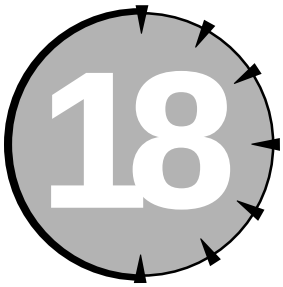
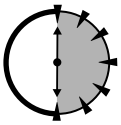
SESSION



Active Classes

Session Checklist

- ✔ Turning classes into active agents through the addition of member functions
- ✔ Naming member functions
- ✔ Defining member functions both inside and outside of the class
- ✔ Calling a member function
- ✔ Accumulating class definitions in #`include` files
- ✔ Accessing `this`



30 Min. To Go

Real-world objects are independent agents (aside from their dependence on things such as electricity, air, and so forth). A class should be as self-sufficient as possible as well. It is impossible for a struct to be independent of its surroundings. The functions that manipulate these classes must be external to the class itself. Active classes have the capability to bundle these manipulator functions into the class itself.

Class Review

A class enables you to group related data elements in a single entity. For example, we might define a class `Student` as follows:

```
class Student
{
  public:
   int   nNSemesterHours;  // hours earned toward
                           // graduation
   float gpa;
};
```

Every instance of `Student` contains its own two data elements:

```
void fn(void)
{
   Student s1;
   Student s2;
   s1.nNSemesterHours = 1;  // this is not the same as...
   s2.nNSemesterHours = 2;  // ...this one
}
```

The two `nNSemesterHours` differ because they belong to two different students (`s1` and `s2`).

A class with nothing but data members is also known as a *structure* and is defined using the keyword `struct`. The `struct` keyword's origins are in C. A `struct` is identical to a class except that the `public` keyword is not necessary.

It is possible to define pointers to class objects and to allocate these objects off of the heap as the following example illustrates:

```
void fn()
{
    Student* pS1 = new Student;
    pS1->nNSemesterHours = 1;
}
```

It is also possible to pass class objects to a function as follows:

```
void studentFunc(Student s);
void studentFunc(Student* pS);
```

```
void fn()
{
    Student s1 = {12, 4.0};
    Student* pS = new Student;

    fn(s1);            // call studentFunc(Student)
    fn(ps);            // call studentFunc(Student*)
}
```

Limitations of Struct

Classes with only data members have significant limitations.

Programs exist in the real world. That is, any nontrivial program is designed to provide some real-world function. Usually, but not always, this function is some analog of what could have been done manually. For example, a program might be used to calculate the grade point average (GPA) of a student. This function could be done by hand with pencil and paper, but it's a lot easier and faster to do it electronically.

The closer that a program can be made to mimic life, the easier it is for the programmer to understand. Thus, if there is a type of thing called a "student," then it would be nice to have a `Student` class that had all the relevant properties of a student. An instance of class `Student` would be analogous to an individual student.

A small class `Student` that describes the properties necessary to keep track of a student's GPA is found at the beginning of this lesson. The problem with this class is that it addresses only passive properties of the student. That is, a student has a number-of-classes-taken property and a GPA property. (The student also has a name, social security number, and so on, but it's OK to leave these properties off if they are not germane to the problem we're trying to solve.) Students start classes, drop classes, and complete classes. These are active properties of the class.

A functional fix

Of course, it is possible to add active properties to a class by defining a set of functions to accompany the class:

```
// define a class to contain the passive properties of
// a student
class Student
{
```

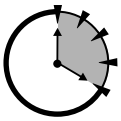
```
  public:
   int   nNSemesterHours;  // hours earned toward graduation
   float gpa;
};

// a Course class
class Course
{
  public:
    char* pszName;
    int   nCourseNumber;
    int   nNumHours;
};

// define a set of functions to describe the active
// properties of a student
void startCourse(Student* pS,
                 Course* pC);
void dropCourse(Student* pS, int nCourseNumber);
void completeCourse(Student* pS, int nCourseNumber);
```

This solution does work — in fact, this is the solution adopted by nonobject-oriented languages such as C. However, there is a problem with this solution.

