Lecture Participation Poll #22

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

Lecture 22: C++
Inheritance

CSE 374: Intermediate Programming Concepts and Tools
Administrivia

▪ HW 3 posted Friday -> Extra credit due date Wednesday Nov 25th @ 9pm
▪ End of quarter due date Wednesday December 16th @ 9pm
Anatomy of C++ Class

Rectangle.h

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
Rectangle has two field width and height
The default Constructor (No parameter)
Constructor with 2 parameters width and height
Method to calculate the area of this rectangle
Method returns the width of this rectangle
Method returns the height of this rectangle
Access Control

Access modifiers for members:
- public: accessible to all parts of the program
- private: accessible to the member functions of the class
  - Private to class, not object instances
- protected: accessible to member functions of the class and any derived classes (subclasses – more to come, later)

Reminders:
- Access modifiers apply to all members that follow until another access modifier is reached
- If no access modifier is specified, struct members default to public and class members default to private
#ifndef POINT_H_
define POINT_H_

class Point {
    public:
        Point(const int x, const int y); // constructor
        int get_x() const { return x_; } // inline member function
        int get_y() const { return y_; } // inline member function
        double Distance(const Point& p) const; // member function
        void SetLocation(const int x, const int y); // member function

    private:
        int x_; // data member
        int y_; // data member
}; // class Point

#endif // POINT_H_
#include <cmath>
#include "Point.h"

Point::Point(const int x, const int y) {
    x_ = x;
    this->y_ = y; // "this->" is optional unless name conflicts
}

double Point::Distance(const Point& p) const {
    // We can access p’s x_ and y_ variables either through the
    // get_x(), get_y() accessor functions or the x_, y_ private
    // member variables directly, since we’re in a member
    // function of the same class.
    double distance = (x_ - p.get_x()) * (x_ - p.get_x());
    distance += (y_ - p.y_) * (y_ - p.y_);
    return sqrt(distance);
}

void Point::SetLocation(const int x, const int y) {
    x_ = x;
    y_ = y;
}
#include <iostream>
#include "Point.h"

using namespace std;

int main(int argc, char** argv) {
    Point p1(1, 2); // allocate a new Point on the Stack
    Point p2(4, 6); // allocate a new Point on the Stack

    cout << "p1 is: (" << p1.get_x() << ", ");
    cout << p1.get_y() << ")" << endl;

    cout << "p2 is: (" << p2.get_x() << ", ");
    cout << p2.get_y() << ")" << endl;

    cout << "dist : " << p1.Distance(p2) << endl;
    return 0;
}

usePoint.cpp

To allocate on the heap use the “new” keyword
Point* p1 = new Point(1, 2);
Constructors in C++

- A constructor (ctor) initializes a newly-instantiated object
  - A class can have multiple constructors that differ in parameters
  - Which one is invoked depends on how the object is instantiated

- Written with the class name as the method name:

  ```
  Point(const int x, const int y);
  ```

  - C++ will automatically create a synthesized default constructor if you have no user-defined constructors
  - Takes no arguments and calls the default ctor on all non-“plain old data” (non-POD) member variables
  - Synthesized default ctor will fail if you have non-initialized const or reference data members

- 4 different types of constructors
  - default constructor – takes zero arguments. If you don’t define any constructors the compiler will generate one of these for you (just like Java)
  - 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 as a copy of the fields in the referenced object
  - user-defined constructors – initialize fields and take whatever arguments you specify
  - conversion constructors – implicit, take a single argument. If you want a single argument constructor that is not implicit must use the keyword “explicit” like: `explicit String(const char* raw);`
Synthesized Default Constructor

```cpp
class SimplePoint {
public:
    // no constructors declared!
    int get_x() const { return x_; }  // inline member function
    int get_y() const { return y_; }  // inline member function
    double Distance(const SimplePoint& p) const;
    void SetLocation(int x, int y);

private:
    int x_;  // data member
    int y_;  // data member
};  // class SimplePoint
```

```
#include "SimplePoint.h"
...
.... // definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
    SimplePoint x;  // invokes synthesized default constructor
    return EXIT_SUCCESS;
}
```
Synthesized Default Constructor

