CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 2
August 31, 2018

Tasks for this week

- Log in to the Zoo. You may see a CPSC 427 subdirectory already created for you.
- Read Chapters 1–3 of *Exploring C++*. (36 pages in all.)
- Do problem set 1.

C++ Overview

Why did C need a ++?

Chapter 2 of Exploring C++

1. C was designed and constructed a long time ago (1971) as a language for writing Unix.
2. The importance of data modeling was poorly understood at that time.
3. Data types were real, integer, character, and array, of various sizes and precisions.
4. It was important for C to be powerful and flexible but not to have clean semantics.
5. Nobody talked much about portability and code re-use at that time.

Today, we demand much more from a language.

C++ was Designed for Modeling

Design goals for C++ (Bjarne Stroustrup)

1. Provide classes (replacing structs) as a means to model data.
2. Let a class encapsulate data, so that its implementation is hidden from a client program.
3. Permit a C++ program to link to libraries from other languages, especially FORTRAN.
4. Produce executable code that is as fast as C, unless run-time binding is necessary.
5. Be fully compatible with C, so that C programs could be compiled under a C++ compiler and still work properly.

General properties of C++

- Widely used in the real world.
- Close to the machine and capable of producing efficient code.
- Gives a programmer fine control over the use of resources.
- Supports the object-oriented programming paradigm.
- Supports modularity and component isolation.
- Supports correctness through privacy, modularity, and use of exceptions.
- Supports reusabale code through derivation and templates.

C++ Extends C

- C++ grew out of C.
- Goals were to improve support for modularity, portability, and code reusability.
- Most C programs will compile and run under C++.
- C++ replaces several problematic C constructs with safer versions.
- Although most old C constructs will still work in C++, several should *not* be used in new code where better alternatives exist.

  Example: Use Boolean constants `true` and `false` instead of 1 and 0.

Some Extensions in C++

- One-line comments //.
- Executable declarations.
- Type `bool`.
- Enumeration constants are no longer synonyms for integers.
- Reference types.
- Definable type conversions and operator extensions.
- Functions with multiple methods.
- Classes with private parts; class derivation.
- Class templates.
- An exception handler.

Building a Project

Modules

A **compilation module** is a collection of **header files** (`.h` or `.hpp`) and an **implementation file** (`.c` or `.cpp`) that can be processed by the C or C++ **compiler** to produce an **object file** (`.o`) file.

A **project** is a collection of compilation modules that can be processed by the **linker** to produce a runnable piece of code called an **application** (or **program** or **executable** or **command**).

Some modules are part of the project. Others come from **libraries** (`.a` or `.so` files) that contain object code for modules written by others and provided by the system for your use.

Whatever the origin of the modules, they must be joined together during final assembly to produce the runnable application. This step of the process is called **linking**.

Separate compilation model

Unlike some languages, C/C++ permits independent compilation of modules. In the traditional **separate compilation model**, each module is compiled separately to produce a corresponding object file. Then the object files and necessary libraries are linked together to produce the executable.

The C/C++ programmer must clearly distinguish between compilation and linking, especially when interpreting error comments from the build process.

The build process

To summarize, the process of building an executable file consists of two phases:

1. Each module in the project is compiled to produce corresponding object files.
2. All object files in the project are linked together with necessary libraries to produce the executable file.

Because the executable must be rebuilt every time one of the source files is changed, manually going through the build process can be tedious and error-prone.

Automating the build process

Two common ways to automate the build process:

1. Use the `make` command. `make` reads a special file (`Makefile` or `makefile`) which contains a description of the necessary steps to build the application. It's also smart about not recompiling modules that have not changed since the last build.
2. Use an Integrated development environments (IDE) such as `Xcode` on the Mac or `Eclipse` on linux machines. The IDE keeps track of which modules belong to the project so that they can be rebuilt when needed.

Local build requirement

In this course, you're free to use whatever build tools you wish. However, you must submit a correct makefile as part of your code so that the grader can simply type `make` in order to produce an executable that will run on the Zoo.

What comprises a module?

A module consists of one or more **header files** and at most one **implementation** file.

Header files provide the context to the compiler for understanding the code in the implementation file. The `#include` directive names a header file that the compiler should process when compiling this module.

Header files for system libraries are often found in the `/usr/include` directory, but they can be put anywhere as long as the *compiler* is told where to look for them.

Header files for the current module are generally located in the same directory as the implementation file being compiled.

Header files

Header files contain class, data, function, and other declarations that are needed by the **client** of the module. They need to be included by every module that uses those declarations. Header files must not contain executable code. Doing so can lead to obscure multiply-defined errors at link time.

There is no uniform naming convention for header files. In C, people generally use the `.h` file name extension. For C++, some people continue to use `.h`. This often works okay, but it can lead to problems with projects that mix modules written in C with those written in C++.

