CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 4
September 13, 2011

Remarks on Laboratory Work

Toolset to use for course work

This course uses a collection of software tools running under Linux, including `g++`, `valgrind`, `eclipse`, `make`, a command shell such as `bash` or `tcsh`, and Linux libraries and header files.

These and other tools you will need are installed and maintained on the Zoo machines.

You are all entitled to Zoo accounts for use in this course, and you will all be granted 24-hour access to the Zoo (in Arthur K. Watson Hall). *I expect you to use the Zoo for all course work.*

Working remotely

For those of you who find it difficult to get to the Zoo and want to work remotely, I offer three suggestions, all of which have some drawbacks.

1. Replicate the Zoo environment on your own machine

This means installing Linux, either in place of your native operating system, beside it in a dual-boot arrangement, or on top of it using virtualization software.

Drawbacks: It takes time and expertise to install and configure all of the software you will need, and there is still the danger of incompatibilities with the Zoo if you don't end up with exactly the same versions of everything.

2. Remote login to the Zoo

Log into the Zoo remotely via `ssh` (which you must install on your machine), and use command-line tools such as `emacs` and `make` to develop your code.

Drawbacks: People used to program using command-line tools, but it is cumbersome compared with using a modern graphical windowing system and an IDE such as `eclipse`.

3. Set up a virtual Zoo desktop on your machine

To use VNC (Virtual Network Computing), you must:

1. install `SSH` and `VNC` clients on your machine;
2. log onto a Zoo machine using `ssh`;
3. start a VNC server on the Zoo;
4. connect to the VNC server using your VNC client.

See handout 3 (pdf) for detailed instructions.
VNC gives you a virtual Zoo desktop that, in principle,will look and feel just as if you were sitting at a Zoo console.

Drawbacks: It won't really feel the same. You will notice delayed response to your mouse actions. It's a bit of a nuisance getting the connection set up each time you want to work. Moving files back and forth between your local machine and the Zoo requires other tools such as `rsync` or `scp`.

Homework submission

Completed homework is to be submitted on the Zoo using the command `/c/cs427/bin/submit`.

In order to submit, *you must have a course account*.

Put the files you want to submit into a subdirectory, go to that directory, and run `submit`.

The first argument to `submit` is the problem set number.
The remaining arguments are the files to be submitted.

Example: `submit 1 *` submits everything in the current directory for problem set #1.

Review and Readings

A brief course review to date

Lecture 1 describes the course goals of how to construct software that is efficient, robust, scalable, maintainable, reusable, and understandable, as well as giving correct outputs on correct inputs.

Lecture 2 looks at how object-oriented design principles can be applied even to C programs, pointing out also inherent limitations of C that motivated the development of C++.

Lecture 3 gives a whirlwind tour of an object-oriented C++ program for insertion sort, looking in particular at how the various pieces of code are split into *interface or header files* (`.hpp`) and *implementation or code files* (`.cpp`).

How to use the textbook

The lectures do not exactly follow the textbook, but they are roughly parallel.

For example, lectures 1–3 generally correspond to chapters 1 and 2, although several concepts from chapters 3 and 4 were also covered briefly.

You should read the corresponding chapters carefully, because there is information in the book that will not be covered explicitly in class but that you should nevertheless know.

A Survival Guide for PS1

Operator extensions

For PS1, you need to extend three operators `<=`, `<<`, and `>>` to work with type `Player`.
The corresponding function names are:

Operator	Function name
`<=`	`operator<=`
`<<`	`operator<<`
`>>`	`operator>>`

Operators extensions are simply new methods for the corresponding functions.

Adding new methods

Every function in C++ may have many **methods**.

Which method is selected in a function call depends on the number and types of its arguments, which we call its **signature**.

Every method must have a distinct signature.

For PS1, the signatures of the methods to be defined are given in `Player.hpp`.

```
bool operator<=(const Player& p2) const;
istream& read(istream& in);
ostream& print(ostream& out) const;
```

Two kinds of functions

Top-level functions: These are ordinary C-style functions.

