**CS 37 Notes Part 6**

**inherit1.cpp**

RAD: Rapid Application Development (ie. using reusable code)

Inheritance: a technique used for RAD

For example, I have written a class with 10 data members and 20 methods and your code is my code with an additional 5 data members and 5 methods. It is ridiculous for you to rewrite all of my code. A much better idea is for you to inherit my class and then add your additional 5 data members and 5 methods to my class.

OUTLINE

Inherit1 consists of two classes, the Point class and the Circle class. The Point class has two data members x and y for the position of the point.

Point::Point(**int** a, **int** b)

{

    x = a;

    y = b;

}

The Circle class is inheriting the Point class and adding a radius data member to its class. Inheritance is done when a class is declared with a colon and the class it is inheriting. *For example, class Circle : public Point (we will cover the public keyword shortly)* Point is called the base class as this is the class being inherited. Circle is called the derived class as this is the class doing the inheriting.

**class** Circle : **public** Point*// circle inherits from point*

*// point is the base class*

*// circle is the derived class*

{

**friend** ostream **&**operator<<( ostream **&**, **const** Circle **&**);

**public:**

    Circle( **double** r = 0.0, **int** x = 0, **int** y = 0);

**double** getRadius() **const**;

**protected:**

**double** radius;

};

We have mentioned the access specifier modes public and private. **With inheritance, there is a third one called protected** which allows access to the data members and methods of the base class only by derived classes or the classes doing the inheriting. In this program, x and y are declared in Point as protected; therefore, x and y can be accessed in the derived class Circle. **Classes that ARE NOT inheriting from Point cannot access the protected data members or methods.**

**protected:**

**int** x,y;*// accessible by classes that inherit the point class*

Inherit1 shows that operator overloading can still be used with inheritance. Both classes contain a stream insertion operator friend function. The stream extraction was not done.

**class** Circle : **public** Point*// circle inherits from point*

*// point is the base class, circle is the derived class*

{

**friend** ostream **&**operator<<( ostream **&**, **const** Circle **&**);

The Point class begins with the friend declaration for the stream insertion operator.

**class** Point

{

**friend** ostream **&**operator<<( ostream **&**, **const** Point **&**);

The constructor and two methods are declared and the data members are declared as protected. The stream insertion operator friend function is given to print the points in the form [x, y].

**class** Point

{

**friend** ostream **&**operator<<( ostream **&**, **const** Point **&**);

**public:**

    Point(**int** = 0, **int** = 0);*// default constructor*

**int** getx() **const** { return x;}

**int** gety() **const** { return y;}

**protected:**

**int** x,y;*// accessible by classes that inherit the point class*

};

&

ostream **&**operator<<( ostream **&**output, **const** Point **&**p)

{

    output << '[' << p.x << ", " << p.y << ']' << endl;

    return output;*// enables cascaded calls*

}

The Circle class begins with class Circle : public Point to show this class is inheriting from the Point class.

**class** Circle : **public** Point*// circle inherits from point*

*// point is the base class*

*// circle is the derived class*

{

The stream insertion friend declaration is

**friend** ostream **&**operator<<( ostream **&**, **const** Circle **&**);

next followed by the declarations for the constructor

**public:**

    Circle( **double** r = 0.0, **int** x = 0, **int** y = 0);

and the getRadius method.

**double** getRadius() **const**;

Radius is declared as a protected data member of the Circle class. A HYPOTHETICAL Cylinder class could then inherit from the Circle class and have access to the radius data member.

**protected:**

**double** radius;

};

**Note that x and y are not declared in the Circle class as they are being inherited from the Point class.**

***How it works***: The code in the constructor works as follows:

The heading line for the constructor is followed by a single : indented on the next line followed by Point(a,b)

Circle::Circle(**double** r, **int** a, **int** b):Point(a,b)*// call base class constructor*

This line will call the Point constructor to load in the values for x and y.

Point::Point(**int** a, **int** b)

{

    x = a;

    y = b;

}

Upon return to the Circle constructor, the radius value is loaded in using a one-line or ternary if statement. *If the value passed is non-negative, it is assigned to the radius, otherwise 0 is assigned to the radius to make sure that radius is not an invalid value.*

Circle::Circle(**double** r, **int** a, **int** b):Point(a,b)*// call base class constructor*

{

    radius = (r >=0 ? r : 0);

}

The Circle stream insertion friend function uses the cast (Point) c which calls the stream insertion friend function of the Point class to place the x and y values of the point into the output stream. Upon return, the radius is then put in the output stream.

ostream **&**operator<<(ostream **&**output, **const** Circle **&**c)

{

    output << "the center of the circle is at " << (Point) c;

    output << endl << "the radius is " << setprecision(2);

    output << fixed << showpoint << c.radius << endl;

*// only outputs 15 without the fixed and showpoint*

    return output; *// RADIUS FOR OUTPUT*

}

The main shows instantiations of both Points and Circles. You can still instantiate a base class object when inheritance is being used.

**inherit.nts** – a file of only comments and no code, nts means notes

We used the line class Circle : public Point to do the inheritance in the last program. The public means that public inheritance is going to be used. One can also specify protected Point or private Point for protected and private inheritance. Public inheritance however is by the far the most used inheritance.