An unambiguous convention is to restrict `.h` to C header files and to use `.hpp` for C++ header files. *We will use that convention in this course.*

What's in an implementation file?

Implementation (`.cpp`) files contain **definitions** of functions and constants that comprise the actual runnable code.

Each compiled definition must appear in exactly one object file. If it appears in more than one, the linker will generate a multiply-defined error.

For this reason, *definitions* are never put in header files.[1]

[1] Template classes are an exception to this rule, but for non-obvious reasons deriving from how the compiler handles templates.

Compiling in linux

The Zoo machines have two different C++ compilers installed: `g++` and `clang++`. Both are good compilers.

`g++` is the venerable Gnu C++ compiler. It is fast and generally very good.

`clang++` is a newer, more modular, compiler. It is slower to run than `g++` but sometimes may give better object code. It also gives different error messages which sometimes are clearer than those from `g++` (and sometime they are less clear).

You may find both compilers useful in developing your code. However, the final result must run using `g++`, and your `makefile` must be written to ensure that `g++` will be used.

Invoking the compiler

`g++` and `clang++` are commands used to invoke the corresponding compilers. However, depending on the command line switches given, they can be instructed to compiler and/or link several modules with one invocation.

For example,

```
g++ -o mycommand mod1.cpp mod2.cpp mod3.cpp
```

will compile all three `.cpp` files and then link the results togeter to produce an executable file `mycommand`. On the other hand, when used with the `-c` switch,

```
g++ -c -o mod1.o mod1.cpp
```

compiles the one module `mod1.cpp` to produce the single object file `mod1.o`.

Linking

When used without the `-c` switch, `g++` calls the linker `ld` to build an executable.

- If all command line arguments are object files, `g++` just does the linking.
- If one or more `.cpp` files appear on the command line, `g++` first compiles them and then links the resulting object files together with any `.o` files given on the command line. In this case, `g++` combines compilation and linking, and it does not write any new object files.

In both cases, the linker completes the linking task by searching libraries for any missing (unresolved) functions and variables and linking them into the final output.

System libraries

System libraries are often found in directories `/lib`, `/lib64`, `/usr/lib`, or `/usr/lib64`, but they can be placed anywhere as long as the *linker* is told where to find them.

The linker knows where to find the standard system libraries, and it searches the basic libraries automatically. Many other libraries are not searched unless specifically requested by the `-L` and `-l` linker flags.

One-line compilation

Often all that is required to compile your code is the single command

```
g++ -o myapp -O1 -g -Wall -std=c++17 *.cpp
```

The switches have the following meanings:

- `-o` name the output file;
- `-O1` do first-level optimization (which improves error detection);
- `-g` add symbols for use by the debugger;
- `-Wall` gives all reasonable warnings;
- `-std=c++17` tells the compiler to expect code in the C++17 language dialect.

The job of the project manager

As we've seen, a project consists of many different files. Keeping track of them and remembering which files and switches to put on the command line can be a major chore.

Project maintenance tools such as `make` and **Integrated Development Environments (IDEs)** are used to aid in this task.

Command line development tools

At the very least, you should become familiar with the basic tools for maintaining and building projects:

- A text editor such as `emacs` or `vi`.
- The compiler suite `g++`.
- The project manager `make`.

`clang++` is a newer alternative to `g++`. There are indications that it produces slightly better error messages and slightly better code than `g++`, but both compilers are very good and are suitable for use in this course. (The MacIntosh Xcode development system now defaults to `clang++`.)

Parts of a simple project

- Header file: `tools.hpp`
- Implementation files: `main.cpp`, `tools.cpp`
- Object files: `main.o`, `tools.o`
- Executable: `myapp`

Object files are built from implementation files and header files.

The executable is built from object files.

The `Makefile` describes how.

Dependencies

Whenever a source file is changed, the object files and executables that are directly or indirectly produced from it become out of date and must be rebuilt. Those files are called **dependencies** of the source file.

`make` uses dependency information stored in `Makefile` to avoid rebuilding files that have *not* changed since the last build. It only recompiles and/or relinks those files that are older than a file that they depend on.

`make` uses file modification dates for this purpose, so if those dates are off, `make` might fail to rebuild a file that is actually out of date.

A sample Makefile

```
#-------------------------------------------------------------
# Macro definitions
CXXFLAGS = -O1 -g -Wall -std=c++17
OBJ = main.o tools.o
TARGET = myapp
#-------------------------------------------------------------
# Rules
all: $(TARGET)
$(TARGET): $(OBJ)
        $(CXX) -o $@ $(OBJ)
clean:
        rm -f $(OBJ) $(TARGET)
#-------------------------------------------------------------
# Dependencies
main.o: main.cpp tools.hpp
tools.o: tools.cpp tools.hpp
```

Parts of a Makefile

A Makefile has three parts:

1. Macro definitions.
2. Rules.
3. Dependencies.

Syntax peculiarities:

- Lines beginning with `#` are comments.
- Indented lines must start with a `tab` character.

