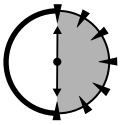
SESSION

Inheritance

Session Checklist

- ✔ Defining inheritance
- ✔ Inheriting a base class
- ✔ Constructing the base class
- ✔ Exploring the IS_A versus the HAS_A relationship



30 Min. To Go

In this Session, we discuss inheritance. *Inheritance* is the capability of one class of things to assume capabilities or properties from another class. For example, I am a human. I inherit from the class *Human* certain properties, such as my ability to converse intelligently (more or less)and my dependence on air, water, and food. These properties are not unique to humans. The class *Human* inherits the dependencies on air, water, and nourishment from the class *Mammal*, of which it is a member.

Advantages of Inheritance

The capability to pass down properties is a powerful one. It allows us to describe things in an economical way. For example, when my son asks "What's a duck?" I can say, "It's a bird that goes quack." Despite what you might think, that answer conveys a considerable amount of information to him. He knows what a bird is, and now he knows all those same things about a duck plus the duck's additional quacking ability.

There are several reasons why inheritance was introduced into C++. Of course, the major reason is the capability to express the inheritance relationship. (I'll return to that in a moment.) A minor reason is to reduce the amount of typing. Suppose we have a class `Student`, and we are asked to add a new class called `GraduateStudent`. Inheritance can drastically reduce the number of things we have to put in the class. All we really need in the class `GraduateStudent` are things that describe the differences between students and graduate students.

A more important, related issue is that major buzzword of the '90s, reuse. Software scientists have realized for some time that it doesn't make much sense to start from scratch with each new project, rebuilding the same software components.

Compare the situation in software to other industries. How many car manufacturers start from ore to build a car? And even if they did, how many would start completely over from ore with the next model? Practitioners in other industries have found it makes more sense to start from screws, bolts, nuts, and even larger off-the-shelf components such as motors and compressors.

Unfortunately, except for very small functions, like those found in the Standard C library, it's rare to find much reuse of software components. One problem is that it's virtually impossible to find a component from an earlier program that does exactly what you want. Generally, these components require "tweaking."

There's a rule of thumb that says, "If you open it, you've broken it." In other words, if you have to modify a function or class to adapt it to a new application, you will have to retest everything, not just the parts you add. Changes can introduce bugs anywhere in existing code. ("The one who last touched it is the one who gets to fix it.")

Inheritance allows existing classes to be adapted to new applications without the need for modification. The existing class is inherited into a new subclass that contains any necessary additions and modifications.

This carries with it a third benefit. Suppose we inherit from some existing class. Later we find that the base class has a bug that must be corrected. If we have modified the class to reuse it, we must manually check for, correct, and retest the bug in each application separately. If we have inherited the class without changes, we can probably adopt the fixed base class without further ado.

Class Factoring

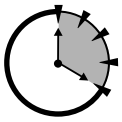
To make sense out of our surroundings, humans build extensive taxonomies. *Fido* is a special case of *dog* which is a special case of *canine* which is a special case of *mammal* and so it goes. This shapes our understanding of the world.

To use another example, a student is a (special type of) person. Having said this, I already know a lot of things about students. I know they have social security numbers, they watch too much TV, they drive a car too fast, and they don't exercise enough. I know all these things because these are properties of all people.

In C++, we call this *inheritance*. We say that the class `Student` *inherits* from the class `Person`. We say also that `Person` is a *base class* of `Student` and `Student` is a *subclass* of `Person`. Finally, we say that a `Student` *IS_A* `Person` (I use all caps as my way of expressing this unique relationship). C++ shares this terminology with other object-oriented languages.

Notice that although `Student` IS_A `Person`, the reverse is not true. A `Person` is not a `Student`. (A statement like this always refers to the general case. It could be that a particular `Person` is, in fact, a `Student`.) A lot of people who are members of class `Person` are not members of class `Student`. This is because the class `Student` has properties it does not share with class `Person`. For example, `Student` has a grade point average, but `Person` does not.

