CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 14
October 24, 2016

Feedback on Programming Style

Smart Pointer Demo

More on Course Goals

Feedback on Programming Style

Coding Hints

In the next few slides, I will point out some miscellaneous programming issues that turned up on PS2. Proper C++ style is somewhat different from other languages (include C). Part of professional-level C++ proficiency is learning not just what works but what is simple and efficient.

Zero-tolerance for compiler warnings

Compiler warnings flag things that are not proper C++ usage but may work anyway in some environments. They generally indicate program errors or sloppy style.

You need to learn what the warnings mean and how to avoid them. Don't just ignore warnings because you think they are unimportant. "Unimportant" warnings will mask important ones that result from real bugs in your code.

Example: Comparing an `unsigned int` with an `int` gives such a warning.

Fix: Use appropriate integer types.

Declaration order in classes

There are two schools of thought on the order of declarations within classes:

1. Put the public functions first followed by the private.
   Rationale: The public functions represent the interface and are what clients of the class wnat to see.
2. Put the private data members and functions first followed by the public.
   Rationale: Generally names must be declarated before they are used. It's natural to declare data members before functions that might use them, even if C++ provides some flexibility.

In this course, I require the second style: private first, public last.

Construct semantically consistent objects

Constructors should leave objects in a semantically meaningful state.

Avoid the paradigm common in other languages to create uninitialized objects and then initialize data members from member functions.

Use `break`

Instead of

```
bool exit = false;
while (!exit) {
   ...
   if (...) exit = true;
   else {
      ...
   }
}
```

use

```
for (;;) {
   ...
   if (...) break;
    ...
}
```

Use `tolower()`

Instead of

```
if (input=='Q' || input=='q') ...
```

use

```
#include <cctype>
...
input = tolower(input);
if (input=='q') ...
```

Use `switch`

Instead of

```
if (input=='a' || input=='b' || input=='c') { ... }
else if (input=='p') {
  ...
```

use

```
switch (input) {
case 'a':
case 'b':
case 'c': ...; break;
case 'p': ...; break;
}
```

Use stream input to read data

Instead of

```
int x;
string s;
s.getline(in);
// extract substring
// convert substring to number
 ...
```

use

```
int x;
in >> x;
```

`continue`: Instead of

```
for (;;) {
   in >> x;
   if ( <error> ) {
      <handle error>
   }
   else {
      <do stuff>
      in >> y;
      if ( <error> ) {
         <handle error>
      }
      else {
        <do stuff>
      }
   }
}
```

use `continue`:

```
for (;;) {
    in >> x;
    if ( <error> ) {
        <handle error>
        continue;
    }
    <do stuff>
    in >> y;
    if ( <error> ) {
        <handle error>
        continue;
    }
    <do stuff>
}
```

Use `new` and `delete`, not `malloc` and `free`

C uses `malloc` and `free` to allocate and free dynamic storage.

C++ uses `new` and `delete`.

What are the differences?

1. `new` and `delete` are type safe; `malloc` and `free` are not.
2. `new` calls the constructor and `delete` calls the destructor. `malloc` and `free` are unaware of C++ classes and just handle uninitialized storage.
3. Array forms `new[]` and `delete[]` call default constructors and destructors of array elements.

Don't use `malloc` and `free` in C++ programs.

End-of-file handling

Don't use

```
while (!in.eof()) {
  in >> x;
  <do stuff with x>
}
```

to read and process a file of numbers. Even if `in.eof()` returns `false`, the next read might fail. Instead, use

```
for (;;) {
  in >> x;
  if (in.fail()) { <handle error/eof condition> }
  <do  stuff with x>
}
```

Include guards

Include guards are a method of using the C preprocessor to make sure that the declarations in a header file are not included more than once in a compilation. Here's how they work:

- A preprocessor symbol `GATE_HPP` is associated with a header file `gate.hpp`. Initially, `GATE_HPP` is undefined.
- Before `gate.hpp` is processed, `#ifndef GATE_HPP` is used to test if `GATE_HPP` is already defined.
- If it is, `gate.hpp` has already been processed and is skipped.
- If not, `#define GATE_HPP` defines `GATE_HPP` and the header file `gate.hpp` is processed.

Where do the include guards go?

They could be used to protect either the `#include "gate.hpp"` statement or the body of the header file `gate.hpp`.

Because there may be many `#include "gate.hpp"` statements in the program but there is only one `gate.hpp` file, they are normally placed inside the header file itself, e.g.,

```
// File gate.hpp
#ifndef GATE_HPP
#define GATE_HPP
   <body of header file>
#endif
```

Smart Pointer Demo

Dangling pointers

Pointers can be used to permit object sharing from different contexts.

One can have a single object of some type `T` with many pointers in different contexts that all point to that object.

Problems with shared objects

If the different contexts have different lifetimes, the problem is to know when it is safe to delete the object.

It can be difficult to know when an object should be deleted. Failure to delete an object will cause memory leaks.

If the object is deleted while there are still points pointing to it, then those pointers become invalid. We call these dangling pointers.

Failure to delete or premature deletion of objects are common sources of errors in C++.

Avoiding dangling pointers

There are several ways to avoid dangling pointers.

1. Have a top-level manager whose lifetime exceeds that of all of the pointers take responsibility for deleting the objects.
2. Use a garbage collection. (This is java's approach.)
3. Use reference counts. That is, keep track somehow of the number of outstanding pointers to an object. When the last pointer is deleted, then the object is deleted at that time.

Modern C++ Smart Pointers

Modern C++ has three kinds of **smart pointers**. These are objects that act very much like raw pointers, but they take responsibility for managing the objects they point at and deleting them when appropriate.

- `shared_ptr`
- `weak_ptr`
- `unique_ptr`

We will discuss them later in the course. For now, we present a much-simplified version of shared pointer so that you can see the basic mechanism that underlies all of the various kinds of shared pointers.

Smart pointers

We define a class `SPtr` of reference-counted pointer-like objects.

An `SPtr` should act like a pointer to a `T`.

This means if `sp` is an `SPtr`, then `*sp` is a `T&`.

We need a way to create a smart pointer and to create copies of them.

Demo `14-SmartPointer` illustrates how this can be done.

More on Course Goals

Low-level details

- C++ is a large and complicated language with many quirks and detailed rules.
- One goal of this course is for you to learn how to deal effectively with a complex system where it is not feasible to know everything about it before beginning to use it.
- Low-level details tend to be easy to find in the documentation once you know what to look for.
- What's important to learn is the overall roadmap of the language and where to look to find out more.

Example picky detail

- If you do not supply a constructor for a class, C++ automatically generates a null default constructor for you, that is, one that takes no parameters and does nothing.
- If you do define a constructor, the default constructor is *not* generated. If you want it, you then need to explicitly request it or define it yourself, e.g.,

```
MyClass() =default;
```

- What if you didn't know this and assumed the default constructor was pre-defined? The compiler would give you an error comment about it not being defined, and you would be started on the track of trying to figure out why.

Efficient use of resources

Efficiency is concerned with making good use of available resources:

- Time (how fast a program works)
- Memory (how much memory the program requires)
- Other resources that are scarce and relatively costly to create:
  - Network connections (TCP sockets)
  - Database connections

Strategy for improving efficiency: Reuse and recycle. Maintain a pool of currently unused objects and reuse rather than recreate when possible.

In the case of memory blocks, this pool is often called a **free list**.

Efficiency measurement

A first step to improving efficiency is to know how the resources are being used.

Measuring resource usage is not always easy.

The next demo is concerned with measuring execution time.