CIS 278 (CS1) Programming Methods: C++

Lesson 18: Inheritance

Skip to Main Content

Section 18.1: Introduction to Inheritance

Inheritance is the process by which a new class -- known as a derived class -- is created from an already existing class, called the base class. A derived class automatically has (inherits) all of the member variables and functions that the base class has, and can have additional members.

To illustrate this, let's consider two classes that we have already used: the istream class and the ifstream class. Consider an object of type ifstream named infile:

ifstream infile;
In addition, consider the object cin, which is an object of type istream. These two objects are very similar. In fact they have several members in common: They both have member functions named .get(), .peek(), .ignore(), .putback(), and many others. The infile object, however, has a few members that the cin object does not have. Examples include .open() and .close(). This is because ifstream is derived from istream, and so automatically has all of the member variables and functions that istream has, and has additional members.

There are three ways that classes can be related to each other. The first way is that they can be completely independent. The second way is that one class may have a data member that is an object of the other class. The third way of relating is inheritance.

When you are in the design phase of a project and you need to decide whether two classes should be related by inheritance or by composition, you should ask whether the classes exhibit an "is-a" relationship or a "has-a" relationship. **Composition should be used when the classes exhibit a "has-a" relationship; inheritance is used when the relationship is an "is-a" relationship**. An ifstream variable IS an istream variable; it just happens to be a special kind of istream variable, and so ifstream is derived from istream (using inheritance).

This brings up an important rule of inheritance. C++ takes the "IS-A" relationship very seriously. In fact, **any place in a program where a base class object is expected, you are allowed to place an object of a class that is derived from that base class instead**. This explains why it is that when we overload the extraction operator, we do not use "ifstream" as the type of the first parameter (see the extraction operator in the invItem class of lesson 16 or in the feetInches class of lesson 13 for examples of this). Instead, we use "istream". But C++ still allows us to use an ifstream object as the argument instead of an istream object. This is because C++ considers an ifstream object to BE an istream object, because the ifstream class is derived from the istream class. Just think, if it weren't for inheritance we would have to write two different extraction operators for each of our classes, one with a first parameter of type istream and one with a first parameter of type ifstream.

Section 18.2: The Employee example:

Suppose we want to design two classes, a salaried employee class and an hourly employee class. These two classes are clearly related in some way. Think for a moment about how you would relate them. Would you use inheritance? Composition? The best solution in this case is to create a third class named employee and then have both salaried employee and hourly employee be derived from this class, since clearly a salaried employee is a (type of) employee, and also an hourly employee is a (type of) employee.

Here is an employee class. I have chosen to include data members for the name, ssn, and netpay of employees, since these are data members that are common to all employees, whether they be hourly, salaried, or some other type of employee that I haven't thought of yet. I have also included constructors and some functions to access and modify some of the data members. This class will serve as a base class for both of our other two classes, but there is nothing in this class that you haven't seen before.

```
// this is the file "employee.h"

#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include <string>
using namespace std;

class employee {
    public:
        employee();
        employee(const string& newName, const string& newSsn);
        string getName() const;
        string getSsn() const;
        void changeName(const string& newName);
        void changeSsn(const string& newSsn);
    private:
        string name;
        string ssn;
        double netPay;
};

#endif
```

```
// this is the file "employee.cpp"

#include "employee.h"
#include <string>
using namespace std;

employee::employee()
{
    netPay = 0;
}

employee::employee(const string& newName, const string& newSsn)
{
    name = newName;
    ssn = newSsn;
    netPay = 0;
}

void employee::changeName(const string& newName)
{
    name = newName;
}

void employee::changeSsn(const string& newSsn)
{
    ssn = newSsn;
}

string employee::getName() const
{
    return name;
}

string employee::getSsn() const
{
    return ssn;
}
```

Now let's write the hourly employee class. Let me show it to you first and then I'll explain a couple of things.

```
// this is the file "hourlyemp.h"

#ifndef HOURLYEMP_H
#define HOURLYEMP_H
#include "employee.h"
```

```
#include <string>
using namespace std;

class hourlyEmp: public employee {
    public:
        hourlyEmp();
        hourlyEmp(const string& newName, const string& newSsn,
                      double newPayRate, double newHours);
        double getHours() const;
        void setHours(double newHours);
        void giveRaise(double amount);
        void printCheck() const;
    private:
        double hours;
        double payRate;
};


#endif
```

```
// this is the file "hourlyemp.cpp"

#include "hourlyemp.h"
#include "employee.h"
#include <iostream>
using namespace std;

void hourlyEmp::setHours(double newHours)
{
    hours = newHours;
    netPay = payRate * hours;
}

void hourlyEmp::giveRaise(double amount)
{
    payRate += amount;
    netPay = payRate * hours;
}

double hourlyEmp::getHours() const
{
    return hours;
}

void hourlyEmp::printCheck() const
{
    cout << "Pay " << name << " the sum of " << netPay << " Dollars."
         << endl;
    cout << "Check Stub: ";
    cout << "Employee number: " << ssn << endl;
    cout << "This is an hourly employee.  Hours worked: " << hours
         << endl;
    cout << "Rate: " << payRate << "    Pay: " << netPay << endl;
}

hourlyEmp::hourlyEmp()
{
    payRate = 0;
    hours = 0;
}

hourlyEmp::hourlyEmp(const string& newName,
                     const string& newSsn,
                     double newPayRate,
                     double newHours)

: employee(newName, newSsn)


{
    payRate = newPayRate;
    hours = newHours;
    netPay = hours * payRate;
}
```

I will make four observations about this derived class.