The inheritance property is transitive however. For example, if I define a new class `GraduateStudent` as a subclass of `Student`, `GraduateStudent` must also be `Person`. It has to be that way: if a `GraduateStudent` IS_A `Student` and a `Student` IS_A `Person`, then a `GraduateStudent` IS_A `Person`.



20 Min. To Go

Implementing Inheritance in C++

To demonstrate how to express inheritance in C++, let's return to the `GraduateStudent` example and fill it out with a few example members:

```
// GSInherit - demonstrate how the graduate
//             student class can inherit
//             the properties of a Student
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// Advisor - let's provide an empty class
//           for now
```

```
class Advisor
{
};

// Student - this class includes all types of
//           students
class Student
{
  public:
    Student()
    {
        // start out a clean slate
        pszName = 0;
        dGPA = nSemesterHours = 0;
    }
    ~Student()
    {
        // if there is a name...
        if (pszName != 0)
        {
            // ...then return the buffer
            delete pszName;
            pszName = 0;
        }
    }

    // addCourse - add in the effects of completing
    //             a course by factoring the
    //             dGrade into the GPA
    void addCourse(int nHours, double dGrade)
    {
        // first find the current weighted GPA
        int ndGradeHours = (int)(nSemesterHours * dGPA + dGrade);

        // now factor in the number of hours
        // just completed
        nSemesterHours += nHours;

        // from that calculate the new GPA
        dGPA = ndGradeHours / nSemesterHours;
    }
```

```
    // the following access functions allow
    // the application access to important
    // properties
    int    hours( )
    {
        return nSemesterHours;
    }
    double gpa( )
    {
        return dGPA;
    }

  protected:
    char*  pszName;
    int    nSemesterHours;
    double dGPA;

    // copy constructor - I don't want any
    //                    copies being created
    Student(Student& s)
    {
    }
};

// GraduateStudent - this class is limited to
//                   students who already have a
//                   BA or BS
class GraduateStudent : public Student
{
  public:
    GraduateStudent()
    {
        dQualifierGrade = 2.0;
    }

    double qualifier( )
    {
        return dQualifierGrade;
    }
```

```
  protected:
    // all graduate students have an advisor
    Advisor advisor;

    // the qualifier grade is the
    // grade below which the gradstudent
    // fails the course
    double dQualifierGrade;
};

int main(int nArgc, char* pszArgs[])
{
    // first create a student
    Student llu;

    // now let's create a graduate student
    GraduateStudent gs;

    // the following is perfectly OK
    llu.addCourse(3, 2.5);
    gs.addCourse(3, 3.0);

    // the following is not
    gs.qualifier();       // this is legal
    llu.qualifier();      // but this isn't
    return 0;
}
```

The class `Student` has been declared in the conventional fashion. The declaration for `GraduateStudent`, however, is different from previous declarations. The name of the class followed by the colon followed by public `Student` declares class `GraduateStudent` to be a subclass of `Student`.

The appearance of the keyword `public` implies that there is probably protected inheritance as well. It's true, but I want to hold off discussing this type of inheritance for a moment.

The function `main()` declares two objects, `llu` and `gs`. The object `llu` is a conventional `Student` object, but the object `gs` is something new. As a member of a subclass of `Student`, `gs` can do anything that `llu` can do. It has the data

members pszName, dSemesterHours, and dAverage and the member function addCourse( ). After all, gs quite literally IS_A Student — it's just a little bit more than a Student. (You'll get tired of me reciting this "IS_A" stuff before the book is over.) In fact, GraduateStudent has the qualifier() property which Student does not have.

The next two lines add a course to the two students llu and gs. Remember that gs is also a Student.

The last two lines in main() are incorrect. It is OK to retrieve the qualifier grade of the graduate student gs. It is not OK, however, to try to retrieve the qualifier property of the llu object. The llu object is only a Student and does not share the properties unique to GraduateStudent.

