CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 8
September 26, 2016

Storage Management (continued)

Static data members

A static data member belongs to the class itself rather than to instantiations of the class. Thus, all instantiations share the static member. Moreover, it exists even before the class has been instantiated.

A static class data member must be *declared* and *defined*.

- It is declared by preceding the member declaration by the qualifier `static`.
- It is defined by having it appear in global context *with* an initializer but *without* the keyword `static`.
- It must be defined *only once*.

Example from brackets demo

In `token.hpp`, variable `brackets` is defined to be a static constant string:

```
static const string brackets;
```

It is initialized in `token.cpp`:

```
const string Token::brackets = "[](){}<>";
```

Static data member example

Here's a case where a non-const static data member is useful. In debugging a program with many objects of the same type, it is often useful to be able to distinguish the objects in diagnostic printouts.

Using a static data member `nextID`, here's how to give each new object a unique ID:

1. Declare `static int nextID;` in the class.
2. Initialize it to 0 in the `.cpp` file.
3. Declare a non-static data member `int uid = nextID++;`

Class definition

```
class MyClass {
    static int nextID;
    int uid = nextID++;
public:
    ostream& printID(ostream& out) const {
        return out << uid << endl;
    }
};
```

In the `.cpp` file, you would place the line:

```
int MyClass::nextID = 0;
```

Static function members

Function class members can also be declared `static`.

- As with static variables, the are declared inside the class by prefixing `static`.
- They may be defined either inside the class (as inline functions) or outside the class.
- If defined outside the class, the `::` prefix must be used and the word `static` omitted.

A static function can be called before the class has been instantiated. Example: Suppose `MyClass` defines

```
static void instructions() { cout <<"To use..."; }
```

It can be called using `MyClass::instructions();`.

Static class functions are global

Ways in which static class functions are like global C-style functions:

1. They do not take an implicit argument.
2. Their lifetime is the same as the lifetime of the entire program.
3. They may be called before the class has been instantiated.

Ways in which they differ:

1. The name visibility is *not* restricted to the file in which the function is declared.
2. The name visibility *is* restricted by the privacy keywords `private, protected, public`.
3. The name must be qualified when called from outside the class (eg., `MyClass::instructions()`).

Debugging memory management errors

Memory management errors can be particularly difficult to debug because they often lead to bizarre symptoms that mislead debugging efforts.

Debugging tools such as `gdb` have their place, but be aware that a program with memory management errors may behave differently when run under a debugger than when run alone.

I'll give a brief overview of several kinds of errors that can happen, what causes them, and what to do about them. Read chapter 6 of the textbook for a much fuller discussion.

Five common kinds of failures

1. **Memory leak**—Dynamic storage that is no longer accessible but has not been deallocated.
2. **Amnesia**—Storage values that mysteriously disappear.
3. **Bus error**—Program crashes because of an attempt to access non-existent memory.
4. **Segmentation fault**—Program crashes because of an attempt to access memory not allocated to your process.
5. **Waiting for eternity**—Program is in a permanent wait state or an infinite loop.

Memory leak

A **memory leak** is when storage that has been allocated to a process becomes inaccessible.

Symptoms are that the process's memory footprint becomes larger and larger over time, eventually leading the process to become very slow as useless data gets swapped in and out of physical memory.

The tool `valgrind` will catch many kinds of memory errors, including memory leaks.

To use, type `valgrind myapp arg1 arg2 ...`. This runs `myapp arg1 arg2 ...` under its control.

Amnesia

"I stored 7 in `x`. Why did `cout << x` print 356991?"

Answer: "Somebody changed it between the time you stored into `x` and when you printed it." A few of many possible reasons:

- You no longer own the memory block `x`, and it is being reused for something else.
- Your program took a different path through your code than you expected.
- Some other part of your code was using memory that didn't belong to it, and it just happened to overwrite `x`. This can be caused by a **buffer overrun** error.

Segmentation fault

From the system's perspective, a memory reference can be one of three kinds:

1. The memory belongs to your process and you are permitted access.
2. The reference is to memory that exists, but you are not permitted to perform the requested operation on it.
3. A signal raised in response to various kinds of addressing errors detected by the hardware.

Segmentation faults often result from attempting to dereference an invalid pointer, as in the following:

```
int* xp = nullptr;
cout << *xp << endl;
```

Bus error

Bus errors are less common in modern operating systems. Dereferencing 0 on older systems often caused bus errors.

They can still happen in various situations, and you may occasionally still see them. The program mistakes that cause them are pretty much the same as the ones that cause segmentation faults.

Waiting for eternity

"I wrote `cout << x;` , but nothing prints. Why?

Some possibilities:

- Your output is in a buffer, waiting for a new line to be printed.
- Program control never reached your print statement.
- Your print statement was accidently commented out, perhaps because of an earlier missing `*/`.
- Your program is in an infinite loop. Maybe there is no termination clause. Maybe you forgot to initialize a variable needed for termination. Maybe the termination condition simply never becomes true because of a logic error.

What kinds of program errors cause these problems?

- `delete` is never executed, or it's executed twice.
- Dereferencing uninitialized pointer.
- Referencing a formerly valid pointer after its memory has been deleted or deallocated.
- Failing to initialize variables that control program logic.

What makes these problems hard to diagnose?

The effects of memory management errors often are only observed long after the original error, and often in an unrelated piece of code.

References to unowned memory may return different values on different runs of the system, leading to seemingly random behavior.

They may also return different values when run in a different environment, such as under the control of `gdb` or `valgrind`.

Seemingly random behavior is almost always an indication of memory errors in your program. Finding the cause of the error requires methodical tracing of control flow through your program.

Bar Graph Demo

Overview of bar graph demo

These slides refer to demo 08-BarGraph.

This demo reads a file of student exam scores, groups them by deciles, and then displays a bar graph for each decile.

The input file has one line per student containing a 3-letter student code followed by a numeric score.

```
AWF  00
MJF  98
FDR  75
...
```

Scores should be in the range $[0, 100]$

Overview (cont.)

The output consists of one line for each group listing all of the students falling in that group. An 11^{th} line is used for students with invalid scores.

Sample output:

```
00..09:  AWF 0
10..19:
20..29:
30..39:  PLK 37
40..49:
50..59:  ABA 56
60..69:  PRD 68 RBW 69
70..79:  HST 79 PDB 71 FDR 75
80..89:  AEF 89 ABC 82 GLD 89
90..99:  GBS 92 MJF 98
Errors:  ALA 105 JBK -1
```

Method

Each student is represented by an `Item` object that consists of the initials and a score.

The program maintains 11 linked lists of `Item`, one for each bar of the graph. A bar is represented by a `Row` object.

For each line of input, an `Item` is constructed, classified, and inserted into the appropriate `Row`.

When all student records have been read in, the bars are printed.

A `Graph` object contains the bar graph as well as the logic for creating a bar graph from a file of scores as well as for printing it out.

Analysis of 08-BarGraph demo

- `main.cpp`
- `graph.hpp`
- `graph.cpp`
- `row.hpp`
- `row.cpp`
- `rowNest.hpp`
- `item.hpp`