CIS 278 (CS1) Programming Methods: C++

Lesson 15: Classes

Skip to Main Content

Section 15.1: Introduction to Classes

The concept of a `class` in C++ is attached to several Computer Science concepts that are more theoretical in nature such as abstract data types, encapsulation, and object oriented programming. The concepts are important and you should understand them before we are done. However, I think that a better way to learn about `classes` is to start from the concrete and move to the abstract, rather than the other way around. So in this lesson I'm going to show you a `class`. You'll need to make sure and read about the more theoretical concepts in chapter 11 of the text.

In order to do this, I'm going to present `classes` simply as a way for a programmer to define a new type. Until now, every type that we have used has been a pre-defined C++ type. Now we are going to discuss how programmers can define their own types and then use them. For example, suppose that a group of programmers at a software firm are working on a project that involves a great deal of date processing. Rather than having three variables to represent a single date (a month, a day, and a year), it would be convenient to have a data type called `Date` which would, with a single variable, represent a date. Once we have this new data type defined, we could declare variables of type `Date` like this:

```
Date date1;
Date date2;
```

Each Date variable would store all of the information necessary to represent a single date. The group leader might assign one of the group members the task of defining the new type. That programmer would use a `class` to do this.

When we declare a variable of type `Date` (or any other class), the variable is called an **object**. **Object** is a loaded term in C++, but don't let it confuse you. An object is simply a special kind of variable. If you see or hear the word "object" and you start to feel panicky because you aren't sure you really understand objects, try just substituting the word "variable". Most of the time that will work fine.

To put it another way: A **class** is a special kind of type. An **object** is a special kind of variable whose type is a class instead of a simple type.

Let me digress for a moment and point out that we have actually been using classes for a long time already. Although classes can be defined by programmers in situations like the one described above, some classes come pre-defined. For example, the `string` data type that we have been using since the first week of the semester is actually not a simple data-type, but is a pre-defined `class`. Here's how you can tell the difference: You may have noticed that the syntax of using `string` objects (remember: objects == variables) is a bit different than it is for the other data types we have used. For example, when you declare a `string` object you can then call a function that operates on that object by putting a dot between the `string` object and the name of the function. The third line below is an example of this:

```
string str1;
str1 = "hi there";
cout << str1.substr(1,3);
```

Clearly you couldn't do anything like this with an `int` variable or a `double` variable. That's because `string` is not a normal data-type. Rather, it is a `class`. When we call a function that is a component of an object like this, we call the object (`str1` in this case) the **calling object** and we say that `str1` **calls** the `substr()` function. In general the "**dot operator**" is used to refer to a particular component of an object. The function call illustrated above means "call the `substr()` function which is a component of the `str1` object."

When we use a class in a program we say that the program we are writing is a **client** of the class. So it would be technically correct to say that any program you have written that uses the string class is a client program. Usually, however, this term is used when we need to distinguish between a class and the program which uses the class. In this case we would call the class a **class definition** and we would call the program which uses the class the **client program**.

Sometimes students have a hard time conceptualizing classes and objects. Here is an analogy to help you think about classes and objects, in case it helps. If it doesn't help, that's ok, you can ignore it for now. Classes are like a blue print, and objects are like individual houses built according to the blue print. Each of these houses is an "instance" of the blue print. Each one has it's own bathrooms and bedrooms, but they all have the same number of bedrooms and bathrooms as the blue print. If you paint the master bedroom of house1 pink, the master bedroom of house2 stays the same.

Section 15.2: Declaring a Class

When we declare a `class`, we start by listing all of the components that objects of that class will have. Consider the Date example again. In order to store a date we must store three things: an integer for the month, an integer for the day, and an integer for the year. (This is the most intuitive way to store a date. Actually we could store dates using a number of different methods. For example, we could store the month in a string, or we could store the entire date in a single string, or we could represent the date as the number of days that have passed since January 1, 1600. We'll stick to the more intuitive solution of storing three integers.)

In addition to these three pieces of data, a Date object should also have functions that it can call, like `string` objects can call `substr()`, `length()`, `find()`, and others. For a `Date` class you might want the following functions:

- A `print()` function to print out the Date on the screen and a `read()` function to read in a Date from the keyboard. These are necessary because we won't be able to use the extraction and insertion operators to read and print, since C++ does not have the extraction and insertion operators defined to work with Date objects. Once we have these 2 functions defined, we would then print out a Date by saying

  ```
  date1.print();
  ```

  We would read in a Date by saying

  ```
  date1.read();
  ```

- A `set()` function that could be used to set the Date. This would take the place of using the assignment statement to set a Date. So if we want to set `date1` to March 19, 1947, we would say

  ```
  date1.set(3,19,1947);
  ```

We will proceed with just these three, and then add more functionality to our `Date` objects as we go.