Now consider the following scenario:

```
// fn - performs some operation on a Student
void fn(Student &s)
{
    //whatever fn it wants to do
}

int main(int nArgc, char* pszArgs[])
{
    // create a graduate student...
    GraduateStudent gs;

    // ...now pass it off as a simple student
    fn(gs);
    return 0;
}
```

Notice that the function fn( ) expects to receive as its argument an object of class Student. The call from main( ) passes it an object of class GraduateStudent. However, this is fine because once again (all together now) "a GraduateStudent IS_A Student."

Basically, the same condition arises when invoking a member function of Student with a GraduateStudent object. For example:

```
int main(int nArgc, char* pszArgs[])
{
   GraduateStudent gs;
   gs.addCourse(3, 2.5); //calls Student::addCourse( )
   return 0;
}
```

Constructing a Subclass

Even though a subclass has access to the protected members of the base class and could initialize them in its own constructor, we would like the base class to construct itself. In fact, this is what happens. Before control passes beyond the open brace of the constructor for `GraduateStudent`, control passes to the default constructor of `Student` (because no other constructor was indicated). If `Student` were based on another class, such as `Person`, the constructor for that class would be invoked before the `Student` constructor got control. Like a skyscraper, the object gets constructed starting at the basement class and working its way up the class structure one story at a time.

Just as with member objects, we sometimes need to be able to pass arguments to the base class constructor. We handle this in almost the same way as with member objects, as the following example shows:

```
// Student - this class includes all types of
//           students
class Student
{
  public:
    // constructor - use default argument to
    // create a default constructor as well as
    // the specified constructor type
    Student(char* pszName = 0)
    {
        // start out a clean slate
        this->pszName = 0;
        dGPA = nSemesterHours = 0;

        // if there is a name provided...
        if (pszName != 0)
        {
            this->pszName =
                    new char[strlen(pszName) + 1];
            strcpy(this->pszName, pszName);
        }
    }
    ~Student()
    {
```

```
        // if there is a name...
        if (pszName != 0)
        {
            // ...then return the buffer
            delete pszName;
            pszName = 0;
        }
    }
    // ...remainder of class definition...
};

// GraduateStudent - this class is limited to
//                   students who already have a
//                   BA or BS
class GraduateStudent : public Student
{
  public:
    // constructor - create a Graduate Student
    //               with an advisor, a name and
    //               a qualifier grade
    GraduateStudent(
            Advisor &adv,
            char*    pszName = 0,
            double  dQualifierGrade = 0.0)
                : Student(pName),
                  advisor(adv)
   {
      // executed only after the other constructors
      // have executed
      dQualifierGrade = 0;
   }
  protected:
    // all graduate students have an advisor
    Advisor advisor;

    // the qualifier grade is the
    // grade below which the gradstudent
    // fails the course
    double dQualifierGrade;
};
```

```
void fn(Advisor &advisor)
continued
{
    // sign up our new marriage counselor
    GraduateStudent gs("Marion Haste",
                       advisor,
                       2.0);
    //...whatever this function does...
}
```

Here a `GraduateStudent` object is created with an advisor, the name "Marion Haste" and a qualifier grade of 2.0. The the constructor for `GraduateStudent` invokes the `Student` constructor, passing it the student name. The base class is constructed before any member objects; thus, the constructor for `Student` is called before the constructor for `Advisor`. After the base class has been constructed, the `Advisor` object `advisor` is constructed using the copy constructor. Only then does the constructor for `GraduateStudent` get a shot at it.

The fact that the base class is constructed first has nothing to do with the order of the constructor statements after the colon. The base class would have been constructed before the data member object even if the statement had been written `advisor(adv), Student(pszName)`**. However, it is a good idea to write these clauses in the order in which they are executed just as not to confuse anyone.**

10 Min. To Go

Following our rule that destructors are invoked in the reverse order of the constructors, the destructor for `GraduateStudent` is given control first. After it's given its last full measure of devotion, control passes to the destructor for `Advisor` and then to the destructor for `Student`. If `Student` were based on a class `Person`, the destructor for `Person` would get control after `Student`.