1. **Public** inheritance means anything public in the base class remains public in the derived class, anything protected in the base class remains protected in the derived class, and private data members or private methods cannot be accessed in the derived class.
2. **Protected** inheritance means anything public or protected in the base class becomes protected in the derived class and private data members or private methods cannot be accessed in the derived class.
3. **Private** inheritance means anything public or protected in the base class becomes private in the derived class and private data members or private methods cannot be accessed in the derived class. Private inheritance basically ends the inheritance as no other classes can inherit from the derived class. An analogy is that I tell you some classified data so you are inheriting this data, but, you are not allowed to pass it on to others.

C++ supports single and multiple inheritance.

Single inheritance means a derived class is inheriting from only one base class.

eg class Circle : public Point

Multiple inheritance means a derived class is inheriting from two or more base classes.

eg class Lakers: public Lebron, public Davis, public Rondo

(Some languages do not support multiple inheritance such as Java. You would then need to have class two inherit from class one and then have class three inherit from class two if you want class three to inherit from class one and class two.)

**inherit2.cpp**

|  |  |
| --- | --- |
| Inheritance uses a “is a” relationship. | composition uses a “has a” relationship. |
| eg Seth is a Dork means Seth is inheriting from the dork class. | eg A student has a set of 3 test scores. |
| class Seth: public Dork | Int t1, t2, t3; |

This program uses the fact that a Worker is a Employee to have an Employee base class and a Worker derived class that inherits from the Employee class.

Inheritance: A worker (derived) is an employee (base)

**This program uses dynamic strings.** In the past, we have always declared strings by asking for example “how long should the name be?” 20 with the NULL character means we declare char name[20]; Dynamic strings are better as we will declare exactly the length we need and dynamic strings can be returned to free memory when we are done with them. We will use the new command to allocate memory for the string and we will use the delete command to return the string to free memory. **It is important to return dynamic strings to free memory so that the memory returned can be used for other tasks.**

#include <cstring>

#include<string>

*(As we are dealing with strings, for Studio, you may need to add the line #define \_CRT\_SECURE\_NO\_WARNINGS as the first line of the program.)*

Also, this program shows the order that the base and derived constructors and destructors are called (ie. It’s written chronologically yay) Please notice that for the constructors, the base constructor is called first and the derived constructor is called second. *For the destructors, the derived destructor is called first and the base constructor is called second.* (Destructors are called in reverse order of the constructors.)

*inside Employee constructor*

*inside the Worker constructor*

*Joe Smith hours worked are 10.50*

*wage is 20.00*

*pay is 210.00*

*inside the Worker destructor*

*inside Employee destructor to kill Joe Smith*

*Press any key to continue*

The Employee constructor has two strings passed for the first name and the last name of the employee. The first name is handled first by using the new command to allocate storage for the string. The number of characters to allocate is the length of the first name + 1 for the NULL character. After the new command is done, the strcpy function is used to copy the parameter to the data member. The last name is handled also with a new command and a strcpy.

The Employee destructor uses the delete command to return the string to free memory so that memory space can be used for other processes.

Employee::~Employee()

{

    cout << " inside Employee destructor to kill " << firstName << ' ' << lastName << endl;

    delete[] firstName;

    delete[] lastName;

}

**If you do not use the delete command, even though the strings cannot be accessed outside the class, the strings are still using memory** which is why it is important for the programmer to return their dynamic strings to free memory in the destructor.

Also note that the first and last names are declared as private in the class.

**class** Employee

{

**public:**

    Employee( **const** **char** **\***, **const** **char** **\***);*// constructor*

**void** print() **const**;

    ~Employee();*// destructor*

**private:**

**char** \*firstName;

**char** \*lastName;*// dynamically allocated string*

};

This works for this program as the first and last names are not accessed in the derived class (of class worker). *If these strings were to be accessed in the derived class, then they would have to declared as protected in the base class.*

The Worker class begins with class Worker : public Employee to show that this class is inheriting from the Employee class.

**class** Worker : **public** Employee*// Worker is the derived class*

*// Employee is the base class*

*// inheritance is used for an "is a" relationship.  A worker is an employee*

*// composition is used for a "has a relationship.  A student has a set of test scores*

{

**public:**

    Worker(**const** **char\***, **const** **char\***, **float**, **float**);

**float** getPay() **const**;

**void** print() **const**;

    ~Worker();

**private:**

**float** wage;

**float** hours;

};

The destructor is here just to show you when it is called when the object goes out of scope.

Worker::~Worker() { cout << " inside the Worker destructor " << endl; }

Note the heading for the Worker constructor.

Worker::Worker (**const** **char** \*first, **const** **char** \*last,

**float** initHours, **float** initWage)

                : Employee(first, last)*// call base-class constructor*

{

    hours = initHours;

    wage = initWage;

    cout << "inside the Worker constructor" << endl;

}

On the bottom, there is a colon followed by Employee(first, last). Before the Worker constructor begins executing, the Employee constructor is first called which is why the order of constructors is base and then derived.

Also note in the print method for the Worker class that the first line is Employee::print();

**void** Worker::print() **const**

{

    Employee::print();*// call the base-class print function*

    cout << " hours worked are " << fixed << showpoint

         << setprecision(2) << hours << endl;

    cout << " wage is " << wage << endl;

    cout << " pay is "  << getPay() << endl;

}

This calls the Employee or base class print method to print out the first and last name of the employee.

**void** Employee::print() **const** { cout << firstName << ' ' << lastName; }

*The scope resolution operator :: is used to access outside of the current scope.*

The main instantiates a Worker object and calls the print method of the Worker class as the variable w instantiated is of type Worker.

**int** main()

{

    Worker w("Joe", "Smith", 10.5, 20.0);*// 10.5 hours at $20 per hour*

    w.print();

    return 0;

}