Observation 1: Derived Class Syntax

We want this class to be derived from the employee class. We accomplish this by putting a colon and the words "public employee" in the class declaration between the name of the class and the open curly brace. Remember that when one class is inherited from another the derived class gets all of the members of the base class. The word "public" means that public members of the base class will be public members of the derived class. You could use the word "private" here and make the public members of the base class be private members of the derived class, but this is probably never going to be what you want. For our purposes you can just assume that the word there will always be public. The use of the word private here is so uncommon that other languages (such as Java) assume that classes are always inherited publicly and don't give the programmer the option of making it private.

Observation 2: Updating netPay

Note that each time we change payRate or hours in the hourlyEmployee class we need to update netPay. (This is probably not a great design decision. Having one data member depend on the value of another is error prone. I will fix this in the next round of edits.)

Observation 3: Constructors and Ineritance

Some interesting issues come up with regard to constructors and inheritance. First, unlike other members, constructors are not inherited. However, **each time a derived class constructor is called, the base class default constructor gets called before execution of the derived class constructor begins**. You can think of this as happening after the parameter list in the function header but before the open curly brace. In fact, if you want a base class constructor other than the default constructor to be called, you can put a call to this constructor in this very location, as illustrated in the last function definition in the hourlyEmp class. This call to the base class constructor forms what is called the "initializor list" for this function. The initializor list appears after the parameter list and before the open curly brace in a constructor definition. It is preceded by a single colon. In addition to a call to a base class constructor, the initializor list can also include direct initializations of data members. So, for example, I could rewrite the constructor above like this:

```
hourlyEmp::hourlyEmp(const string& newName,
                     const string& newSsn,
                     double newPayRate,
                     double newHours)

: employee(newName, newSsn),
  payRate(newPayRate),
  hours(newHours),
  netPay(hours * payRate)


{
}
```

Note that in this case the body of the constructor is intentionally left empty, because all of the initializations are done in the initializor list.

There is some disagreement about which of these two methods is best practice. Some authors argue that in a derived class constructor all of the initializations should be done in the initializor list. Others prefer to place only a call to the base class constructor in the initializor list since it is the only thing that is necessary to put there (you can't make an explicit call to a constructor in the body of a function). These authors would recommend doing the rest of the initializations in the body of the constructor. You are free to choose the method you prefer.

Why do we need an initializor list? Can't we just initialize the base class data members directly, in our derived class constructor? Here's the issue. It is always better to have the base class initialize its own data members, rather than having the derived class initialize them, because the base class programmer may know something about how to do this correctly that the derived class programmer doesn't know anything about. The following is an explanation of how the base class and derived class constructors should work together. This might be a bit complex the first time, so make a note to yourself to come back and go over this after you have written a derived class or two.

- The base class DEFAULT constructor is called automatically before a derived class DEFAULT constructor is executed, so that the data members that belong to the base class can be correctly initialized to their default values by the base class DEFAULT constructor.
- Regarding derived class PARAMETERIZED constructors: If you take no other action, the base class DEFAULT constructor will also be automatically called before a derived class PARAMETERIZED constructor begins executing.
- Sometimes this is a good thing, because you want the data members that belong to the base class to be initialized to their default values, and the base class DEFAULT constructor will do this, so the derived class PARAMETERIZED constructor only needs to initialize the data members that it did not inherit from the base class.
- Other times, you want the data members that belong to the base class to be initialized to non-default values (which would be provided by the parameters). This is the case where you use an initializer list to make it so that INSTEAD of automatically calling the base class DEFAULT constructor, you force the base class PARAMETERIZED constructor to be called instead.

Observation 4: Protected

We still have one serious problem. The data members in the employee class are all private. This means that no other class, **not even a derived class**, has access to those data members. But we have repeatedly accessed those data members in our hourly employee class. If we try to run this now, we would get syntax errors telling us of illegal accesses to private members.

The solution to this is to change the word "private" in the base class to the word "protected". The word protected means that the members are private to every class **except** derived classes. Derived classes are granted access to protected members of the base class. I won't show the change to the base class, but I will use the word "protected" instead of "private" in the definition of the salaried employee class shown below, so that you will be able to derive other classes from the salaried employee class. In order to make the code shown in this section work, you'll have to change the word "private" to "protected" in the employee class.

Several students who work with this stuff in software companies have told me that in "real" production code one would never use the "protected" keyword. Instead, one would provide accessors and mutators to allow derived classes to access or mutate the private data members.