The destructor for the base class `Student` **is executed even though there is no explicit** `~GraduateStudent` **constructor.**

This is logical. The blob of memory which will eventually become a `GraduateStudent` object is first converted to a `Student` object. Then it is the job of the `GraduateStudent` constructor to complete its transformation into a `GraduateStudent`. The destructor simply reverses the process.

Note a few things in this example. First, default arguments have been provided to the `GraduateStudent` constructor in order to pass along this ability to the `Student` base class. Second, arguments can only be defaulted from right to left. The following would not have been legal:

```
GraduateStudent(char* pszName = 0, Advisor& adv)...
```

The non-defaulted arguments must come first.

Notice that the class `GraduateStudent` contains an `Advisor` object within the class. It does not contain a pointer to an `Advisor` object. The latter would have been written:

```
class GraduateStudent : public Student
{
  public:
    GraduateStudent(
            Advisor& adv,
            char*    pszName = 0)
                : Student(pName),
   {
        pAdvisor = new Advisor(adv);
   }
  protected:
    Advisor* pAdvisor;
};
```

Here the base class `Student` is constructed first (as always). The pointer is initialized within the body of the `GraduateStudent` constructor.

The HAS_A Relationship

Notice that the class `GraduateStudent` includes the members of class `Student` and `Advisor`, but in a different way. By defining a data member of class `Advisor`, we know that a `Student` has all the data members of an `Advisor` within it, yet we say that a `GraduateStudent` HAS_A `Advisor`. What's the difference between this and inheritance?

Let's use a car as an example. We could logically define a car as being a subclass of *vehicle*, and so it inherits the properties of other vehicles. At the same time, a car has a motor. If you buy a car, you can logically assume that you are buying a motor as well.

Now if some friends asked you to show up at a rally on Saturday with your vehicle of choice and you came in your car, there would be no complaint because a car IS_A vehicle. But if you appeared on foot carrying a motor, they would have reason to be upset because a motor is not a vehicle. It is missing certain critical properties that vehicles share. It's even missing properties that cars share.

From a programming standpoint, it's just as straightforward. Consider the following:

```
class Vehicle
{
};
class Motor
{
};
class Car : public Vehicle
{
  public:
    Motor motor;
};
void VehicleFn(Vehicle &v);
void motorFn(Motor &m);
int main(int nArgc, char* pszArgs[])
{
   Car c;
   VehicleFn(c);    //this is allowed
   motorFn(c);      //this is not allowed
   motorFn(c.motor);//this is, however
   return 0;
}
```

The call `VehicleFn(c)` is allowed because `c` IS_A `Vehicle`. The call `motorFn(c)` is not because `c` is not a `Motor`, even though it contains a `Motor`. If what was intended was to pass the motor portion of `c` to the function, this must be expressed explicitly, as in the call `motorFn(c.motor)`.

Of course, the call `motorFn(c.motor)` is only allowed if `c.motor` is public.

One further distinction: the class `Car` has access to the protected members of `Vehicle`, but not to the protected members of `Motor`.

Review

Understanding inheritance is critical to understanding the whole point behind object-oriented programming. It's also required in order to understand the next chapter. If you feel you've got it down, move on to Chapter 19. If not, you may want to reread this chapter.

Quiz Yourself

1. What is the relationship between a graduate student and a student? Is it an IS_A or a HAS_A relationship? (See "The HAS_A Relationship.")
2. Name three benefits from including inheritance to the C++ language? (See "Advantages of Inheritance.")
3. Which of the following terms does not fit: *inherits*, *subclass*, *data member* and *IS_A*? (See "Class Factoring.")