Before we start defining our class, we should illustrate what a client program that uses the class might look like. We don't need a full-blown program here, just a simple program that demonstrates the use of each of the member functions. Here is an example client program for the `Date` class, along with the output that it should produce:

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

int main()
{
    Date date1;
    Date date2;

    date1.set(4, 27, 1947);
    cout << "date1 should be 4/27/1947: ";
    date1.print();
```

```
    cout << endl;
    cout << "enter a date (M/D/Y): ";
    date2.read();
    cout << "the date you entered was: ";
    date2.print();
    cout << endl;
}
```

```
date1 should be 4/27/1947: 4/27/1947
enter a date (M/D/Y): 11/1/2000
the date you entered was: 11/1/2000
```

Here is the syntax for declaring a `class`. To start with, a `class` declaration has the following form:

```
class Date {
    <list the components that each
     object of this class will contain>
};
```

Make careful note of the semi-colon at the end of this definition. It is unusual in C++ to see a semi-colon immediately after the close curly brace, and so beginning C++ programmers often forget to put it there. Unfortunately, this will result in some rather bizarre error messages, so be sure to get it right the first time.

We use declaration statements to list each component of the class. Recall that a prototype is really a function declaration.

```
class Date {
    void print();
    void read();
    void set(int month, int day, int year);
    int month;
    int day;
    int year;
};
```

At this point we need to make several observations. Each component of a class is called a **member**. Notice that there are two types of members we are listing. We call the members that represent data **data members** or **member variables**. We call the members that are functions **member functions**.

Notice that the `print()` and `read()` member functions have no parameters. This should cause some doubt in your mind. If there are no parameters to the print() functions, what is printed? The answer is that it is the calling object that is being printed. Let me explain. Inside a member function you have access to the data members of the calling object, even though they are not explicitly stated as parameters. Remember that when we call print() in our client program, it will look like this:

`date1.print();`

The objective of this function call is to print out `date1`. So inside our `print()` function we will have statements which print out the month, day, and year data members of the calling object (`date1`, in this example).

There is one more thing we need to fix before we are done with our class declaration. Some of the members we have listed for our `Date`class are intended to be available to the client programmer. Some of the members we have listed are intended to be available **only** to the other members of the class. This is an extremely important concept to understand if you are to use classes effectively. The most important thing to understand about classes is that not only are we defining a new type, we are making that new type completely independent and self-contained, so that the class will be easily reusable; in fact, we are going to maintain a much higher degree of independence and self-containedness than we were able to with functions. In order to maintain this high degree of independence and thus reusability, we are going to deny the client program

access to any of the data members of a class. The result of this is that we could completely change the way that Dates are stored, and completely rewrite each of our member functions to reflect this change, but any client programs that use our class will still work.

On the other hand, we **do** want the client program to be able to call the member functions of our class. We make this distinction by having a "private" section and a "public" section of our class declaration.

To summarize: class members that we declare in the "public" section of the class can be accessed by the client program. Class members that we declare in the "private" section of the class can only be accessed by functions in the same class. They cannot be accessed by the client program.

```
class Date {
    public:
        void print();
        void read();
        void set(int month, int day, int year);
    private:
        int month;
        int day;
        int year;
};
```

It is customary but not required that we place the public section at the top of the class declaration and the private section at the bottom, since the public section is the only section that a client programmer using this class should look at.

Section 15.3: Defining Member Functions

Now we are ready to begin defining the member functions. The definitions of the member functions follow the class declaration. Actually, we will ultimately want to place the class declaration and member function definitions in separate files; however, we'll keep things simple for now.

Consider this definition of the print() member function:

```
class Date {
    public:
        void print();
        void read();
        void set(int month, int day, int year);
    private:
        int month;
        int day;
        int year;
};

void Date::print()
{
    cout << month << "/" << day << "/" << year;
}
```

There are a few observations we need to make about even this very simple function definition. First we need to talk about scope for a minute. In chapter 8 you read about scope. At that time we knew about two kinds of scope: global scope and local scope. To review: a variable or function that is declared in the global scope ("globally") is accessible from anywhere in the program. A variable or function that is declared in a local scope ("locally") is accessible only from the function in which is was declared. All of the functions we have seen so far have been global functions, able to be called from anywhere in your program. All of the variables we have seen so far have been local variables, since I made a rule of not having global variables. We now need to talk about a third type of scope called **class scope**. The function print() is not a global function like the ones we have been writing. You can't call it by simply saying "print()" anywhere in your program. Since the print() function is part of the Date class we say that it is declared in a class scope, specifically the scope of the Date class.

When we go to write our class definition for the print() function, we have to specify that it is not a global function but rather it is in the scope of the Date class. We do this by putting "Date::" in front of the function name. The :: is called the "scope resolution operator". **In summary:** when defining a member function, you have to precede the function name with "classname::" in order to tell C++ that it is not a global function but rather a member of a particular class.

The second observation is a very important one to get right: when I say "month" inside my print() function, it means "the month data member of the calling object". So if the client program included the statement

```
date1.print();
```

using month inside the print member function would mean the month data member of `date1`. To state this rule more generally, **when a data member of the class is used inside the definition of a member function, it refers to that data member of the calling object.**

The entire class is listed below. Study it carefully to make sure you understand how the mechanisms we have discussed are working in each member function. Also, make sure to refer back to the original client program and sample output I provided above to see how the class works together with the client program.

```
class Date {
    public:
        void print();
        void read();
        void set(int month, int day, int year);
    private:
        int month;
        int day;
        int year;
};

void Date::print()
{
    cout << month << "/" << day << "/" << year;
}

void Date::read()
{
    char dummy;

    cin >> month >> dummy >> day >> dummy >> year;
}

void Date::set(int inMonth, int inDay, int inYear)
{
    month = inMonth;
    day = inDay;
    year = inYear;
}
```

Section 15.4: Date Class Continued

Let's discuss in more detail the definitions of `Date::read()` and `Date::set()` from section 15.3.

In the`read()` function, we see the statement `cin >> month`. You might notice that this is a bit different from situations we have seen before this, because it appears that `month` is a variable which has not been declared locally. This is acceptable in this case, because `month` is a data member of the `Date`class. When we say `month` in a member function, we mean the `month` data member of the calling object. So this statement has the effect of placing whatever the user types into the `month` data member of the calling object.

But what is the calling object?? We would need to see the client code to know this. If the client code had a statement like `date1.read()`, then the calling object would be `date1`. If the client code had a statement like `x.read()` (where `x` has been declared as a `Date`object), then `x` would be the calling object.

The next item to be extracted is a variable called `dummy`. This is necessary to consume the slash (/) character that the user is going to include when he enters a Date. After this, the next integer in the input stream goes

into the variable `day`, which refers to the `day` data member of the calling object. Then another slash character is read into the variable `dummy`, and finally the last integer goes into `year`, the `year` data member of the calling object.

The `set` function is slightly more interesting than the `print` or `read` functions. The `set` function has three parameters. Why did I name these `inMonth`, `inDay`, and `inYear` instead of just `month`, `day`, and `year`? Primarily because we need to distinguish between the parameters and the data members. The data members are already named `month`, `day`, and `year`, so we have to think of different names for the parameters.

The first assignment statement inside the `set` function says to set the `month` data member of the calling object equal to the value of the first parameter, `inMonth`. The second assignment statement says to set the `day` data member of the calling object equal to the value of the second parameter, `inDay`. And similarly for the `year` data member. So when the client program has a statement `date1.set(4, 27, 1947);` (as our client program from earlier in this lesson does), what happens is the `month` data member of `date1` gets set to `4`, the `day` data member of `date1` gets set to `27`, and the `year` data member of `date1` gets set to `1947`.

Let's move on now to extend the functionality of our class. We would like to be able to do more than just read Dates, print Dates, and set Dates. For example, we would like to be able to compare two Dates and see if one comes before another, so we will define a function named `comesBefore` which takes one parameter and compares the calling object with the parameter. If the calling object comes before the parameter, `comesBefore` will return `true`, otherwise `comesBefore` will return `false`. A code segment in a client program that uses this function might look like this:

```
if (date1.comesBefore(date2)){
    cout << "date1 comes before date2" << endl;
} else {
    cout << "date1 does not come before date2" << endl;
}
```

To implement this function, we will first compare years. We return true if the year data member of the calling object is less than the year data member of the parameter, and false if the year data member of the calling object is greater than the year data member of the parameter. If neither of these first two conditions is true, it means that the years are equal, and we must go on to compare months. We compare the months in the same manner in which we compared the years, and then go on to compare days if the months are equal. Here is the function definition for comesBefore. It is possible to write this function much more concisely, but I think that this solution is easier to understand than the more concise solution.

