CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 2
August 30, 2019
Task List

C++ Overview
  C++ Language Design Goals
  Comparison of C and C++

Describing an Application
  C/C++ Compilation Model

Source Code

Compiling and Linking
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 as soon as it becomes available.
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.
C++ Language Design Goals

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.
Describing an Application
What is a project?

A **C/C++ project** is a collection of files that work together to define an **application**.

In order to run an application, the files must first be processed by the system.

C/C++ requires three stages of processing:

1. **Compilation** produces a **module** from a collection of **source files**.
2. **Linking** produces a runnable piece of code called an **application** (or **program** or **executable** or **command**) from a collection of modules.
3. **Execution** loads an application into the computer and runs it.
File types

A compilation module is a collection of header files (.h or .hpp) and a single implementation file (.c or .cpp) that can be compiled to produce an object file (.o) file.

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 are 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.
Source Code
Source files

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

The implementation file contains the source code for your program. Header files provide the context to the compiler for understanding the implementation file. The `#include` directives name header files 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.
What goes in header files?

Header files contain class, data, function, and other **declarations** that are needed by the **client** of the module. They are 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

¹Template classes are an exception to this rule, but for non-obvious reasons deriving from how the compiler handles templates.
Compiling and Linking
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 and linkers. Depending on the command line switches given, they can be instructed to compile 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 together 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. Some libraries are only searched when 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.