For completeness here is the salaried employee class and a client program to test the two classes.

```
// this is the file "salariedemp.h"

#ifndef SALARIEDEMP_H
#define SALARIEDEMP_H
#include "employee.h"
#include <string>
using namespace std;

class salariedEmp : public employee {
    public:
        salariedEmp();
        salariedEmp(const string& newName,
                    const string& newSsn,
                    double newWeeklySalary);
        void printCheck() const;
        void giveRaise(double amount);
    protected:
        double salary;
};
```

```
 #endif
```

```
// this is the file "salariedemp.cpp

#include "salariedemp.h"
#include "employee.h"
#include <iostream>
using namespace std;

void salariedEmp::giveRaise(double amount)
{
    salary += amount;
    netPay += amount;
}

salariedEmp::salariedEmp()
{
    salary = 0;
    netPay = 0;
}

salariedEmp::salariedEmp(const string& newName,
                         const string& newSsn,
                         double newWeeklySalary)

: employee(newName, newSsn)


{
    salary = newWeeklySalary;
    netPay = salary;
}


void salariedEmp::printCheck() const
{
    cout << "pay " << name << endl;
    cout << "the sum of " << netPay << " Dollars." << endl;
    cout << "Check Stub: " << endl;
    cout << "Employee number: " << ssn << endl;
    cout << "This is a salaried employee.  Regular pay: "
         << salary << endl;
}
```

```
// this is the file emptest.cpp

#include <iostream>
#include "hourlyemp.h"
#include "salariedemp.h"
using namespace std;

int main()
{
    hourlyEmp arnold("Arnonld Jones","23456664",13,20);
    arnold.setHours(10);
    cout << "Check for " << arnold.getName() << " for "
         << arnold.getHours() << " hours:" << endl;
    arnold.printCheck();

    cout << endl << endl << endl;

    arnold.giveRaise(5);
    cout << "Check for " << arnold.getName() << " for "
         << arnold.getHours() << " hours:" << endl;
    arnold.printCheck();

    cout << endl << endl << endl;

    salariedEmp sally("Sally Wu", "345123456", 1234.45);
    cout << "Check for " << sally.getName() << endl;
    sally.printCheck();

    cout << endl << endl << endl;

    sally.giveRaise(10);
    cout << endl << endl << endl;
    cout << "after a raise of 10, Sally's check:" << endl;
    sally.printCheck();
}
```

Section 18.3: Virtual Functions

A programming language is required to provide three things in order for it to be considered an object oriented programming language. The first requirement is encapsulation, which is the ability to restrict access to an object's internal properties or, to put it another way, the ability to hide a class's implementation. In C++ this is accomplished by making a the data members of a class private. The second requirement is inheritance. The third requirement is **polymorphism**.

Polymorphism can be loosely defined as the ability to have several different functions with the same name. There are three situations in C++ where we have different functions with the same name. The first is when two functions exist in the same scope with the same name but have different signatures. This technique is called "overloading", a technique that we have used repeatedly in this course. In this course we have overloaded constructors and operators, but in fact any function can be overloaded in C++. The only requirement is that the two functions have different signatures.

The second situation in which we can have different functions with the same name is when a base class and its derived class have functions with the same name. In this case the functions will have exactly the same signature. By default the compiler decides at compile time which of the two functions should be called. (Later we will discuss in detail how this decision is made.) In this case, the technique is called "redefining".

The third situation in which we can have different functions with the same name is similar to the second in that a base class and its derived class have functions with the same name. The difference is that in this third situation C++ decides at run-time (instead of at compile time) which function to call. In this case, the technique is called "overriding".

Usually when someone uses the term "polymorphism", they are referring to this third situation. To summarize: usually the term "polymorphism" refers to the ability to have several different functions with the same name **and** have C++ figure out which one to call at run-time. This is done using a technique called **late binding** or **run-time binding**. Binding is the process of "binding" a function call to a particular function definition. When this binding takes place at compile time it is called **early binding** or **compile time** binding. Languages that support object oriented programming (such as C++) allow this decision to be postponed until runtime. It is probably unclear to you at this point why this would ever make any difference. The rest of this section will be spent demonstrating situations in which it is important to use late-binding instead of early-binding and explaining how to make C++ use late-binding.

Take a close look at the program below.

```
// This program taken from "C++: The Complete Reference"
// by Herbert Schildt

#include <iostream>
using namespace std;

class base {
    public:
        void vfunc() {
            cout << "This is base's vfunc()." << endl;
        }
};


class derived1 : public base {
    public:
        void vfunc() {
            cout << "This is derived1's vfunc()." << endl;
        }
};


class derived2: public base {
    public:
        void vfunc() {
            cout << "This is derived2's vfunc()." << endl;
        }
};

int main()
{
    base *p, b;
    derived1 d1;
    derived2 d2;

    p = &b;
    p -> vfunc();
    // remember, this is equivalent to (*p).vfunc();

    p = &d1;
    p -> vfunc();

    p = &d2;
    p -> vfunc();
}
```

```
OUTPUT:

this is base's vfunc().
this is base's vfunc().
this is base's vfunc().
```

There are three classes in this program. Each of them has only one member, a function that simply prints out a statement telling which function it is. This is so we have a way of knowing which function got called. In the client program we define one object for each class (a base class object, a derived1 class object, and a derived2 class object) and, in addition, a pointer to a base class object. Recall that in C++ we are allowed to use a derived class object anywhere that a base class object was expected; in particular, we can make a variable that has been declared as a pointer to a base class object point to a derived class object.