```
bool Date::comesBefore(Date otherDate)
{
    if (year < otherDate.year){
        return true;
    }

    if (year > otherDate.year){
        return false;
    }

    if (month < otherDate.month){
        return true;
    }

    if (month > otherDate.month){
        return false;
    }

    return day < otherDate.day;
}
```

Don't forget that in addition to adding this function definition to the function definitions we have, we also need to include the prototype for this function in the class declaration.

Section 15.5: Adding an `increment` Member Function

Let's now add a member function to the `Date` class that will allow us to increment a Date. By "increment" we mean change the value of the `Date` object so that it represents the date that is one day later than the date currently represented. A code segment in a client program that uses this function might look like this:

```
cout << "One day later, date2 is: ";
date2.increment();
date2.print();
cout << endl;
```

Notice that the `increment` function is a `void` function (you can tell because it is called like a statement, not like an expression) that actually modifies the value of the calling object. In a moment we will see a similar example except that instead of modifying the calling object, the function returns the resulting value but does not modify the calling object itself.

The implementation of the `increment` function is not as straightforward as it may at first seem. We begin by simply adding one to the `day` data member. The problem is, what if `day` was originally the last day of the month? In this case, we would have to set `day` back to `1`, and add one to the `month` data member. But what if the `month` was originally `12`? We don't want the `month` to be `13`! So in this case we would have to set the `month` back to `1` and add one to the `year` data member. To further complicate matters, it is not easy to tell whether the day is the last day of the month, because each month has a different number of days. So we'll write an auxiliary function named `numDaysInMonth` that returns the number of days in the month of its calling object. In order to write this function we will need yet another auxiliary function to determine whether the year of the calling object is a leap year. Whew! Here is the code for the `increment` function and its two auxiliary functions:

```
int Date::numDaysInMonth()
{
    switch (month) {
        case 2:if (isLeapYear()){
                    return 29;
                } else {
                    return 28;
                }

        case 4:
        case 6:
        case 9:
        case 11: return 30;

        default: return 31;
    }
}

bool Date::isLeapYear()
{
    if (year % 400 == 0){
        return true;
    }

    if (year % 100 == 0){
        return false;
    }

    return year % 4 == 0;
}


void Date::increment()
{
    day++;

    if (day > numDaysInMonth()){
        day = 1;
        month++;
    }
```

```
        if (month > 12){
            month = 1;
            year++;
        }
    }
```

Consider the call to the `numDaysInMonth` function for a moment. This function will be a member function of the `Date`class. However, it is called without an object in front. So far, every time a member function has been called it has been called using the syntax

```
dateObject.dateMemberFunction();
```

The reason that we don't use this syntax when calling `numDaysInMonth` is this: we want to call `numDaysInMonth` using the object that called `increment` as the calling object. For example, if date1 in the client program was the calling object that called `increment`, we want that same calling object to call the `numDaysInMonth` function. This way when we say `month` in the `numDaysInMonth` function, we are still referring to the same calling object that we had in the `increment` function. The rule for using member functions is the same as the rule for using data members. Recall that when we want to refer to the `month` data member of the calling object, we just use the word `month` by itself. In the same way, when we want to refer to the `numDaysInMonth` member function of the calling object, we just use the function call by itself, without putting an object in front. This same thing occurs again when the `numDaysInMonth` function calls the `isLeapYear` function.

Section 15.6: Adding an `increasedBy` Member Function

Let's now add a member function called `increasedBy` to the `Date`class that will add a certain number of days to a Date. We will design this function somewhat differently than we designed the `increment` function. The `increment` function was designed as a void function that did not return a value but did modify the calling object. Let's design our `increasedBy` function so that it does not modify the calling object, but rather returns a Date that is equal to the calling object increased by the number of days indicated in the parameter. A code segment in a client program that uses this function might look like this:

```
date1 = date2.increasedBy(12);
cout << "After setting date1 to equal date2 + 12,";
cout << "date 2 is still: ";
date2.print();
cout << endl;
cout << "but date1 is now: ";
date1.print();
cout << endl;
```

The statement in which the call to `increasedBy` appears should NOT modify the value of `date2`. Rather it should modify `date1` so that it represents the date that comes 12 days after the date represented by `date2`. The output of this code segment should be:

```
After setting date1 to date2 + 12, date2 is still 7/24/1947
but date1 is now 8/5/1947
```

Compared to the `increment` function, `increasedBy` is fairly straightforward. We simply call `increment` the appropriate number of times. For example, if the parameter is 7, we would call `increment` 7 times. The first interesting thing about the `increasedBy` function is that its return type is `Date`. Although we have not yet seen a function that has a class as the return type, there should be no trouble seeing how this works. It simply means that the value that is returned by this function with the `return` statement must be a `Date`. In addition, when this function is called in the client program, it must be used in the place where you would normally expect to see a `Date`value. In the code segment example above, for example, it is used on the right

side of an assignment statement where the left side of the assignment statement is a `Date`object and so the compiler is expecting a `Date`object on the right side as well.

A second interesting thing about the `increasedBy` function is that we have to have a temporary `Date` object. We set the temporary `Date` object to the value of the calling object by using the `set` function, then we increment the temporary `Date` the appropriate number of times using the `increment` function, and finally we return the temporary `Date`. We need to have a temporary `Date` to avoid modifying the value of the calling object. Let me explain that statement. Some students might initially try to solve this problem by simply incrementing the calling object the appropriate number of times and then returning the calling object. The problem with this approach is that the calling object gets modified, and according to the code segment example above, we don't want the calling object to be modified by this function.

What follows is the complete `Date` class as it now stands. I haven't given the solution to the increasedBy function separately, since you can simply find it in the code below. Notice that the functions numDaysInMonth and isLeapYear are listed as private members. This is because these functions are not intended to be called from the client program; rather they are intended to be called only from other member functions. So we make them private. **All of your data members should always be private**. The rule for member functions, however, is not so simple. Most member functions must be public so that they can be called by the client. Member functions which are not intended to be called by the client, however, should be private.

```
#include
using namespace std;

class Date {
    public:
        void print();
        void read();
        void set(int inMonth, int inDay, int inYear);
        bool comesBefore(Date otherDate);
        void increment();
        Date increasedBy(int numDays);
    private:
        int numDaysInMonth();
        bool isLeapYear();
        int month;
        int day;
        int year;
};


void Date::print()
{
    cout << month << "/" << day << "/" << year;
}


void Date::read()
{
    char dummy;
    cin >> month >> dummy
        >> day >> dummy >> year;
}


void Date::set(int inMonth, int inDay, int inYear)
{
    month = inMonth;
    day = inDay;
    year = inYear;
}


bool Date::comesBefore(Date otherDate)
{
    if (year < otherDate.year){
        return true;
    }
```

```
    if (year > otherDate.year){
        return false;
    }

    if (month < otherDate.month){
        return true;
    }

    if (month > otherDate.month){
        return false;
    }

    return day < otherDate.day;
}



int Date::numDaysInMonth()
{
    switch (month) {
        case 2:if (isLeapYear()){
                    return 29;
               } else {
                    return 28;
               }

        case 4:
        case 6:
        case 9:
        case 11: return 30;

        default: return 31;
    }
}

bool Date::isLeapYear()
{
    if (year % 400 == 0){
        return true;
    }

    if (year % 100 == 0){
        return false;
    }

    return year % 4 == 0;
}




void Date::increment()
{
    day++;

    if (day > numDaysInMonth()){
        day = 1;
        month++;
    }

    if (month > 12){
        month = 1;
        year++;
    }
}



Date Date::increasedBy(int numDays)
{
    Date tempDate;
    int count;

    tempDate.set(month, day, year);
    for (count = 0; count < numDays; count++){
        tempDate.increment();
    }

    return tempDate;
}
```

```
int main()
{
    Date date1;
    cout << "When first declared, date1 is: ";
    date1.print();
    cout << endl;

    date1.set(7, 24, 1949);
    cout << "After being set to 7/24/1949, date1 is: ";
    date1.print();
    cout << endl;

