Some Notes on Defining Classes*

Jeffrey D. Oldham

2000 Feb 21

Contents

Caveat

These terse notes are *not* a substitute for reading the textbook. The textbook provides many more examples and explanations. These are a rough draft of my lecture notes.

1 Steps When Defining Classes

A *type* is a group of elements and permitted operations on them. A *class* is a user-defined type. It consists of a group of *objects* and permitted operations.

1. Determine what the class should do.

   Write a short English description of the class's purpose and then write code using the class from a user's point of view. This reveals what public functions the class must provide.

2. Define the class. We now work from the class implementor's point of view.
   (a) Write the class name, e.g., `class` *`foo`*.
   (b) Write `public` and `private`.
   (c) Write function prototypes for each publicly used function in the use code.
       After this point, there is no need to look at the use code.
   (d) Determine a strategy for implementing the class.
       Write a short English description for how to implement the class. For example, "Our shopping cart's contents will be stored in a vector of pairs, each storing an item's name and price."
       I suppose it would be useful to document this design decision as a comment in the code.
   (e) Write the member function definitions.

2 Defining Member Functions

2.1 Public vs. Private

Public members can be accessed by users of a class. *Private members* cannot be accessed by users of a class. They exist to ease implementation. For example, if a helper function would be useful, there is high probability it should be a private member function.

Summary:

public: used by class users

private: implementation helpers

2.2 Properties of Class Members

C++ designer Stroustrup writes that ordinary member functions have three logically distinct properties:

- Can access both the public and private portions of a class.
- Are inside the class.
- Must be invoked using an object, e.g., `goo.size()`.

We will now consider writing functions without all three properties. *Static member functions* have only the first two properties: They can access public and private portions and are inside the class. *Friend member functions* have only the first property: They can access public and private portions.

2.3 Invocation Using an Object

When using a member function, we must specify which object to use. For example, to invoke the `size()` member function on the `goo` object, we use `goo.size()`. When defining a member function, we are given "free" access to the "current object" (called `*this` in C++). Conceptually, when we move from the user's point of view to the implementor's point of view, we move from using an explicit object `goo` to the implicit "current object."

The current object stores its values for the class variables. For example, in the shopping cart definition, `insert`'s definition uses the private variable `items` residing in the object for which `insert` was invoked.

In an ordinary member function, if a variable is not defined in the function as either a local variable or a parameter, the current object is searched for the variable.

Conceptually, one way to view member function invocation such as `goo.size()` is as invoking `size` with a hidden first argument of the `goo` object. Thus, *conceptually*, `goo.size()` is equivalent to `size(goo)`. This may help understand why a C++ compiler gives strange error messages sometimes.

2.4 Static Class Members

Static class members can access both the private and public portions of a class and are members of the class, but they are not invoked using an object. Because they are not invoked using an object, they cannot use the current object to look up values for class variables such as `items` described above.

To define a static member function, prefix the word `static` to the beginning of the member function definition. Be sure the definition does not access any non-static members, but it may access both private and public static members. In other words, it may not access any variables whose values are stored in the current object.

To invoke a static member `int foo(int x)` defined in a class `goo`,

inside the class: use `foo(3)`, where `3` is an arbitrary `int` value.

outside the class: use `goo::foo(4);`. Prefixing with `goo::` indicates `foo` is part of the class `goo`.

2.4.1 When to Use

Use a static member whenever you want to write code that is independent of any particular object in the class. One frequent use is to define helper functions for use with STL functions. These helper functions usually can have only a fixed number of parameters and we want to ignore the hidden "current object" parameter that would be present if we used a member function.

2.5 Friend Functions

A *friend function* is a function logically associated with a class that is not a member of the class but still needs access to private portions. For example, the output operator `operator<<` for the shopping cart class needs access to the private `vector` of items in the shopping cart but cannot be a member function because of the way the `iostream` library is constructed. Looking at the code for `operator<<`, we see that it does access the private `items` portion of its `shoppingCart` argument `sc`, e.g., `sc.items.begin()`.

Even though friend functions are not part of the class, their definitions can appear directly inside the class if desired. Just prefix the definition with `friend`. Make sure that they access only static class members or use an object passed as an argument to access the private parts.

The public and private labels do not impact friends since they are not class members.

2.5.1 When to Use

Friend functions should be used when a function needs access to the private portion of a class or one of its objects, but the function cannot or should not be part of the class. Common examples include input and output operators. Another use is for binary operators for which both parameters have equal importance so, if it was written as a member function, it would not be clear which parameter to use as the hidden current object.

3 Overloaded Functions

Overloaded functions are two or more functions with the same name but different parameters. The parameters must differ in number and/or type. (Differing only in return type is not sufficient.) For example, the STL function `sort` takes either two or three arguments, depending on whether one specifies a comparison operator.

3.0.1 When to Use

Use overloaded functions when two or more functions perform the same task, but different functions are needed to handle different parameters.

3.1 Overloaded Operator Functions

Almost all C++ operators can be defined for use with classes. We introduce the syntax by example: `bool operator||(const foo & f, const foo & g)`. Since `||` for built-in types takes two arguments, it must take two arguments when overloading it. At least one of these two arguments must be a user-defined type; otherwise you would be trying to redefine the operation on two C++ built-in types.

Overloaded operators that are class members have one fewer parameters than if they were not class members. For example, if you decide to make `operator!` a class member, it will take no arguments, while, if not a class member, it would have one argument. This is because class members have a first parameter of the hidden current object.

Tips:

- Choose an operator appropriate for the function's purpose. For example, if a function is to add two complex numbers, use `operator+`.
- For many operators, also consider implementing their assignment form. For example, if `operator-` is defined, it may also be useful to define `operator-=`.

4 Heuristics

It is an art, not science, to decide whether a function should be overloaded, given an operator name, be a friend or static function.

4.1 General Questions

- Does it need access to the private portion of a class? If yes, make it an ordinary member, a static member, or a friend.
- Does it need access to an object? If yes, make it an ordinary member or a friend function. If no, consider a static member.
- Can it have a hidden first parameter consisting of the current argument? That is, must the number of parameters be fixed? If the number is fixed, consider a static member.
- Does it need access to the private portion of multiple objects? If none of these objects is most important, consider a friend function.
- Is its function similar to other functions? to an operator? If yes, make it an overloaded function, possibly an overloaded operator function.

4.2 Specific Heuristics for Ordinary, Static, and Friends

1. By default, make a class function an ordinary member function.
2. If it is to be used as an STL helper function, make it static.
3. If it is a binary operator with two object arguments, make it a friend.
4. How many objects' private portions does it use? zero $\Rightarrow$ static, one $\Rightarrow$ ordinary, two or more $\Rightarrow$ friend.