The way this excerpt is written `Student` has only passive properties. There is a nebulous "thing" out there that has active agents, such as `startCourse()`, that operates on `Student` objects (and `Course` objects, for that matter). Furthermore, this nebulous thing has no data properties of its own. This description, though workable, does not correspond to reality.

We would like to take the active properties out of this undefined thing and attribute them to the `Student` class itself so that each instance of `Student` is outfitted with a complete set of properties.

Adding active properties to the class rather than leaving them unassigned may seem minor now, but it will grow in importance as we make our way through the remainder of the book.



20 Min. To Go

Defining an Active Class

The active properties of a student may be added to the `Student` class as follows:

```
class Student
{
  public:
   // the passive properties of a class
   int   nNSemesterHours;  // hours earned toward graduation
   float gpa;

   // the active properties of a class
   float startCourse(Course*);
   void  dropCourse(int nCourseNumber);
   void  completeCourse(int nCourseNumber);
};
```

The function `startCourse(Course*)` is a property of the class as are `nNSemesterHours` and `gpa`.

A function that is a member of a class is called a *member function*. For historical reasons that have little to do with C++, a member function is also referred to as a *method*. Probably because the term *method* is the more confusing of the two, it is the term of preference.

There isn't a name for functions or data that are not members of a class . I refer to them as nonmembers. All the functions that have been shown so far are nonmember functions because they didn't belong to a class.

C++ doesn't care about the order of members within a class. The data members may come before or after the member functions or they appear mixed together. My personal preference is to put the data members first with the member functions bringing up the rear.

Naming member functions

The full name of the function `startCourse(Course*)` is `Student::startCourse (Course*)`. The class name in front indicates that the function is a member of the class `Student`. (The class name is added to the extended name of the function just as arguments are added to an overloaded function name.) We could have other functions called `startCourse()` that are members of other classes, such as `Teacher::startCourse()`. A function `startCourse()` without any class name in front is a conventional nonmember function.

Actually, the full name of the nonmember function `addCourse()` is `::addCourse()`. The `::` without any class name in front indicates expressly that it is not a member of any class.

Data members are not any different than member functions with respect to extended names. Outside a structure, it is not sufficient to refer to `nSemester Hours` by itself. The data member `nSemesterHours` makes sense only in the context of the class `Student`. The extended name for `nSemesterHours` is `Student::nSemesterHours`.

The `::` is called the *scope resolution operator* because it indicates the class to which a member belongs. The `::` operator can be used with a nonmember function as well as with a null structure name. The nonmember function `startCourse()` should actually be referred to as `::startCourse()`.

The operator is optional except when two functions of the same name exist. For example:

```
float startCourse(Course*);

class Student
{
  public:
   int   nSemesterHours;  // hours earned toward graduation
   float gpa;

   // add a completed course to the record
   float startCourse(Course* pCourse)
   {
      // ...whatever stuff...

      startCourse(pCourse);// call global function(?)
```

```
      // ...more stuff...
   }
};
```

We want the member function `Student::startCourse()` to call the nonmember function `::startCourse()`. Without the `::` operator, however, a call to `startCourse()` from `Student` refers to `Student::startCourse()`. This results in the function calling itself. Adding the `::` operator to the front directs the call to the global version, as desired:

```
class Student
{
  public:
   int   nSemesterHours;  // hours earned toward graduation
   float gpa;

   // add a completed course to the record
   float startCourse(Course* pCourse)
   {
      ::startCourse(pCourse);// call global function
   }
};
```

Thus, the fully extended name of a nonmember function includes not only the arguments, as we saw in Session 9, but also the class name to which the function belongs.