- If you define any constructors, C++ assumes you have defined all the ones you intend to be available and will not add any others

```cpp
#include "SimplePoint.h"

// defining a constructor with two arguments
SimplePoint::SimplePoint(const int x, const int y) {
    x_ = x;
    y_ = y;
}

void foo() {
    SimplePoint x; // compiler error: if you define any ctors, C++ will NOT synthesize a default constructor for you.

    SimplePoint y(1, 2); // works: invokes the 2-int-arguments constructor
}```
# Overloading Constructors

```cpp
#include "SimplePoint.h"

// default constructor
SimplePoint::SimplePoint() {
    x_ = 0;
    y_ = 0;
}

// constructor with two arguments
SimplePoint::SimplePoint(const int x, const int y) {
    x_ = x;
    y_ = y;
}

void foo() {
    SimplePoint x;       // invokes the default constructor
    SimplePoint y(1, 2); // invokes the 2-int-arguments ctor
    SimplePoint a[3];    // invokes the default ctor 3 times
}
```
Copy Constructors

C++ has the notion of a copy constructor (cctor)
- Used to create a new object as a copy of an existing object
- Initializer lists can also be used in copy constructors
- initializes a new bag of bits (new variable or parameter)
- assignment (=) replaces an existing value with a new one
- may need to clean up old state (free heap data?)

```cpp
Point::Point(const int x, const int y) : x_(x), y_(y) { }

// copy constructor
Point::Point(const Point& copyme) {
    x_ = copyme.x_;  
    y_ = copyme.y_;  
}

void foo() {
    Point x(1, 2);  // invokes the 2-int-arguments constructor

    Point y(x);    // invokes the copy constructor
    Point z = y;   // also invokes the copy constructor
}
```
Synthesized Copy Constructor

- If you don’t define your own copy constructor, C++ will synthesize one for you
  - It will do a shallow copy of all of the fields (i.e. member variables) of your class
  - Sometimes the right thing; sometimes the wrong thing

```cpp
#include "SimplePoint.h"

... // definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
    SimplePoint x;
    SimplePoint y(x); // invokes synthesized copy constructor
    ...
    return EXIT_SUCCESS;
}
```
When Do Copies Happen?

- The copy constructor is invoked if:
  - You *initialize* an object from another object of the same type:
    ```
    Point x;     // default ctor
    Point y(x);  // copy ctor
    Point z = y; // copy ctor
    ```
  - You pass a non-reference object as a value parameter to a function:
    ```
    void foo(Point x) { ... }
    Point y;       // default ctor
    foo(y);        // copy ctor
    ```
  - You return a non-reference object value from a function:
    ```
    Point foo() {
      Point y;   // default ctor
      return y; // copy ctor
    }
    ```
Initialization Lists

- C++ lets you *optionally* declare an initialization list as part of a constructor definition
  - Initializes fields according to parameters in the list
  - The following two are (nearly) identical:

```cpp
Point::Point(const int x, const int y) {
    x_ = x;
    y_ = y;
    std::cout << "Point constructed: (" << x_ << ",";
    std::cout << y_ << ")" << std::endl;
}
```

// constructor with an initialization list
Point::Point(const int x, const int y) : x_(x), y_(y) {
    std::cout << "Point constructed: (" << x_ << ",";
    std::cout << y_ << ")" << std::endl;
}
Initialization vs Construction

- Data members in initializer list are initialized in the order they are defined in the class, not by the initialization list ordering
  - Data members that don’t appear in the initialization list are *default initialized/constructed* before body is executed

- Initialization preferred to assignment to avoid extra steps
  - Never mix the two styles

```cpp
class Point3D {
public:
  // constructor with 3 int arguments
  Point3D(const int x, const int y, const int z) : y_(y), x_(x) {
    z_ = z; // Next, constructor body is executed.
  }

private:
  int x_, y_, z_; // data members
}; // class Point3D
```
Destructors

- C++ has the notion of a destructor (dtor)
  - Like “free” in c. In fact, invokes free under the hood to clean up when freeing memory
  - Invoked automatically when a class instance is deleted, goes out of scope, etc. (even via exceptions or other causes!)
  - Do not need to call destructors explicitly
  - Place to put your cleanup code – free any dynamic storage or other resources owned by the object
  - Standard C++ idiom for managing dynamic resources
    - Slogan: “Resource Acquisition Is Initialization” (RAII)