Macros

```
CXXFLAGS = -O1 -g -Wall -std=c++17
OBJ = main.o tools.o
TARGET = myapp
```

Macros are named strings.

- `CXXFLAGS` is added to the `g++` command line in **implicit rules**. Here we want level-1 optimization, symbols for the debugger, all warnings, and dialect c++17.
- `OBJ` lists the object files for our application.
- `TARGET` lists the final product (command).

Rules

```
all: $(TARGET)
$(TARGET): $(OBJ)
        $(CXX) -o $@ $(OBJ)
clean:
        rm -f $(OBJ) $(TARGET)
```

Rules tell how to build product files.

1. To build `all`, first build everything listed in `TARGET`.
2. To build `TARGET`, first build the `.o` files in `OBJ`. Then call the linker to create `TARGET` from the files in `OBJ`.
3. To build `clean`, generated files, delete everything in `OBJ` and `TARGET`.

Rules

```
all: $(TARGET)
$(TARGET): $(OBJ)
        $(CXX) -o $@ $(OBJ)
clean:
        rm -f $(OBJ) $(TARGET)
```

Notes:

- `CXX` is predefined to be the system default C++ compiler.
- `$@` is a special macro that refers the target of the current rule (`myapp` in the above example).
- `$(name)` refers to the definition of macro *name*.

Dependencies

```
main.o: main.cpp tools.hpp
tools.o: tools.cpp tools.hpp
```

Dependencies are a kind of degenerate rule.

- To build `main.o`, first "build" `main.cpp` and `tools.hpp`.
- To build `tools.o`, first "build" `tools.cpp` and `tools.hpp`.

But those dependencies are source files, so there is nothing to build. And where is the rule to build `main.o`?

What make does is compare the file modification dates on the target and on the dependencies in order to know if the target needs to be rebuilt.

Implicit rules

To build a target such as `main.o` for which there is no explicit rule, `make` uses an **implicit rule** that knows how to build any `.o` file from the corresponding `.cpp` file. In this case, the implicit rule invokes the `$(CXX)` compiler to produce output `main.o`. The compiler is called with the switches listed in `$(CXXFLAGS)`.

Integrated Development Environments

Graphical development tools: IDEs

Integrated Development Environments provide graphical tools to aid the programmer in many common tasks:

- Manage source files comprising a project;
- Display syntactic structure while editing;
- Search/replace over multiple files;
- Easy refactoring;
- Identifier completion;
- Display compiler error output in more readable form;
- Simplify edit-compile-run development cycle;

Recommended IDE's

Eclipse/CDT is a powerful, well-supported IDE that runs on many different platforms. Xcode is an Apple-proprietary IDE that only runs on Macs. Mac users may prefer it for its greater stability and even more features. I recommend either of these for serious C++ code development.

Geany is a lightweight IDE. It starts up much faster and is much more transparent in what it does. It should be more than adequate for this course.

Both Eclipse and Geany are installed on the Zoo, ready for your use.

The early part of this course can be perfectly well done in Emacs, so you don't have to learn Eclipse or Geany in order to get started.

Integrated Development Environment (e.g., Eclipse)

Advantages

- Supports notion of *project* — all files needed for an application.
- Provides graphical interface to all aspects of code development.
- Automatically creates `makefile`.
- Builds project with a mouse click or keyboard shortcut.
- Analyzes code as it is being written. Provides helpful feedback.
- Allows easy navigation among project components.
- Error comments are linked back to source code.

Integrated Development Environment (e.g., Eclipse)

Disadvantages

- Complicated to learn how to use — big learning curve.
- "Simple" things can become complicated for the non-expert (e.g., providing compiler flags) or making the font larger.
- Metadata can become inconsistent and difficult to repair.

Integrated Development Environment

If you use an IDE, before submitting your assignment, you should:

1. Copy your source code and test data files from the IDE to a separate `submit` directory *on the Zoo*.
2. Create a `Makefile` to build your project.
3. Test that everything works. Type `make` to make sure the project builds. Then run the resulting executable on your test suite to make sure it still does what you expect.

Submission Instructions

Submitting your assignments

1. Create a submission directory in your Zoo account named `ps1-netid123`, where you replace "ps1" with the current assignment number and "netid123" with your own net id.
2. Copy into it all the files you intend to submit.
3. Type `make` in that directory to make sure all needed files are present and your program builds and runs correctly.
4. Create required output files from your test runs.
5. Create a notes file that describes the submitted files.
6. Go up a level and create a gzipped tar file `ps1-netid123.tar.gz` using the command `tar -czvf ps1-netid123.tar.gz ps1-netid123`.
7. Submit the file `ps1-netid123.tar.gz` using Canvas.