Defining a Member Function in the Class

A member function can be defined either in the class or separately. Consider the following in-class definition of a method `addCourse(int, float)`:

```
class Student
{
  public:
   int   nSemesterHours;  // hours earned toward graduation
   float gpa;

   // add a completed course to the record
   float addCourse(int hours, float grade)
   {
```

```
        float weightedGPA;

        weightedGPA = nSemesterHours * gpa;

        // now add in the new course
        nSemesterHours += hours;
        weightedGPA += grade * hours;
        gpa = weightedGPA / nSemesterHours;
        return gpa;
    }
};
```

The code for `addCourse(int, float)` doesn't appear different than that of any other function except that it appears embedded within the class.

Member functions defined in the class default to inline (see sidebar). Mostly, this is because a member function defined in the class is usually very small, and small functions are prime candidates for inlining.

Inline Functions

Normally a function definition causes the C++ compiler to generate machine code in one particular place in the executable program. Each time the function is called, C++ inserts a type of jump to the location where that function is stored. When the function completes, control passes back to the point where it was when it was originally called.

C++ defines a special type of function called an *inline* function. When an inline function is invoked, C++ compiles the machine code into the spot of the call. Each call to the inline function gets its own copy of machine code.

Inline functions execute more quickly because the computer does not need to jump to some other location and set up before starting to execute. Keep in mind, however, that inline functions take up more space. If an inline function is called 10 times, the machine code is duplicated in 10 different locations.

Because the difference in execution speed is small between an inline and a conventional, sometimes referred to an outline function, only small functions are candidates for inlining. In addition, certain other constructs force an inline function outline.

Writing Member Functions Outside of the Class

For larger functions, putting the code directly in the class definition can lead to some very large, unwieldy class definitions. To prevent this, C++ lets us define member functions outside of the class.

When written outside the class definition, our `addCourse()` method looks like this:

```
class Student
{
  public:
    int   nSemesterHours;  // hours earned toward graduation
    float gpa;

    // add a completed course to the record
    float addCourse(int hours, float grade);
};
float Student::addCourse(int hours, float grade)
{
    float weightedGPA;

    weightedGPA = nSemesterHours * gpa;

    // now add in the new course
    nSemesterHours += hours;
    weightedGPA += grade * hours;
    gpa = weightedGPA / nSemesterHours;
    return gpa;
}
```

Here we see that the class definition contains nothing more than a prototype *declaration* for the function `addCourse()`. The actual function *definition* appears separately.

A *declaration* defines the type of a thing. A *definition* defines the contents of a thing.

The analogy with a prototype declaration is exact. The declaration in the structure is a prototype declaration and, like all prototype declarations, is required.

When the function was among its Student buddies in the class, it wasn't necessary to include the class name with the function name — the class name was assumed. When the function is by itself, the fully extended name is required. It's just like at my home. My wife calls me only by my first name (provided I'm not in the doghouse). Among the family, the last name is assumed. Outside the family (and my circle of acquaintances), others call me by my full name.

Include files

It is common to place class definitions and function prototypes in a file carrying the extension .h separate from the .cpp file that contains the actual function definitions. The .h file is subsequently "included" in the .cpp source file as follows.

The student.h include file is best defined as follows:

```
class Student
{
  public:
   int   nSemesterHours;  // hours earned toward graduation
   float gpa;

   // add a completed course to the record
   float addCourse(int hours, float grade);
};
```

The student.cpp file would appear as follows:

```
#include "student.h";
float Student::addCourse(int hours, float grade)
{
   float weightedGPA;

   weightedGPA = nSemesterHours * gpa;

   // now add in the new course
   nSemesterHours += hours;
   weightedGPA += grade * hours;
   gpa = weightedGPA / nSemesterHours;
   return gpa;
}
```

The #include directive says "replace this directive with the contents of the student.h file."

The #include directive doesn't have the format of a C++ statement because it is interpreted by a separate interpreter that executes before the C++ compiler.