Member functions: These are functions that belong to a class. When run, the special variable `this` is an **implicit parameter** which points to an instance of the class.

Corresponding to the two kinds of functions are two different calling syntaxes:

Top-level call: Like C, e.g., `f(x, a)`.

Member call: Uses the field selector "dot" notation, e.g., `x.g(a)`. Here, `x` is an **object** (instance of a class) containing a member function `g()`.
`x` becomes the implicit parameter of `g()`; `a` the explicit parameter.

An ambiguity with operator extensions

An operator like `+` invokes its associated function `operator+()`.

But which kind of call does `a+b` correspond to?

1. `operator+(a, b)`?
2. `a.operator+(b)`?

The answer is "both", with preference given to (2) if the corresponding method is defined.

Operator call example: Top-level function

```
class Foo {
private: int a;
public:  int getA() const { return a; }
};

int operator+(Foo x, int b) {
    return (x.getA()+b)/(x.getA()*b);
}
...
Foo x;
int y, z;
z = x+y;
```

Operator call example: Member function

```
class Foo {
private:
    int a;
public:
    int operator+(int b) const { return (a+b)/(a*b); }
};
...
Foo x;
int y, z;
z = x+y;
```

Back to PS1

For PS1, the declaration

```
bool operator<=(const Player& p2) const;
```

appears inside of the definition of `class Player`, so it is a member function.

The declarations

```
istream& operator>>(istream& in);
ostream& operator<<(ostream& out) const;
```

appear outside of any class definition, so they are top-level.

Note that the latter are already declared and defined in `player.hpp`, so you don't need to do anything more than define the methods `read()` and `print()` on which they depend.

More on C++ I/O

Opening and closing streams

Some ways of opening a stream.

- `ifstream fin ( "myfile.in" );` opens stream `fin` for reading. This implicitly invokes the constructor `ifstream( "myfile.in" )`.
- `ifstream fin;` creates an input stream not associated with a file. `fin.open( "myfile.in" );` attaches it to a file.

Can also specify open modes.

To close, use `fin.close();`.

Reading data

Simple forms. Assume `fin` is an open input stream.

- `fin >> x >> y >> z;` reads three fields from `fin` into `x`, `y`, and `z`.
- The kind of input conversion depends on the types of the variables.
- No need for format or `&`.
- Standard input is called `cin`.
- Can read a line into a buffer with `fin.get(buf, buflen);`. This function stops before the newline is read. To continue, one must move past the newline with a simple `fin.get(ch);` or `fin.ignore();`.

Writing data

Simple forms. Assume `fout` is an open output stream.

- `fout << x << y << z;` writes `x`, `y`, and `z` into `fout`.
- The kind of output conversion depends on the types of the variables or expressions..
- Standard output is called `cout`. Other predefined output streams are `cerr` and `clog`. They are usually initialized to standard output but can be redirected.
- Warning: The eclipse debug window does not obey the proper synchronization rules when displaying `cout` and `cerr`. Rather, the output lines are interleaved arbitrarily. In particular, a line written to `cerr` **after** a line written to `cout` can appear **before** in the output listing. This won't happen with a Linux terminal window.

Manipulators

Manipulators are objects that can be arguments of `>>` or `<<` but do not necessarily produce data.

Example: `cout << hex << x << y << dec << z << endl;`

- Prints `x` and `y` in hex and `z` in decimal.
- After printing `z`, a newline is printed and the output stream is flushed.

Manipulators are used in place of C formats to control input and output formatting and conversions.

End of file and error handling

I/O functions set status flags after each I/O operation.

`bad` means there was a read or write error on the file I/O.

`fail` means the data was not appropriate to the field, e.g., trying to read a non-numeric character into a numeric variable.

`eof` means that the end of file has been reached.

`good` means that the above three bits are all off.

The whole state can be read with one call to `rdstate()`.
Individual bits can be tested with `bad()`, `fail()`, `eof()`, `good()`.

As in C, correct end of file and error checking require paying close attention to detail of exactly when these state bits are turned on.

To continue after a bit has been set, must call `clear()` to clear it.