     Date date2;
    cout << "enter a date: ";
    date2.read();
    cout << "you entered: ";
    date2.print();
    cout << endl;

    if (date1.comesBefore(date2)){
        cout << "date1 comes before date2" << endl;
    } else {
        cout << "date1 does not come before date2" << endl;
    }

    date2.increment();
    cout << "one day later, date2 is: ";
    date2.print();
    cout << endl;

    date1 = date2.increasedBy(12);
    cout << "After setting date1 to equal date2 + 12,";
    cout << "date 2 is still: ";
    date2.print();
    cout << endl;
    cout << "but date1 is now: ";
    date1.print();
    cout << endl;
}
```

Section 15.7: Constructors

A constructor is a member function that is automatically called when the client program declares an object. In the future we will discuss some very important uses of constructors; however, for now, the only use that constructors have is initializing objects. For example, the current situation is that when a client of the `Date` class declares a `Date` object, it starts off with uninitialized data -- in other words, junk. As we are writing the `Date` class, we may decide that we would like to have `Date` objects automatically get initialized to a particular date, say January 1, 1600. We would do this by declaring and defining a constructor. To be more specific, consider the following code:

```
Date date1;
date1.print();
```

The current situation is that `date1` is not initialized when declared, so the call to the `print` member function will result in junk being printed on the screen. However, once we add a constructor to the `Date` class, this code segment would cause the date 1/1/1600 to get printed to the screen. As we go on, you may find yourself getting confused about constructors. If this happens, keep in mind that a constructor is fundamentally simply a function that is automatically called when an object is declared.

In order to actually add a constructor to our `Date` class, you'll need to know two things about the syntax of constructors. First, a constructor must have the same name as the class to which it belongs. Second, a constructor has no return type. This does not mean that the return type is `void`. It means that there is no return type whatsoever. Where you might expect to see the word `void` or `int`, there is no word at all. The return type is missing.

Here's the definition we will use for the constructor in our `Date` class:

```
Date::Date()
{
    month = 1;
    day = 1;
    year = 1600;
}
```

Of course, in addition to adding this to our function definitions, we would also have to add a prototype in the class declaration. In the interest of space, we will not illustrate that here. You may refer to the complete listing of the `Date`class at the end of this lesson.

To add even more flexibility to our class, we would like to also include a constructor that allows the client program to initialize the a `Date`object at the same time that it is being declared. For example, we'd like to be able to declare a `Date` object named `date2` and initialize it to February 28, 1965. The syntax for this is as follows:

```
Date date2(2,28,1965);
```

This syntax may seem a bit strange at first. It looks like a cross between a variable declaration and a function call, except the name of the function being called is not `date2`, but rather `Date`! As a matter of fact, this is exactly what is happening. In addition to declaring a new object, we are calling a constructor to initialize the new object. Here's what the new constructor will look like.

```
Date::Date(int inMonth, int inDay, int inYear)
{
    month = inMonth;
    day = inDay;
    year = inYear;
}
```

The body of this constructor looks a lot like the body of the `set` function, and rightly so, since the tasks are similar. The only difference is that the `set` function is dealing with a `Date` object which has already been declared elsewhere in the client program, while the constructor defined here is dealing with a brand new `Date` object.

You might be wondering if this is possible, since we now have two functions both named `Date`! As it turns out, and despite the fact that we have neglected to tell you this before, it is perfectly acceptable to have two functions in C++ with the same name. The compiler identifies which one to execute based on the parameter list, formally called the **signature** of the function. This process (defining a function with the same name as one that already exists, but giving it a different parameter list) is called **function overloading**. We will discuss this in more detail in CS 10b. Another piece of terminology that you should be aware of is that a constructor with no parameters is called a **default constructor**, since it is the constructor that gets called if an object is declared with no arguments.

Please refer to the complete listing of the `Date` class at the end of this lesson to see the constructors tested by the client program.

Section 15.8: Using Multiple Files

I have not yet said anything at all about the how to compile and execute a program that consists of both a class and a client program. First let me say that we could simply put the entire client program and the entire class definition into a single file and compile and run it that way. This, however, doesn't really make sense, since a primary purpose of classes is to allow independent development of the class and the client program. So clearly the client program and the class should be in separate files. This is the way I have been illustrating things in these lessons. I have shown you a snapshot of the client program or a snapshot of the class, but not both at the same time.

In order to do this we would use a `#include` statement at the top of our client program to include the file that contains the class. The file that contains the class is by convention named `name-of-class.h`. For example, we would save our `Date` class in a file named `date.h`. When we want to include a class that we have developed ourselves (as opposed to a class that has been predefined) we surround it with double quotes, instead of with the angle brackets that we are used to. We need not add the file that contains the class to our project, because when we say `#include "date.h"` we are telling the compiler to actually, physically insert the code found in the `date.h` file at this point in the client program.

The other very important thing to watch out for when using 2 files like this is that your .h file should go in the same folder with your project. When you do this wrong, you will get an error that says something like "could not find date.h". (Or, even worse, you may actually be including a date.h file that isn't the same one you are editing!) Here is one way to make sure that your .h file is in the same folder as your project. Get into your IDE and then pretend that you are opening a file. when you get to the folder in which your project is located, you should be able to see the name of your project and the name of your .h file both right there in the same list.

Having said all of this, I now have to tell you that the convention is not to split your program up into two files as we have been discussing, but rather to divide it into three files. The class declaration (the part where we simply list all of the members of the class) goes in one file, and the definitions of the member functions go in a second file. To review, we now have three files:

1. A file which contains the client program. It is named anything.cpp (e.g. "dateclient.cpp") and must be added to the project and must #include the .h file. Its location is not important.

2. A file which contains the definitions of the class member functions. This is by convention named name-of-class.cpp (e.g. "date.cpp"). It must be added to the project along with the client program, and must #include the .h file. This file is usually called the **implementation** file. Its location is not important.

3. A file which contains the class declaration. This is by convention named name-of-class.h (e.g. "date.h"). It is not added to the project, but rather must be #included by the client program and also #included by the implementation file. It must be located in the same folder as the project. This file is called the **specification**file or the **interface** file or the **header** file.

One more detail about the header file. There is a danger that the header file may get included from multiple files in such a way as to cause the compiler to try to compile the code twice. If this happens, you'll get error messages saying that your class has been illegally redefined. In order to avoid this, we place lines similar to the following two lines at the top of every header file:

```
#ifndef DATE_H
#define DATE_H
```

and the following line at the very bottom of every header file:

```
#endif
```

What we are doing here is taking advantage of C++'s preprocessor. The preprocessor can define variables for its own use. In this case, if the variable `DATE_H` has not already been defined, it will be defined and the rest of this code up until the `#endif` will be compiled. If, on the other hand, the variable `DATE_H` has already been defined, all of the code up to the `#endif` will be ignored by the compiler. So the first time the compiler hits this file it will compile its contents. After that it is prevented from attempting to do so again.

Section 15.9: Using `const`

When you define a member function you have access to the calling object's private data members much like you would if the data members of the calling object had all been passed by reference to the function. In fact the data members of the calling object are sometimes called **implicit parameters**, meaning that they are there even though they are not explicitly declared. Since we are able to modify these data members, it would be incorrect to think of them as pass-by-value parameters. Rather, we should think of them as reference parameters. But there is something bad about this. It is good practice to use value parameters whenever

possible, so that we guard against unintentional modifications of the parameter. We should also have a way to guard against unintentional modification of the calling object's data members. This is provided in C++ with the `const` modifier. When we have a member function that is not intended to change the calling object, we put the word `const` at the end of the function header. This has been done in the final version of our `Date` class shown below. Study it and make sure you understand why the word `const` appears where it does and doesn't appear where it doesn't!

There is one more matter of good practice that we should discuss. Objects can sometimes be large and take up a lot of memory. It can be very inefficient to pass them by value, because when you pass by value a copy has to be made of the value being passed. So it is good practice to ALWAYS PASS OBJECTS BY REFERENCE INSTEAD OF BY VALUE. However, this brings up the problem that our objects are once again not guarded against unintentional modification. To remedy this, we can use the word `const` in yet another context. We put the word in front of the parameter in the parameter list, and this means that even though we are using the pass-by-reference mechanism, the compiler will not allow the object to get changed.

Take a good look at the `comesBefore` function below. It illustrates both uses of the word `const` that we have just discussed. The word appears in the parameter list, making it so that the value of the parameter `otherDate` cannot be changed, and then it also appears at the end of the function header, making it so that the calling object cannot be changed.

What follows is our final version of the `Date` class, along with a client program that illustrates all of its features. This version is split into three files. The `Date` class has also been documented according to Style Convention 1D. Please take a moment to read over that Style Convention and make sure you understand how that is implemented here.

```
This is the file dateclient.cpp.  It is added to the project.