Including class definitions and function prototypes in an include file enables multiple C++ source modules to include the same definitions without the need to repeat them. This reduces effort and, more important, it reduces the chances that multiple source files will get out of synch.

10 Min. To Go

Calling a Member Function

Before we look at how to call a member function, let's review how to reference a data member:

```
#include "student.h"
Student s;
void fn(void)
{  // access one of the data members of s
   s.nSemesterHours = 10;
   s.gpa            = 3.0;
}
```

We must specify an object along with the member name when referencing an object member. In other words, the following makes no sense:

```
#include "student.h"
void fn(void)
{
   Student s;

   // access one of the data members of s
   // neither of these is legal
   nSemesterHours = 10;   // member of what object
                          // of what class?
   Student::nSemesterHours = 10; // okay, I know the class
                          // but I still don't know which
                          // object
```

```
   s.nSemesterHours = 10; // this is OK
}
```

Member functions are invoked with an object just as data members are:

```
void fn()
{
   Student s;

   // reference the data members of the class
   s.nSemesterHours = 10;
   s.gpa            = 3.0;

   // now access the member function
   s.addCourse(3, 4.0);
}
```

Calling a member function without an object makes no more sense than referencing a data member without an object. The syntax for calling a member function looks like a cross between the syntax for accessing a data member and for calling a conventional function.

Calling a member function with a pointer

The same parallel for the objects themselves can be drawn for pointers to objects. The following references a data member of an object with a pointer:

```
#include ""student.h"

void someFn(Student *pS)
{
   // access the data members of the class
   pS->nSemesterHours = 10;
   pS->gpa            = 3.0;

   // now access the member function
   // (that is, call the function)
   pS->addCourse(3, 4.0);
}

int main()
{
```

```
    Student s;

    someFn(&s);
    return 0;
}
```

Calling a member function with a reference to an object appears identical to using the object itself. Remember that when passing or returning a reference as an argument to a function, C++ passes only the address of the object. In using a reference, however, C++ dereferences the address automatically, as the following example shows:

```
#include "student.h"

// same as before, but this time using references
void someFn(Student &refS)
{
    refS.nSemesterHours = 10;
    refS.gpa            = 3.0;
    refS.addCourse(3, 4.0);  // call the member function
}

Student s;
int main()
{
    someFn(s);
    return 0;
}
```

Accessing other members from a member function

It is clear why you can't access a member of a class without an object. You need to know, for example, which `gpa` from which `Student` object? Take a second look at the definition of the member function `Student::addCourse()`.This function is accessing class members without reference to an object in direct contradiction to my previous statement.

You still can't reference a member of a class without an object; however, from within a member function the object is taken to be the object on which the call was made. It's easier to see this with an example:

```
#include "student.h"
float Student::addCourse(int hours, float grade)
```

```
{
   float weightedGPA;
   weightedGPA = nSemesterHours * gpa;

   // now add in the new course
   nSemesterHours += hours;
   weightedGPA += hours * grade;
   gpa = weightedGPA / nSemesterHours;
   return gpa;
}

int main(int nArgs, char* pArgs[])
{
   Student s;
   Student t;

   s.addCourse(3, 4.0);  // here's an A+
   t.addCourse(3, 2.5);  // give this guy a C
   return 0;
}
```

When `addCourse()` is invoked with the object `s`, all of the otherwise unqualified member references in `addCourse()` refer to `s`. Thus, `nSemesterHours` becomes `s.nSemesterHours`, `gpa` becomes `s.gpa`. In the call `t.addCourse()` on the next line, these same references refer to `t.nSemesterHours` and `t.gpa` instead.

The object with which the member function is invoked is the "current" object, and all unqualified references to class members refer to this object. Put another way, unqualified references to class members made from a member function are always against the current object.