When p is a pointer to the base class object and we call vfunc(), we naturally expect the base class's vfunc() to be called. When p is a pointer to a derived class object, however, we naturally expect that derived class's vfunc() to be called. As you can see from the output, however, that is not what happens. The problem is that because C++ uses early binding by default, the compiler must decide at compile time which function to call. At compile time the compiler has no way of knowing whether p is pointing to a base class object or to one of the derived class objects. This cannot be determined except by actually letting the program run. So the compiler makes the decision based on the only information it has: p was originally declared as a pointer to a base class object. So the base class's vfunc() is called.

The way to get C++ to use late binding instead of early binding is to put the word "virtual" in front of the function's declaration in the base class. The word virtual should not go on the function's definition (if it is separate from its declaration). It need not go on the function's declaration in the derived class, although it is acceptable to put it there. When the function is declared as a virtual function (so late binding is used), C++ is able to (at runtime) tell what type of object p is pointing to each time vfunc() is called, so the correct version of vfunc() is called. Here is the new program (the only change is the addition of the word "virtual"), along with the new output.

```
#include <iostream>
using namespace std;

class base {
    public:
        virtual void vfunc() {
            cout << "This is base's vfunc()." << endl;
        }
};


class derived1 : public base {
    public:
        void vfunc() {
            cout << "This is derived1's vfunc()." << endl;
        }
};

class derived2: public base {
    public:
        void vfunc() {
            cout << "This is derived2's vfunc()." << endl;
        }
};

int main()
{
    base *p, b;
    derived1 d1;
    derived2 d2;

    p = &b;
    p -> vfunc();
    // remember, this is equivalent to (*p).vfunc();

    p = &d1;
    p -> vfunc();

    p = &d2;
    p -> vfunc();
}
```

```
OUTPUT:

This is base's vfunc().
This is derived1's vfunc().
This is dervied2's vfunc().
```

There are three situations in which it matters whether early binding or late binding is used. The first is when a program has a variable declared as a pointer to a base class object but which sometimes points to derived class objects, as we have just illustrated. The second situation is really the same as the first but it is worth mentioning. It is when a variable is declared as a **reference** to a base class object but sometimes is a reference to a derived class object. The following program illustrates this situation, using the same three classes. The code is shown with the word virtual included, then 2 sets of output are shown: the first is the result of running the program when the word virtual missing, the second the result when the word virtual is present.

```
#include <iostream>
using namespace std;

class base {
    public:
        virtual void vfunc() {
            cout << "This is base's vfunc()." << endl;
        }
};

class derived1 : public base {
    public:
        void vfunc() {
            cout << "This is derived1's vfunc()." << endl;
        }
};

class derived2 : public derived1 {
    public:
        void vfunc() {
            cout << "This is derived2's vfunc()." << endl;
        }
};

void f(base &r);

void f(base &r) {
    r.vfunc();
}

int main()
```

```
{
    base b;
    derived1 d1;
    derived2 d2;

    f(b);
    f(d1);
    f(d2);
}
```

```
OUTPUT WHEN "VIRTUAL" IS NOT USED:

This is base's vfunc().
This is base's vfunc().
This is base's vfunc().
```

```
OUTPUT WHEN "VIRTUAL" IS USED:

This is base's vfunc().
This is derived1's vfunc().
This is dervied2's vfunc().
```

The third situation in which it matters whether early binding or late binding is used is a bit more complicated to understand. It occurs **when a base class function calls a function that is defined in both the base class and a derived class**. The problem in this case is that the compiler cannot determine at compile time whether the base class calling function was called by a base class object (in which case the right thing to do would be to call the base class version of the called function) or by a derived class object (in which case the right thing to do would be to call the derived class version of the called function). The following program (inspired by Walter Savitch in "Problem Solving with C++: The Object of Programming") illustrates this situation. It might be a little confusing because it is kind of a "pretend" graphics program. It has graphics functions that don't actually do graphics, but instead simply print out a statement saying what it would do if it could do graphics. The base class has a function named center that is supposed to center a box or a triangle by calling box::erase() (or triangle::erase()) to first erase the object and then calling box::draw() (or triangle::draw()) to redraw the box or triangle at the center. It won't do us any good if the function calls figure::erase() and figure::draw() because these base class functions do not correctly erase and draw the derived class objects.