```cpp
Point::~Point() {  // destructor
    // do any cleanup needed when a Point object goes away
    // (nothing to do here since we have no dynamic resources)
}
```
Nonmember Functions

▪ “Nonmember functions” are just normal functions that happen to use some class
  - Called like a regular function instead of as a member of a class object instance
  - These do not have access to the class’ private members

▪ Useful nonmember functions often included as part of interface to a class
  - Declaration goes in header file, but outside of class definition

▪ A class can give a nonmember function (or class) access to its non-public members by declaring it as a friend within its definition
  - Not a class member, but has access privileges as if it were
  - friend functions are usually unnecessary if your class includes appropriate “getter” public functions

```cpp
class Complex {
  ...
  friend std::istream& operator>>(std::istream& in, Complex& a);
  ...
}; // class Complex

Complex.h

Complex.cpp

std::istream& operator>>(std::istream& in, Complex& a) {
  ...
}
```
Inheritance in C++

- Inheritance is the formal establishment of hierarchical relationships between classes in order to facilitate the sharing of behaviors.

- A parent-child “is-a” relationship between classes
  - A child (derived class) extends a parent (base class)

- Benefits:
  - Code reuse
    - Children can automatically inherit code from parents
  - Polymorphism
    - Ability to redefine existing behavior but preserve the interface
    - Children can override the behavior of the parent
    - Others can make calls on objects without knowing which part of the inheritance tree it is in
  - Extensibility
    - Children can add behavior

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superclass</td>
<td>Base Class</td>
</tr>
<tr>
<td>Subclass</td>
<td>Derived Class</td>
</tr>
</tbody>
</table>
### Inheritance Design Example: Stock Portfolio

- A portfolio represents a person’s financial investments
  - Each asset has a cost (i.e. how much was paid for it) and a market value (i.e. how much it is worth)
  - The difference between the cost and market value is the profit (or loss)
- Different assets compute market value in different ways
  - A stock that you own has a ticker symbol (e.g. “GOOG”), a number of shares, share price paid, and current share price
  - A dividend stock is a stock that also has dividend payments
  - Cash is an asset that never incurs a profit or loss

#### Class Diagram

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Asset</strong></td>
<td>symbol_</td>
<td>GetMarketValue()</td>
</tr>
<tr>
<td></td>
<td>total_shares_</td>
<td>GetProfit()</td>
</tr>
<tr>
<td></td>
<td>total_cost_</td>
<td>GetCost()</td>
</tr>
<tr>
<td></td>
<td>current_price_</td>
<td></td>
</tr>
<tr>
<td><strong>Stock</strong></td>
<td>symbol_</td>
<td>GetMarketValue()</td>
</tr>
<tr>
<td></td>
<td>total_shares_</td>
<td>GetProfit()</td>
</tr>
<tr>
<td></td>
<td>total_cost_</td>
<td>GetCost()</td>
</tr>
<tr>
<td></td>
<td>current_price_</td>
<td></td>
</tr>
<tr>
<td><strong>DividendStock</strong></td>
<td>symbol_</td>
<td>GetMarketValue()</td>
</tr>
<tr>
<td></td>
<td>total_shares_</td>
<td>GetProfit()</td>
</tr>
<tr>
<td></td>
<td>total_cost_</td>
<td>GetCost()</td>
</tr>
<tr>
<td></td>
<td>current_price_</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dividends_</td>
<td></td>
</tr>
<tr>
<td><strong>Cash</strong></td>
<td>amount_</td>
<td>GetMarketValue()</td>
</tr>
</tbody>
</table>

---

**Kasey Champion**
Class Derivation List

- Comma-separated list of classes to inherit from:

```cpp
#include "BaseClass.h"

class Name : public BaseClass {
...
};
```

- Focus on single inheritance, but *multiple inheritance* possible