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

int main()
{
    Date date1;
    cout << "When first declared, date1 is: ";
    date1.print();
    cout << endl;

    date1.set(7, 24, 1949);
    cout << "After being set to 7/24/1949, date1 is: ";
    date1.print();
    cout << endl;

    Date date2(2, 28, 1965);
    cout << "When first declared and initialized to "
         << "2/28/1965, date2 is: ";
    date2.print();
    cout << endl;

    cout << "enter a date: ";
    date2.read();
    cout << "you entered: ";
    date2.print();
    cout << endl;

    if (date1.comesBefore(date2)){
        cout << "date1 comes before date2" << endl;
    } else {
        cout << "date1 does not come before date2" << endl;
    }

    date2.increment();
    cout << "one day later, date2 is: ";
    date2.print();
    cout << endl;

    date1 = date2.increasedBy(12);
    cout << "After setting date1 to equal date2 + 12,";
    cout << "date 2 is still: ";
    date2.print();
    cout << endl;
```

```
    cout << "but date1 is now: ";
    date1.print();
    cout << endl;
}
```

This is the file date.cpp. It is added to the project

```
#include <iostream>
#include "date.h"
using namespace std;

/*

    Class Invariant: a Date object has 3 int data members: month, which stores the month number
    between 1 and 12, day, which stores the date, and year, which stores the year.  Internal
    operations always result in valid dates (month between 1 and 12, day between 1 and the
    number of days in the month).  However, no effort is made to prevent the client from
    providing an invalid date.

*/

Date::Date()
{
    month = 1;
    day = 1;
    year = 1600;
}




Date::Date(int inMonth, int inDay, int inYear)
{
    month = inMonth;
    day = inDay;
    year = inYear;
}




void Date::read()
{
    char dummy;

    cin >> month;
    cin >> dummy;
    cin >> day;
    cin >> dummy;
    cin >> year;
}




void Date::print() const
{
    cout << month << "/" << day << "/" << year;
}




void Date::set(int inMonth, int inDay, int inYear)
{
    day = inDay;
    month = inMonth;
    year = inYear;
}




bool Date::comesBefore(const Date &otherDate) const
{
    if (year < otherDate.year){
        return true;
```

```
    }

    if (year > otherDate.year){
        return false;
    }

    if (month < otherDate.month){
        return true;
    }

    if (month > otherDate.month){
        return false;
    }

    return day < otherDate.day;
}




void Date::increment()
{
    day++;

    if (day > daysInMonth()){
        day = 1;
        month++;
    }

    if (month > 12){
        month = 1;
        year++;
    }
}




// This private member function returns the number of days in the month of the calling object.

int Date::daysInMonth() const
{
    switch (month) {
        case 2:if (isLeapYear()){
                    return 29;
                } else {
                    return 28;
                }

        case 4:
        case 6:
        case 9:
        case 11: return 30;

        default: return 31;
    }
}




// This private member function returns true if the year of the calling object is a leapyear.
// Otherwise returns false.

bool Date::isLeapYear() const
{
    if (year % 400 == 0){
        return true;
    }

    if (year % 100 == 0){
        return false;
    }

    return year % 4 == 0;
}
```

```
Date Date::increasedBy(int numDays) const
{
    Date tempDate;

    tempDate.set(month, day, year);
    for (int count = 0; count < numDays; count++){
        tempDate.increment();
    }

    return tempDate;
}
```

This is the file date.h. It is not added to the project. It must be located in the same folder as the other source files.

```
#ifndef DATE_H
#define DATE_H

/*

The Date class can be used to create objects that store a date, including month, day, and year.
The following functions are available:

Date();
  post: The calling object has been created and initialized to January 1, 1600

Date(int inMonth, int inDay, int inYear);
  post: The calling object has been created and initialized so that the month is inMonth, the
        day is inDay, and the year is inYear.

void set(int inDay, int inMonth, int inYear);
  post: The calling object has been modified so that the month is inMonth, the day is inDay,
       and the year is inYear.

void print() const;
  post: The calling object has been printed to the console window in the format M/D/Y.

void read();
  post: The calling object has been initialized to the data entered at the console window.
        The date must be entered in the format M/D/Y.

bool comesBefore(const Date &otherDate) const;
  post: Returns true if the calling object comes before the parameter "otherDate". Otherwise
         returns false.

void increment();
  post: The calling object has been advanced by one day.

Date increasedBy(int numDays) const;
  post: Returns the Date determined by starting with the calling object and advancing the day
         numDays times.

*/


class Date {
    public:
        Date();
        Date(int inMonth, int inDay, int inYear);
        void set(int inDay, int inMonth, int inYear);
        void print() const;
        void read();
        bool comesBefore(const Date &otherDate) const;
        void increment();
        Date increasedBy(int numDays) const;
    private:
        int daysInMonth() const;
        bool isLeapYear() const;
        int day;
        int month;
        int year;
};

#endif
```