Here is the code, followed by the output produced when the word "virtual" is missing, and then the output produced when the word "virtual" is present.

```
#include <iostream>
using namespace std;

class figure {
    public:
        figure()
        {
            cout << "constructing a base class figure object."
                 << endl;
        }

        virtual void draw()
        {
            cout << "drawing a base class figure object." << endl;
        }

        virtual void erase()
        {
            cout << "erasing a base class figure object." << endl;
        }

        void center();
};

void figure::center()
{
    cout << "centering a drawing by first erasing it:" << endl;
    cout << "           ";
```

```
        erase();
        cout <<"then drawing it at the center:" << endl;
        cout << "          ";
        draw();
    }

    class box : public figure {
        public:
            box()
            {
                cout << "constructing a derived class box object."
                     << endl;
            }

            void draw()
            {
                cout << "drawing a derived class box object." << endl;

            }
            void erase()
            {
                cout << "erasing a derived class box object." << endl;
            }
    };


    class triangle : public figure {
        public:
            triangle()
            {
                cout << "constructing a derived class triangle object."
                     << endl;
            }

            void draw()
            {
                cout << "drawing a derived class triangle object." << endl;
            }

            void erase()
            {
                cout << "erasing a derived class triangle object." << endl;
            }
    };


    int main()
    {
        triangle x;
        x.draw();
        cout << "derived class triangle object calling 'center'." << endl;
        x.center();
    }
```

```
    OUTPUT WHEN "VIRTUAL" IS NOT USED (this is bad output because it
    erases and redraws a base class object instead of a triangle object):

    constructing a base class figure object.
    constructing a derived class triangle object.
    drawing a derived class triangle object.
    derived class triangle object calling 'center'.
    centering a drawing by first erasing it:
              erasing a base class figure object.
    then drawing it at the center:
              drawing a base class figure object.
```

```
    OUTPUT WHEN "VIRTUAL" IS USED:

    constructing a base class figure object.
    constructing a derived class triangle object.
    drawing a derived class triangle object.
    derived class triangle object calling 'center'.
    centering a drawing by first erasing it:
              erasing a derived class triangle object.
    then drawing it at the center:
              drawing a derived class triangle object.
```

A final note about virtual functions. Students often ask why we don't just make all of our functions virtual. This is a good question. The only reason to make some functions not virtual is that virtual functions make your code run a little less efficiently, and C++ wants to give the programmer the option of using compile time binding in case small differences in efficiency are a big deal in a particular application. In Java, for example, all functions are considered virtual.

Section 18.4: Pure Virtual Functions

In the example above, we never actually want objects of type figure. We have the figure class so that we will be able to derive other classes from it. So we don't have to write the center() function over and over again. Really what we want to say is: "we don't need this function (draw()) in the base class. But, we're calling a function named draw() from the center() function. So, it is required that any class that is derived from this class must implement a function named draw()." The way we say that in C++ is by adding "= 0" at the end of the draw() prototype, which makes it a "pure virtual function".

When there is at least one pure virtual function in a class, we call the class an abstract class. What that means is that we can't actually have an instance (object) of that class, it's just an abstract class, which means that we are going to have other classes derived from it.

With this change, our base class will look like this:

```
class figure {
    public:
        figure()
        {
            cout << "constructing a base class figure object."
                 << endl;
        }

        virtual void draw() = 0;

        virtual void erase() = 0;

        void center();
};
```

So far I have shown you a lot of things that you can do with inheritance. But it may not be clear exactly when you would want to use this, in what circumstances it's actually going to be useful. With inheritance that is a little hard, because it's not really going to be useful until you've got something complex and big, and all of the examples we've seen so far have been little toy pieces of code just to show you the mechanics of how things work. So, let's look at a complete example of a project that uses inheritance. The following example is from "Data Structures and Other Objects Using C++" by Michael Main and Walter Savitch.

The project we are going to look at has 5 files. The base class is going to be a game class. The idea is that when we want to write a class that represents a game, instead of doing everything from scratch, we can derive it from this game class. If I want to write a class that implements a particular game, I can save myself a lot of trouble by deriving it from this game class. The game class will provide a lot of the functionality that many games exhibit, games that can be represented with some sort of board or playing field, the player take turns with the computer, and you can make a list of the possible moves.

We are going to take a look at a class that is derived from the game class called connect 4. I would suggest taking a few minutes to play at this website: **http://www.knowledgeadventure.com/games/connect-four/**.

What follows is the complete Game project with all 5 required files. Take a couple of hours and look it over carefully until you get a feel for how the inheritance is working. Don't get too bogged down with the details of the eval_with_lookahead() function. The beauty of inheritance is that you can now create a game that includes a look-ahead algorithm just by deriving your class from this game class.

Make sure that you understand the four different categories we have for the base class functions. They are (1) functions that must not be overridden, (2) functions that may be overridden, (3) functions that must be overridden, and must call the base class version of the function when they finish, and (4) functions that must be overridden (pure virtual functions).

