pointers to objects)

C: can stack-allocate a struct, then initialize it (An actual object)
C++: stack-allocate and call a constructor - Thing t(10000); - is a local copy

Java: new Thing(...) calls constructor, returns heap allocated pointer
C: Use malloc and then initialized, must free exactly once later, untyped pointers
C++: Like Java, new Thing(…), but can also do new int(42). Like C must deallocate, but must use delete instead of free. (never mix malloc/free with new/ delete!)

Subclasses

- **Polymorphism.** In essence, polymorphism is the ability access different objects through the same *interface*. For instance, if you have an interface that represents an electronic device, that interface would have the ability to turn the device on and off. You can use the actual physical types - computer, phone, television, etc - as if they were an electronic device, because they all have the on/off capability.
- **Inheritance.** This is one of the meatiest pieces of OO programming. Inheritance allows the sharing of BEHAVIORS. For instance, a Square is a type of Rectangle, and has the same way to compute its area (width times height) - therefore by make Square inherit from Rectangle, we can share that behavior and avoid duplicating the code.

Subclassing supports fundamental Object Oriented precepts. It may not always be what we want, but understanding OO requires understanding subclassing.

Base-class and Derived-class

- Not all classes have superclasses (unlike Java with Object)
  - (and classes can have multiple superclasses
  - more general and complexity-prone than Java)
- Terminology
  - Java (and others): “superclass” and “subclass”
  - C++ (and others): “base class” and “derived class”
- As in Java, can add fields/methods/constructors, and override methods
  - Also, private, protected, and public methods act as expected

Public Subclassing

Basic:

```
class Foo : public Bar { ... }
```

This is *public inheritance*; C++ has other kinds too (won’t cover)

Differences affect visibility and issues when you have multiple superclasses (won’t cover)

So do not forget the public keyword

Constructing and Destructing

- Constructor of base class gets called before constructor of derived class
  - Default (zero-arg) constructor unless you specify a different one after the : in the constructor
  - Initializer syntax: `Foo::Foo(…): Bar(args); it(x) { … }`
  - Needed to execute base class constructor with arguments; also works on instance variables and is preferred in production code (slogan: “initialization preferred over assignment”)
- Destructor of base class gets called after destructor of derived class
- So constructors/destructors really extend rather than override
  - Typically what you want
  - Java is the same

Mystery.cpp

Constructing and Destructing

- Constructor of base class gets called before constructor of derived class
  - Default (zero-arg) constructor unless you specify a different one after the : in the constructor
  - Initializer syntax: `Foo::Foo(…): Bar(args); it(x) { … }`
  - Needed to execute baseclass constructor with arguments; also works on instance variables and is preferred in production code (slogan: "initialization preferred over assignment")
- Destructor of base class gets called after destructor of derived class
- So constructors/destructors really extend rather than override
  - Typically what you want
  - Java is the same

```
class Derived: public Base {
public:
    double m_cost;
    Derived(double cost=0.0, int id=0)
        : Base{ id }, // Call Base(int) constructor
          m_cost{ cost }  // assign parameter values
    {
    }
    double getCost() const { return m_cost; }
};
```

Mystery.cpp

Method Override

If a derived class defines a method with the <u>same method name and argument types</u> as one defined in the base class, it *overrides* (i.e., replaces) the base class method.

Remember constructors EXTEND, new methods OVERRIDE

If you want to use the base-class code, you specify the base class when making a method call (`base::method(…)`)

Like super in Java (no such keyword in C++ since there may be multiple inheritance)

Virtual methods

Code for class functions is stored in a function table

Look up the functions for a class based on object type

If we want an object to look in the function table for the constructed class, not the variable type(often a base type), we make the function ‘virtual’.

Destructors should always be virtual, so if a derived class is constructed, it is destructed appropriately.

Virtual methods - details

- A non-virtual method-call is resolved using the (compile-time) type of the receiver expression (the datatype it is when the method is called)
- A virtual method-call is resolved using the (run-time) class of the receiver object (the datatype it was constructed as)
    - Called “dynamic dispatch”
- A method-call is virtual if the method called is marked virtual or overrides a virtual method
    - So “one virtual” somewhere up the base-class chain is enough, but it’s probably better style to repeat it

?? So, why are destructors always Virtual?

Pure virtual methods and interfaces (?)

- A C++ “pure virtual” method is like a Java “abstract” method.
  - Subclass must override because there is no definition in base class
- Makes sense with dynamic dispatch
- Funny syntax in base class; override as usual:

```
class C {
    virtual t0 m(t1,t2,...,tn) = 0;
    ...
};
```

- Side-comment: with multiple inheritance and pure-virtual methods, no need for a separate notion of Java-style interfaces

Bank Account ex.

- **Static fields.** A bank account has an associated ID number, which is automatically generated when the account is created. So, we added a static field accountCount_ to the BankAccount class. The "static" keyword means that there is only ONE variable for all objects of this class, not one per object like normal fields. We can use the accountCount_ to generate a unique ID number in the constructor by using that count as the ID and then incrementing the count by 1.
- **Deleted constructors.** Remember C++ automatically generates a "copy constructor" for your class if you do not provide one. However, making copies of the account would be a really bad thing! So we can declare a copy constructor in the header file and set that constructor "= delete;", which means we "delete" it and prevent it from being used anywhere in the code.
- **Pure virtual functions.** A "bank account" is a general concept; if you go into the bank and ask to open a bank account, they'll ask "what kind?" Each type of accounts has a slightly different implementation of withdrawal. All accounts do have the ability to withdraw, but since each type of account has a different implementation, we'll declare the withdraw() function but NOT provide an implementation. We do this as follows by declaring the function "virtual" and setting it equal to 0:

```
virtual void withdraw(int amount) = 0;
```

- This is called a "pure virtual" function, and it make this BankAccount class equivalent to Java's abstract class! Any subclass of BankAccount will have to implement the withdraw() function.

Savings Account

- We provide a constructor that gives more information than just the BankAccount's constructor - a SavingsAccount also generates interest, so we have an interest rate associated with the account.
- The derived class can add additional functions, like getInterestRate().
- The derived class adds an implementation of the withdraw() function from the base class BankAccount. We mark this function "override" so the compiler verifies that we've done the overriding correctly.

(Up) casting

- An object of a derived class *cannot* be cast to an object of a base class.
  - For the same reason a `struct T1 {int x,y,z;}` cannot be cast to type `struct T2 {int x,y;}` (different size)
- A pointer to an object of a derived class *can* be cast to a pointer to an object of a base class.
  - For the same reason a `struct T1*` can be cast to type `struct T2*` (pointers to a location in memory)
  - (Story not so simple with multiple inheritance)
- After such an *upcast*, field-access works fine (prefix)
  - but what do method calls mean in the presence of overriding? (see virtual)

(Down) casting

- C pointer-casts: unchecked; be careful
- Java: checked; may raise ClassCastException
- New: C++ has “all the above” (several different kinds of casts)
  - If you use single-inheritance and know what you are doing, the C-style casts (same pointer, assume more about what is pointed to) should work fine for downcasts
  - Worth learning about the differences on your own