How does the member function know what the current object is? It's not magic—the address of the object is passed to the member function as an implicit and hidden first argument. In other words, the following conversion occurs:

```
s.addCourse(3, 2.5);
```

is like

```
Student::addCourse(&s, 3, 2.5);
```

(You can't actually use this interpretive syntax; this is just a way of understanding what C++ is doing.)

Inside the function, this implicit pointer to the current object has a name, in case you need to refer to it. The hidden object pointer is called `this`, as in "Which object? *this* object." The type of `this` is always a pointer to an object of the appropriate class. Thus within the `Student` class, `this` is of type `Student*`.

Anytime that a member function refers to another member of the same class without providing an object explicitly, C++ assumes `this`. You also can refer to `this` explicitly. We could have written `Student::addCourse()` as follows:

```
#include "student.h"
float Student::addCourse(int hours, float grade)
{
   float weightedGPA;

   // refer to 'this' explicitly
   weightedGPA = this->nSemesterHours * this->gpa;

   // same calculation with 'this' understood
   weightedGPA = this->nSemesterHours * gpa;

   // now add in the new course
   this->nSemesterHours += hours;
   weightedGPA += hours * grade;
   this->gpa = weightedGPA / this->nSemesterHours;
   return this->gpa;
}
```

Whether we explicitly include `this` or leave it implicit, as we did before, the effect is the same.

Overloading Member Functions

Member functions can be overloaded in the same way that conventional functions are overloaded. Remember, however, that the class name is part of the extended name. Thus, the following functions are all legal:

```
class Student
{
  public:
   // grade - return the current grade point average
   float grade();
```

```
    // grade - set the grade and return previous value
    float grade(float newGPA);

    // ...data members and stuff...
};

class Slope
{
  public:
    // grade - return the percentage grade of the slope
    float grade();

    // ...stuff goes here too...
};

// grade - return the letter equivalent of a numerical grade
char grade(float value);

int main(int nArgs, char* pArgs[])
{
    Student s;
    Slope o;

    // invoke the different variations on grade()
    s.grade(3.5);         // Student::grade(float)
    float v = s.grade();  // Student::grade()
    char c = grade(v);    // ::grade(float)
    float m = o.grade();  // Slope::grade()
    return 0;
}
```

Each call made from `main()` is noted in the comments with the extended name of the function called.

When calling overloaded functions, both the arguments of the function and the type of the object (if any) with which the function is invoked are used to disambiguate the call.

The term *disambiguate* is object-oriented talk for "decide at compile time which overloaded function to call." One can also say that the calls are being *resolved*.

Done!

In the example code segment, the first two calls to the member functions `Student::grade(float)` and `Student::grade()` are differentiated by their argument lists. The third call has no object, so it unambiguously denotes the nonmember function `grade(float)`. Because the final call is made with an object of type `Slope`, it must refer to the member function `Slope::grade()`.

REVIEW

The closer you can model the problem to be solved with C++ classes, the easier it is to solve the problem. Those classes containing only data members can only model the passive properties of objects. Adding member functions makes the class much more like a real-world object in that it can now respond to the "outside world," that is the remainder of the program. In addition, the class can be made responsible for its own health, in the same sense that real-world objects protect themselves.

- Members of a class can be functions as well as data. Such member functions give the class an active aspect. The full name of a member function includes the name of the class.
- Member functions may be defined either inside or outside the class. Member functions written outside of the class are more difficult to associate with the class, but avoid cluttering up the class definition.
- From within a member function, the current object is referred to by the keyword `this`.

QUIZ YOURSELF

1. What's wrong with defining functions external to the class that directly manipulates class data members? (See "A Functional Fix.")
2. A function that is a member of a class is known as a what? There are two answers to this question. (See "Defining an Active Class.")
3. Describe the significance of the order of the functions within a class. (See "Defining an Active Class.")
4. If a class `X` has a member `Y(int)`, what is the "full" name of the function? (See "Writing Member Functions Outside of the Class.")
5. Why is an include file called by that name? (See "Include Files.")