11.2 — Basic inheritance in C++

BY ALEX ON JANUARY 4TH, 2008 | LAST MODIFIED BY ALEX ON DECEMBER 28TH, 2015

Now that we've talked about what inheritance is in an abstract sense, let's talk about how it's used within C++.

Inheritance in C++ takes place between classes. When one class inherits from another, the derived class inherits the variables and functions of the base class. These variables and functions become part of the derived class.

A Person base class

Here's a simple base class:

```
#include <string>
class Person
{
public:
    std::string m_strName;
    int m_nAge;
    bool m_bIsMale;

    std::string GetName() { return m_strName; }
    int GetAge() { return m_nAge; }
    bool IsMale() { return m_bIsMale; }

    Person(std::string strName = "", int nAge = 0, bool bIsMale = false)
        : m_strName(strName), m_nAge(nAge), m_bIsMale(bIsMale)
    {
    }
};
```

This base class is meant to hold information about a person -- in this case, the name, age, and sex. There are two things to note here. First, we have only defined fields that are common to ALL people. This is a generic person class meant to be reused with anybody who is a person. Thus, it's appropriate to only include information used for all people.

Second, note that we've made all of our variables and functions public. This is purely for the sake of keeping these examples simple right now. Normally we would make the variables private. We will cover those cases in future lessons.

A BaseballPlayer derived class

Let's say we wanted to write a program that keeps track of information about some baseball players. Baseball players have information that only people who are baseball players -- for example, we might want to store a player's batting average, and the number of home runs they've hit. Here's our incomplete Baseball player class:

```
class BaseballPlayer
{
public:
    double m_dBattingAverage;
    int m_nHomeRuns;
};
```

Note that we have not included the baseball player's name, age, or sex in this class, even though we want that information. While we could add member variables to hold this information directly to BaseballPlayer, we've already written a generic Person class that we can simply reuse to handle those details.

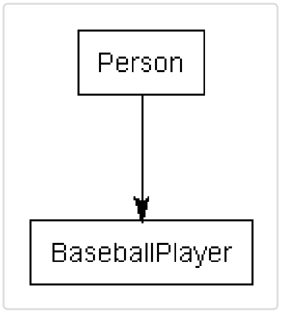
Logically, we know that BaseballPlayer and Person have some sort of relationship. Which makes more sense: a baseball player "has a" person, or a baseball player "is a" person? A baseball player "is a" person, therefore, our baseball player class will use inheritance rather than composition.

To inherit our Person class, the syntax is fairly simple. After the `class BaseballPlayer` declaration, we use a colon, the word "public", and the name of the class we wish to inherit. This is called *public inheritance*. We'll talk more about what public inheritance means in a future section.

```
// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_dBattingAverage;
    int m_nHomeRuns;

    BaseballPlayer(double dBattingAverage = 0.0, int nHomeRuns = 0)
       : m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
    {
    }
};
```

Using a derivation diagram, our inheritance looks like this:



When BaseballPlayer inherits from Person, BaseballPlayer automatically receives the functions and variables from Person. Thus, BaseballPlayer objects will have 5 member variables (m_dBattingAverage and m_nHomeRuns from BaseballPlayer, and m_strName, m_nAge, and m_bIsMale from Person).

This is easy to prove:

```
#include <iostream>
int main()
{
    // Create a new BaseballPlayer object
    BaseballPlayer cJoe;
    // Assign it a name (we can do this directly because m_strName is public)
    cJoe.m_strName = "Joe";
    // Print out the name
    std::cout << cJoe.GetName() << std::endl;

    return 0;
}
```

Which prints the value:

```
Joe
```

This compiles and runs because cJoe is a BaseballPlayer, and all BaseballPlayer objects have a m_strName member variable that they inherit from the Person class.