The file game.h

```
#ifndef MAIN_SAVITCH_GAME
#define MAIN_SAVITCH_GAME
#include <queue>   // Provides queue
#include <string>  // Provides string

namespace main_savitch_14
{
    class game
    {
    public:
        // ENUM TYPE
    enum who { HUMAN, NEUTRAL, COMPUTER }; // Possible game outcomes

    // CONSTRUCTOR and DESTRUCTOR
    game( ) { move_number = 0; }
    virtual ~game( ) { }

    // PUBLIC MEMBER FUNCTIONS
    // The play function should not be overridden. It plays one game,
    // with the human player moving first and the computer second.
    // The computer uses an alpha-beta look ahead algorithm to select its
    // moves. The return value is the winner of the game (or NEUTRAL for
    // a tie).
    who play( );

    protected:
    // *******************************************************************
    // OPTIONAL VIRTUAL FUNCTIONS (overriding these is optional)
    // *******************************************************************
    virtual void display_message(const std::string& message) const;
    virtual std::string get_user_move( ) const;
    virtual who last_mover( ) const
        { return (move_number % 2 == 1 ? HUMAN : COMPUTER); }
    virtual int moves_completed( ) const { return move_number; }
    virtual who next_mover( ) const
        { return (move_number % 2 == 0 ? HUMAN : COMPUTER); }
    virtual who opposite(who player) const
        { return (player == HUMAN) ? COMPUTER : HUMAN; }
    virtual who winning( ) const;

    // *******************************************************************
    // VIRTUAL FUNCTIONS THAT MUST BE OVERRIDDEN:
    // The overriding function should call the original when it finishes.
    // *******************************************************************
    // Have the next player make a specified move:
        virtual void make_move(const std::string& move) { ++move_number; }
        // Restart the game from the beginning:
        virtual void restart( ) { move_number = 0; }

    // *******************************************************************
        // PURE VIRTUAL FUNCTIONS
    // *******************************************************************
    // (these must be provided for each derived class)
        // Return a pointer to a copy of myself:
        virtual game* clone( ) const = 0;
        // Compute all the moves that the next player can make:
        virtual void compute_moves(std::queue<std::string>& moves) const = 0;
        // Display the status of the current game:
        virtual void display_status( ) const = 0;
        // Evaluate a board position:
    // NOTE: positive values are good for the computer.
        virtual int evaluate( ) const = 0;
        // Return true if the current game is finished:
        virtual bool is_game_over( ) const = 0;
        // Return true if the given move is legal for the next player:
        virtual bool is_legal(const std::string& move) const = 0;

    private:
        // MEMBER VARIABLES
    int move_number;                     // Number of moves made so far

    // STATIC MEMBER CONSTANT
```

```
        static const int SEARCH_LEVELS = 4;  // Levels for look-ahead evaluation

        // PRIVATE FUNCTIONS (these are the same for every game)
        int eval_with_lookahead(int look_ahead, int beat_this);
        void make_computer_move( );
        void make_human_move( );
    };
}

#endif
```

The file game.cpp

```
#include <cassert>    // Provides assert
#include <climits>    // Provides INT_MAX and INT_MIN
#include <iostream>   // Provides cin, cout
#include <queue>      // Provides queue<string>
#include <string>     // Provides string
#include "game.h"     // Provides definition of game class
using namespace std;

namespace main_savitch_14
{
    //*************************************************************************
    // STATIC MEMBER CONSTANTS
    const int game::SEARCH_LEVELS;

    //*************************************************************************
    // PUBLIC MEMBER FUNCTIONS

    game::who game::play( )
    // The play function should not be overridden. It plays one round of the
    // game, with the human player moving first and the computer second.
    // The return value is the winner of the game (or NEUTRAL for a tie).
    {
    restart( );

    while (!is_game_over( ))
    {
        display_status( );
        if (last_mover( ) == COMPUTER)
        make_human_move( );
        else
        make_computer_move( );
    }
    display_status( );
    return winning( );
    }



    //*************************************************************************
    // OPTIONAL VIRTUAL FUNCTIONS (overriding these functions is optional)

    void game::display_message(const string& message) const
    {
    cout << message;
    }

    string game::get_user_move( ) const
    {
    string answer;

    display_message("Your move, please: ");
    getline(cin, answer);
    return answer;
    }

    game::who game::winning( ) const
    {
    int value = evaluate( ); // Evaluate based on move that was just made.

    if (value > 0)
        return COMPUTER;
    else if (value < 0)
        return HUMAN;
    else
        return NEUTRAL;
```

```
}


//***********************************************************************
// PRIVATE FUNCTIONS (these are the same for every game)

int game::eval_with_lookahead(int look_ahead, int beat_this)
// Evaluate a board position with lookahead.
// --int look_aheads:  How deep the lookahead should go to evaluate the move.
// --int beat_this: Value of another move that weíre considering. If the
// current board position can't beat this, then cut it short.
// The return value is large if the position is good for the player who just
// moved.
{
    queue<string> moves;   // All possible opponent moves
int value;             // Value of a board position after opponent moves
    int best_value;        // Evaluation of best opponent move
    game* future;          // Pointer to a future version of this game

    // Base case:
if (look_ahead == 0 || is_game_over( ))
{
    if (last_mover( ) == COMPUTER)
            return evaluate( );
    else
    return -evaluate( );
}

    // Recursive case:
    // The level is above 0, so try all possible opponent moves. Keep the
// value of the best of these moves from the opponent's perspective.
    compute_moves(moves);
assert(!moves.empty( ));
    best_value = INT_MIN;
    while (!moves.empty( ))
    {
    future = clone( );
    future->make_move(moves.front( ));
    value = future->eval_with_lookahead(look_ahead-1, best_value);
    delete future;
    if (value > best_value)
    {
    if (-value <= beat_this)
        return INT_MIN + 1; // Alpha-beta pruning
    best_value = value;
    }
    moves.pop( );
    }

    // The value was calculated from the opponent's perspective.
    // The answer we return should be from player's perspective, so multiply times -1:
    return -best_value;
}

void game::make_computer_move( )
{
queue<string> moves;
int value;
int best_value;
string best_move;
game* future;

// Compute all legal moves that the computer could make.
compute_moves(moves);
assert(!moves.empty( ));

// Evaluate each possible legal move, saving the index of the best
// in best_index and saving its value in best_value.
best_value = INT_MIN;
while (!moves.empty( ))
{
    future = clone( );
    future->make_move(moves.front( ));
    value = future->eval_with_lookahead(SEARCH_LEVELS, best_value);
    delete future;
    if (value >= best_value)
    {
    best_value = value;
    best_move = moves.front( );
    }
    moves.pop( );
}
```

```
    // Make the best move.
    make_move(best_move);
    }

    void game::make_human_move( )
    {
        string move;

    move = get_user_move( );
    while (!is_legal(move))
    {
        display_message("Illegal move.\n");
        move = get_user_move( );
        }
    make_move(move);
    }

}
```