```cpp
#include "BaseClass.h"
#include "BaseClass2.h"

class Name : public BaseClass, public BaseClass2 {
...
};
```

- Almost always use "public" inheritance
  - Acts like extends does in Java
  - Any member that is non-private in the base class is the same in the derived class; both *interface and implementation inheritance*
    - Except that constructors, destructors, copy constructor, and assignment operator are *never* inherited

- **public**: visible to all other classes
- **protected**: visible to current class and its derived classes
- **private**: visible only to the current class

- Use **protected** for class members only when:
  - Class is designed to be extended by derived classes
  - Derived classes must have access but clients should not be allowed
Inheritance Design Example: Stock Portfolio

A derived class:
- **Inherits** the behavior and state (specification) of the base class
- **Overrides** some of the base class’ member functions (opt.)
- **Extends** the base class with new member functions, variables (opt.)

<table>
<thead>
<tr>
<th>Stock</th>
<th>DividendStock</th>
<th>Stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbol_</td>
<td>dividends_</td>
<td>symbol_</td>
</tr>
<tr>
<td>total_shares_</td>
<td></td>
<td>total_shares_</td>
</tr>
<tr>
<td>total_cost_</td>
<td></td>
<td>total_cost_</td>
</tr>
<tr>
<td>current_price_</td>
<td></td>
<td>current_price_</td>
</tr>
<tr>
<td>GetMarketValue ()</td>
<td></td>
<td>GetMarketValue ()</td>
</tr>
<tr>
<td>GetProfit()</td>
<td></td>
<td>GetProfit()</td>
</tr>
<tr>
<td>GetCost()</td>
<td></td>
<td>GetCost()</td>
</tr>
<tr>
<td>PayDividend()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this diagram, the `DividendStock` class inherits from the `Stock` class and overrides the `PayDividend()` function.
Polymorphism in C++

- **In Java:** `PromisedType var = new ActualType();`
  - `var` is a reference (different term than C++ reference) to an object of `ActualType` on the Heap
  - `ActualType` must be the same class or a subclass of `PromisedType`

- **In C++:** `PromisedType* var_p = new ActualType();`
  - `var_p` is a *pointer* to an object of `ActualType` on the Heap
  - `ActualType` must be the same or a derived class of `PromisedType`
  - (also works with references)
  - `PromisedType` defines the *interface* (i.e. what can be called on `var_p`), but `ActualType` may determine which *version* gets invoked
Questions
RAII

- "Resource Acquisition is Initialization"
- Design pattern at the core of C++
- When you create an object, acquire resources
  - Create = constructor
  - Acquire = allocate (e.g. memory, files)
- When the object is destroyed, release resources
  - Destroy = destructor
  - Release = deallocate
- When used correctly, makes code safer and easier to read

```c
char* return_msg_c() {
  int size = strlen("hello") + 1;
  char* str = malloc(size);
  strncpy(str, "hello", size);
  return str;
}
```

```cpp
std::string return_msg_cpp() {
  std::string str("hello");
  return str;
}
```

```c
using namespace std;
char* s1 = return_msg_c();
cout << s1 << endl;
string s2 = return_msg_cpp();
cout << s2 << endl;
```
The compiler sometimes uses a “return by value optimization” or “move semantics” to eliminate unnecessary copies.
- Sometimes you might not see a constructor get invoked when you might expect it.

```cpp
Point foo() {
    Point y; // default ctor
    return y; // copy ctor? optimized?
}

Point x(1, 2); // two-ints-argument ctor
Point y = x;   // copy ctor
Point z = foo(); // copy ctor? optimized?
```
Namespaces

- Each namespace is a separate scope
  - Useful for avoiding symbol collisions!

- Namespace definition:
  - `namespace name {`
    // declarations go here
  `}`
  - Doesn’t end with a semi-colon and doesn’t add to the indentation of its contents
  - Creates a new namespace name if it did not exist, otherwise *adds to the existing namespace* (!)
    - This means that components (e.g. classes, functions) of a namespace can be defined in multiple source files

- Namespaces vs classes
  - They seem somewhat similar, but classes are *not* namespaces:
  - There are no instances/objects of a namespace; a namespace is just a group of logically-related things (classes, functions, etc.)
  - To access a member of a namespace, you must use the fully qualified name (i.e. `nsp_name::member`)
    - Unless you are using that namespace
    - You only used the fully qualified name of a class member when you are defining it outside of the scope of the class definition
C++ introduces the “const” keyword which declares a value that cannot change.

```
const int CURRENT_YEAR = 2020;
```