An Employee derived class

Now let's write another class that also inherits from Person. This time, we'll write an Employee class. An employee "is a" person, so using inheritance is appropriate:

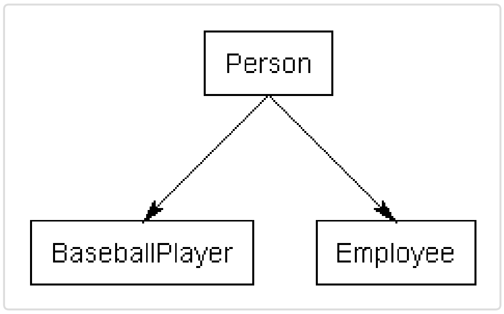
```
// Employee publicly inherits from Person
class Employee: public Person
{
public:
    std::string m_strEmployerName;
    double m_dHourlySalary;
    long m_lEmployeeID;

    Employee(std::string strEmployerName, double dHourlySalary, long lEmployeeID)
        : m_strEmployerName(strEmployerName), m_dHourlySalary(dHourlySalary),
            m_lEmployeeID(lEmployeeID)
    {
    }

    double GetHourlySalary() { return m_dHourlySalary; }
    void PrintNameAndSalary()
    {
        std::cout << m_strName << ": " << m_dHourlySalary << std::endl;
    }
};
```

Employee inherits m_strName, m_nAge, and m_bIsMale from Person (as well as the three access functions), and adds three more member variables and a couple of member function of it's own. Note that PrintNameAndSalary() uses variables both from the class it belongs to (Employee) and the parent class (Person).

This gives us a derivation chart that looks like this:


$$\begin{tikzcd} & \text{Person} \arrow[dl] \arrow[dr] & \\ \text{BaseballPlayer} & & \text{Employee} \end{tikzcd}$$

Note that Employee and BaseballPlayer don't have any direct relationship, even though they both inherit from Person.

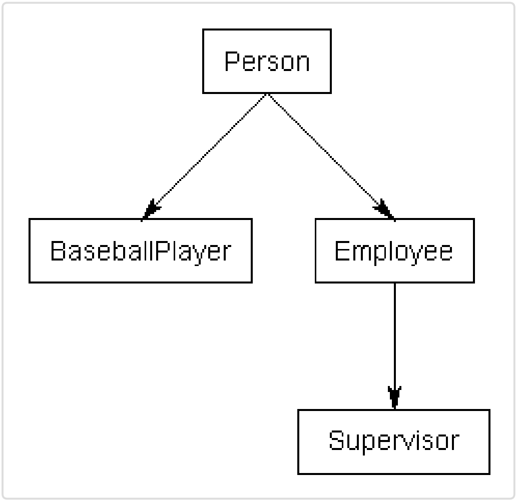
Inheritance chains

It's possible to inherit from a parent that is itself derived from another class. There is nothing noteworthy or special when doing so -- everything proceeds as in the examples above.

For example, let's write a Supervisor class. A supervisor is an employee, which is a person. We've already written an Employee class, so let's use that as the base class from which to derive Supervisor:

```
class Supervisor: public Employee
{
public:
    // This Supervisor can oversee a max of 5 employees
    long m_nOverseesIDs[5];
};
```

Now our derivation chart looks like this:



All Supervisor objects inherit the functions and variables from Employee, and add their own m_nOverseesIDs member variable.

By constructing such inheritance chains, we can create a set of reusable classes that are very general (at the top) and become progressively more specific at each level of inheritance.

Conclusion

Inheriting from a base class means we don't have to redefine the information from the base class in our derived classes. We automatically receive the member functions and member variables of the base class through inheritance, and then simply add the additional functions or member variables we want. This not only saves work, but also means that if we ever update or modify the base class (eg. add new functions, or fix a bug), all of our derived classes will automatically inherit the changes!

For example, if we ever added a new function to Person, both Employee and Supervisor would automatically gain access to it. If we added a new variable to Employee, Supervisor would also gain access to it. This allows us to construct new classes in an easy, intuitive, and low-maintenance way!

11.3 -- Order of construction of derived classes

Index

11.1 -- Introduction to inheritance

Share this:

Email Facebook 3 Twitter Google Pinterest