The file connect4.h

```
// File: connect4.h (part of the namespace main_savitch_14)
#ifndef MAIN_SAVITCH_CONNECT4
#define MAIN_SAVITCH_CONNECT4
#include <queue>    // Provides queue<string>
#include <string>   // Provides string
#include "game.h"   // Provides the game base class

namespace main_savitch_14
{
    class connect4 : public game
    {
    public:
    // STATIC CONSTANTS
    static const int ROWS = 6;
    static const int COLUMNS = 7;

    // CONSTRUCTOR
    connect4( );

    protected:
    // *******************************************************************
    // VIRTUAL FUNCTIONS (these are overridden from the game base class)
    // *******************************************************************
    // Return a pointer to a copy of myself:
    virtual game* clone( ) const;
        // Compute all the moves that the next player can make:
    virtual void compute_moves(std::queue<std::string>& moves) const;
    // Display the status of the current game:
    virtual void display_status( ) const;
    // Evaluate the current board position.
    // NOTE: Positive values are good for the computer.
    virtual int evaluate( ) const;
    // Return true if the current game is finished:
    virtual bool is_game_over( ) const;
    // Return true if the given move is legal for the next player:
    virtual bool is_legal(const std::string& move) const;
    // Have the next player make a specified move:
    virtual void make_move(const std::string& move);
    // Restart the game from the beginning:
    virtual void restart( );

    private:
    // HELPER FUNCTIONS
    int value(int row, int column, int delta_r, int delta_c) const;

    // HELPER CONSTANTS. See connect4.cxx for their values.
    static const int COLUMN_DISPLAY_WIDTH;
    static const int FOUR_VALUE;
    static const std::string MOVE_STRINGS[COLUMNS];

        // MEMBER VARIABLES TO TRACK THE STATE OF THE GAME
    who data[ROWS][COLUMNS];
    int many_used[COLUMNS];
    int most_recent_column;
    };
}

#endif
```

