CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 3
January 26, 2016

Project management (cont.)

Parts of a simple project

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

Object files are built from implementation files and header files.

The executable is built from object files.

The `Makefile` describes how.

Make also keeps track of **dependencies**. When a header or implementation file changes, only those object and executable files that depend on it need to be rebuilt.

A sample Makefile

```
#-------------------------------------------------------------
# Macro definitions
CXXFLAGS = -O1 -g -Wall -std=c++11
OBJ = main.o tools.o
TARGET = aboutme
#-------------------------------------------------------------
# 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++11
OBJ = main.o tools.o
TARGET = aboutme
```

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++11.
- `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 `main.o` and `tools.o`. Then call the linker to create `TARGET` from `main.o` and `tools.o`.
3. To 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 (`aboutme` in the above example).
- Uses of macros start with $ and enclose the name in parenthesis.

Dependencies

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

Dependencies are kind of degenerate rules.

- 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.

When no corresponding rule is specified, a **implicit rules** is invoked. Make knows that to build a `.o` file from a `.cpp` file, one must invoke the compiler.

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 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.

Submitting your assignments

Regardless of how you prepared your code, you should follow these instructions when you submit your assignment.

1. Type `make` in your Zoo submission directory to make sure your program builds and runs correctly.
2. Cut and past the output from your test runs into output files.
3. Create a notes file that describes the submitted files.
4. zip or gzip and tar the entire directory into a compressed archive file. The name should be of the form `ps1-netid123.zip` or `ps1-netid123.tar.gz`, where you replace "ps1" with the current assignment number and "netid123" with your own net id.
5. Submit the archive file using classes*v2.

Insertion Sort Example

Generic Insertion Sort

We give three implementations of simple insertion sort:

1. **C version:** Written in object-oriented style to the extent possible in C.
2. **C++ version:** Similar objected-oriented code but with C++ support.
3. **Monolithic C++ version:** All of the executable code has been placed in one big main program. All of the object-oriented structure and modularity code has been stripped out, leaving behind an opaque monolithic piece of code.

Click on the links below to see the code for three demos:

- 03-InsertionSortC
- 03-InsertionSortCpp
- 03-InsertionSortMonolith

C++ version

We look at the C++ version in some detail.

This will be a whirlwind tour of **classes** in C++, which we will be covering in greater detail in the coming lectures.

dataPack.hpp

```
#pragma once
```

A more efficient but non-standard replacement for include guards:

```
#ifndef DATAPACK_H
#define DATAPACK_H
// rest of header
#endif
```

class DataPack

```
class DataPack {
...
};
```

defines a new class named `DataPack`.

By convention, class names are capitalized.

Note the *required* semicolon following the closing brace.

If omitted, here's the error comment:

../datapack.hpp:11: error: new types may not be defined in a return type
../datapack.hpp:11: note: (perhaps a semicolon is missing after the definition of 'DataPack')
../datapack.cpp:12: error: two or more data types in declaration of 'readData'

Class elements

- A class contains declarations for *data members* and *function members* (or *methods*).
- `int n;` declares a data member of type `int`.
- `int getN(){ return n; }` is a complete member function definition.
- `void sortData();` declares a member function that must be defined elsewhere.
- By convention, member names begin with lower case letters and are written in camelCase.

Inline functions

- Methods defined inside a class are *inline* (e.g., `getN()`).
- Inline functions are recompiled for every call.
- Inline avoids function call overhead but results in larger code size.
- `inline` keyword makes following function definition inline.
- Inline functions must be defined in the header (.hpp) file. Why?

Visibility

- The visibility of declared names can be controlled.
- `public:` declares that following names are visible outside of the class.
- `private:` restricts name visibility to this class.
- Public names define the interface to the class.
- Private names are for internal use, like local names in functions.

Constructor

A *constructor* is a special kind of method.

Automatically called whenever a new class instance is allocated.

Job is to initialize the raw data storage of the instance to become a valid representation of an initial data object.

In `dataPack` example, `store` must point to storage of `max` bytes, `n` of which are currently in use.

Constructor

```
DataPack(){
    n = 0;
    max = LENGTH;
    store = new BT[max]; cout << "Store allocated.\n";
    readData();
}
```

`new` does the job of `malloc()` in C.

`cout` is name of standard output stream (like `stdout` in C).

`<<` is output operator.

`readData()` is private function to read data set from user.

Design question: Why is this a good idea?

Destructor

A *destructor* is a special kind of method.

Automatically called whenever a class instance about to be deallocated.

Job is to perform any final processing of the data object and to return any previously-allocated storage to the system.

In `dataPack` example, the storage block pointed to by `store` must be deallocated.

Destructor

```
~DataPack(){
    delete[] store;
    cout << "Store deallocated.\n";
}
```

Name of the destructor is class name prefixed with `~`.

`delete` does the job of `free()` in C.

Empty square brackets `[]` are for deleting an array.

dataPack.cpp

Ordinary (non-inline) functions are defined in a separate *implementation file*.

Function name must be prefixed with class name followed by `::` to identify which class's member function is being defined.

Example: `DataPack::readData()` is the member function `readData()` declared in class `DataPack`.

File I/O

C++ file I/O is described in Chapter 3 of textbook. Please read it.

`ifstream infile( filename );` creates and opens an input stream `infile`.

The Boolean expression `!infile` is true if the file failed to open.

This works because of a built-in coercion from type `ifstream` to type `bool`. (More later on coercions.)

`readData()` has access to the private parts of class `dataPack` and is responsible for maintaining their consistency.

main.cpp

As usual, the header file is included in each file that needs it:
`#include "datapack.hpp"`

`banner();` should be the first line of every program you write for this course. It helps debugging and identifies your output. (Remember to modify `tools.hpp` with your name as explained in Chapter 1 of textbook.)

Similarly, `bye();` should be the last line of your program before the return statement (if any).

The real work is done by the statements `DataPack theData;` and `theData.sortData();`. Everything else is just printout.

Manual compiling and linking

One-line version

```
g++ -o isort main.cpp datapack.cpp tools.cpp
```

Separate compilation

```
g++ -c -o main.o main.cpp
g++ -c -o datapack.o datapack.cpp
g++ -c -o tools.o tools.cpp
g++ -o isort main.o datapack.o tools.o
```

Compiling and linking using `make`

The sample Makefile given earlier is easily adapted for this project.

Compare the Makefile on the next slide with the sample.

```
#-----------------------------------------------------
# Macro definitions
CXXFLAGS = -O1 -g -Wall -std=c++11
OBJ = main.o datapack.o tools.o
TARGET = isort
#-----------------------------------------------------
# Rules
all: $(TARGET)
$(TARGET): $(OBJ)
        $(CXX) -o $@ $(OBJ)
clean:
        rm -f $(OBJ) $(TARGET)
#-----------------------------------------------------
# Dependencies
main.o: main.cpp datapack.hpp tools.hpp
datapack.o: datapack.cpp datapack.hpp tools.hpp
tools.o: tools.cpp tools.hpp
```