The file connect4.cpp

```
// File: connect4.cxx


#include <algorithm>  // Provides fill_n
#include <cassert>    // Provides assert macro
#include <cctype>     // Provides isdigit
#include <iomanip>    // Provides setw
#include <iostream>   // Provides cin, cout
#include <queue>      // Provides queue<string> class
#include <string>     // Provides string
#include "connect4.h" // Provides definition of connect4 class (derived from game)
using namespace std;

namespace main_savitch_14
{
    // Public static member constants. These were defined in connect.h:
    const int connect4::ROWS;
    const int connect4::COLUMNS;

    // Private static member constants, defined here. The COLUMN_DISPLAY_WIDTH
    // is the number of characters to use in the display( ) function for each
    // column in the output display. The FOUR_VALUE is the value returned by
    // the value function when it finds four-in-a-row. For the current
    // implementation of evaluate to work, the FOUR_VALUE should be at least
    // 24 times as large as the total number of spots on the board.
    // MOVE_STRINGS is an array of all possible moves, which must be strings
    // corresponding to the integers 0 through COLUMNS-1.
    const int connect4::COLUMN_DISPLAY_WIDTH = 3;
    const int connect4::FOUR_VALUE = 24*ROWS*COLUMNS;
    const string connect4::MOVE_STRINGS[COLUMNS] = {"0","1","2","3","4","5","6"};

    connect4::connect4( )
    : game( )
    {
    restart( );
    }

    game* connect4::clone( ) const
    {
    // Return a pointer to a copy of myself, made with the automatic copy
    // constructor. If I used dynamic memory, I would have to alter this
    // copy so that it used its own dynamic memory instead of mine. But
    // the connect4 class does not use any dynamic memory, so this is ok:
    return new connect4(*this);
    }

    void connect4::compute_moves(queue<string>& moves) const
    {
    int i;

    for (i = 0; i < COLUMNS; i++)
    {
        if (many_used[i] < ROWS)
        moves.push(MOVE_STRINGS[i]);
    }
    }

    void connect4::display_status( ) const
    {
    int row, column;

    cout << "\nCurrent game status\n(HUMAN = #  and  COMPUTER = @):\n";
    if (moves_completed( ) != 0)
        cout << "Most recent move in column " << most_recent_column << endl;
    for (row = ROWS-1; row >= 0; --row)
    {
        for (column = 0; column < COLUMNS; ++column)
        {
        switch (data[row][column])
        {
        case HUMAN:    cout << setw(COLUMN_DISPLAY_WIDTH) << "#"; break;
        case COMPUTER: cout << setw(COLUMN_DISPLAY_WIDTH) << "@"; break;
        case NEUTRAL:  cout << setw(COLUMN_DISPLAY_WIDTH) << "."; break;
        }
        }
        cout << endl;
    }
    for (column = 0; column < COLUMNS; ++column)
        cout << setw(COLUMN_DISPLAY_WIDTH) << column;
    if (is_game_over( ))
        cout << "\nGame over." << endl;
    else if (last_mover( ) == COMPUTER)
        cout << "\nHuman's turn to move (by typing a column number)" << endl;
    else
        cout << "\nComputer's turn to move (please wait)..." << endl;
    }

    int connect4::evaluate( ) const
    {
    // NOTE: Positive answer is good for the computer.
    int row, column;
    int answer = 0;

        // For each possible starting spot, calculate the value of the spot for
    // a potential four-in-a-row, heading down, left, and to the lower-left.
    // Normally, this value is in the range [-3...+3], but if a
    // four-in-a-row is found for the player, the result is FOUR_VALUE which
    // is large enough to make the total answer larger than any evaluation
    // that occurs with no four-in-a-row.

    // Value moving down from each spot:
    for (row = 3; row < ROWS; ++row)
        for (column = 0; column < COLUMNS; ++column)
        answer += value(row, column, -1, 0);

    // Value moving left from each spot:
    for (row = 0; row < ROWS; ++row)
        for (column = 3; column < COLUMNS; ++column)
        answer += value(row, column, 0, -1);

    // Value heading diagonal (lower-left) from each spot:
    for (row = 3; row < ROWS; ++row)
        for (column = 3; column < COLUMNS; ++column)
        answer += value(row, column, -1, -1);

    // Value heading diagonal (lower-right) from each spot:
    for (row = 3; row < ROWS; ++row)
        for (column = 0; column <= COLUMNS-4; ++column)
        answer += value(row, column, -1, +1);

    return answer;
    }

    bool connect4::is_game_over( ) const
    {

    int row, column;
    int i;

    // Two simple cases:
    if (moves_completed( ) == 0)
        return false;
    if (moves_completed( ) == ROWS*COLUMNS)
        return true;

    // Check whether most recent move is part of a four-in-a-row
    // for the player who just moved.
    column = most_recent_column;
    row = many_used[column] - 1;
    // Vertical:
    if (abs(value(row, column, -1, 0)) == FOUR_VALUE) return true;
    for (i = 0; i <= 3; i++)
    {
        // Diagonal to the upper-right:
        if (abs(value(row-i, column-i, 1,  1)) == FOUR_VALUE) return true;
        // Diagonal to the lower-right:
          if (abs(value(row-i, column+i, 1, -1)) == FOUR_VALUE) return true;
        // Horizontal:
        if (abs(value(row,   column-i, 0,  1)) == FOUR_VALUE) return true;
    }

    return false;
    }

    bool connect4::is_legal(const string& move) const
    {
    int column = atoi(move.c_str( ));

    return
        (!is_game_over( ))
        &&
        (move.length( ) > 0)
        &&
        (isdigit(move[0]))
        &&
        (column < COLUMNS)
        &&
        (many_used[column] < ROWS);
    }

    void connect4::make_move(const string& move)
    {
    int row, column;

    assert(is_legal(move));
    column = atoi(move.c_str( ));
    row = many_used[column]++;
    data[row][column] = next_mover( );
    most_recent_column = column;
    game::make_move(move);
    }

    void connect4::restart( )
    {
    fill_n(&(data[0][0]), ROWS*COLUMNS, NEUTRAL);
    fill_n(&(many_used[0]), COLUMNS, 0);
    game::restart( );
    }

    int connect4::value(int row, int column, int delta_r, int delta_c) const
    {
    // NOTE: Positive return value is good for the computer.
    int i;
    int end_row = row + 3*delta_r;
    int end_column = column + 3*delta_c;
    int player_count= 0;
    int opponent_count = 0;

    if (
        (row < 0) || (column < 0) || (end_row < 0) || (end_column < 0)
        ||
        (row >= ROWS) || (end_row >= ROWS)
        ||
        (column >= COLUMNS) || (end_column >= COLUMNS)
       )
        return 0;

    for (i = 1; i <= 4; ++i)
    {
        if (data[row][column] == COMPUTER)
        ++player_count;
        else if (data[row][column] != NEUTRAL)
        ++opponent_count;
        row += delta_r;
        column += delta_c;
    }

    if ((player_count > 0) && (opponent_count > 0))
        return 0; // Neither player can get four-in-a-row here.
    else if (player_count == 4)
        return FOUR_VALUE;
    else if (opponent_count == 4)
        return -FOUR_VALUE;
    else
        return player_count - opponent_count;
    }
}
```

The Client File, play_connect4.cpp

```
#include <cctype>
#include <cstdlib>
#include <iostream>
#include <string>
#include "connect4.h"
using namespace std;
using namespace main_savitch_14;

int main( )
```

```
{
    connect4 instance;
    connect4::who winner;
    char answer;
    string restline;

    do
    {
    winner = instance.play( );
    switch (winner)
    {
    case connect4::HUMAN:    cout << "You win" << endl; break;
    case connect4::COMPUTER: cout << "I win"   << endl; break;
    case connect4::NEUTRAL:  cout << "A draw"  << endl; break;
    }
    cout << "Do you want to play again? [Y/N] ";
    cin >> answer;
    getline(cin, restline);
    }   while (toupper(answer) == 'Y');

    return EXIT_SUCCESS;
}
```