CS11 Intro C++

Spring 2018 – Lecture 4
Build Automation

• When a program grows beyond a certain size, compiling gets annoying...
  
  g++ -std=c++14 -Wall units.cpp testbase.cpp \ 
  hw3testunits.cpp -o hw3testunits
  g++ -std=c++14 -Wall units.cpp convert.cpp -o convert

• Also, if only units.cpp changes, why recompile testbase.cpp / 
  hw3testunits.cpp / convert.cpp source files?

• Typical development process:
  • Write or modify some code
  • Compile
  • Test
  • Repeat until done...

• Automating this process saves lots of time and effort
**make**

- **make** is a standard tool for automating builds
  - Command-line utility, very ubiquitous!
  - Takes input files and produces output files, based on a “makefile”
  - Several versions of **make**: GNU, BSD, ...

- **make** is often used for C and C++ projects
  - Sometimes other build tools are used for C/C++
  - CMake is becoming increasingly popular
  - Visual C++ provides **nmake** command-line build program
  - Other languages typically have their own build tools
• **make** requires a **makefile** that describes how to build your program
  • Typical filenames are **Makefile** (preferred) or **makefile**
  • Can specify a nonstandard makefile name with:
    ```
    make -f some-other-makefile
    ```

• The makefile describes **build targets**
  • Files that need to be generated from other files

• Each target specifies its **dependencies** – the files needed to build the target

• Can also specify how to build the target from its dependencies
Example Makefile

• Example **Makefile**:

```makefile
convert : units.o convert.o
g++ -std=c++14 -Wall units.o convert.o \ 
-o convert

units.o : units.cpp units.h
  g++ -std=c++14 -Wall -c units.cpp

... (more rules for other .o files)

clean :
  rm -f convert hw3testunits *.o *~
```

• Lines are indented with tab characters – spaces won’t work!
• A line can be wrapped to next line by ending with \n• Can specify multiple commands in a rule, as long as rules are separated by blank lines
Running `make`

- When `make` is run, it automatically looks for the makefile in the current directory
- `make` will automatically try to build the first target specified in the makefile

- Usually, the first target in the makefile is named `all`, and it builds everything of interest
  - `all : convert hw3testunits`
  - (this rule doesn’t need to specify any commands)

- Can optionally specify one or more build targets to `make`:
  - `make clean convert`
Real Build Targets

• From our example makefile:
  ```
  units.o : units.cpp units.h
  g++ -std=c++14 -Wall -c units.cpp
  ```

• In this case, `units.o` is a **real file**

• **make** will only build what is **needed**
  • If a target file’s date is older than any dependency, **make** will rebuild that target
  • **make** will only rebuild the parts of the program that **actually changed**

• To force a file to be rebuilt, you can **touch** it
  ```
  touch units.cpp
  ```
  • Sets file’s modification-time to current system time
  • Touching a nonexistent file will create a new empty file
Phony Build Targets

• From our example:
  
  clean :
    rm -f convert hw3testunits *.o *~

• In this case, clean is not a real file

• What if there happened to be a file named clean?
  
  • Our rule wouldn’t run!
  • make would see the “build-target” file, with no dependencies, and assume that nothing needed to be done

• Use .PHONY to say that the clean target isn’t a real file

  .PHONY: clean
  
  • Now if a file named clean exists, make ignores it
  • (The all target should also be marked as phony...)
Chains of Build Rules

• *make* figures out the graph of dependencies

  ```
  convert : units.o convert.o
  g++ -std=c++14 -Wall units.o convert.o \ 
  -o convert
  ```

• If any of `convert`’s dependencies don’t exist, *make* will use their build rules to make them

  ```
  units.o : units.cpp units.h
  g++ -std=c++14 -Wall -c units.cpp
  ```

• *make* will give up if:
  • A dependency can’t be found, and there’s no build rule that shows how to make it
  • It finds a cycle in the graph of dependencies
Makefile Variables

• Makefiles can define variables
  
  \texttt{CONVERT\_OBJS = units.o convert.o}

• Can use variables in build rules
  
  \texttt{convert : $(CONVERT\_OBJS)}
  
  \texttt{g++ $(CONVERT\_OBJS) -o convert}

• \texttt{$(\text{var-name})}$ tells \texttt{make} to expand the variable
  
  • Use variables to avoid listing the same things over and over again, all over the place
  
  • Same reasons as code reuse: state things \texttt{once}, so we only have to change things in one place

• Makefile variable names are usually \texttt{ALL\_CAPS}
Implicit Build Rules

• **make** already knows how to build certain targets
  • Those targets have built-in rules for building them
  • These built-in rules are called *implicit build rules*

• Example:
  • A makefile has **units.o** as a dependency, but no corresponding build rule
  • If **units.c** exists, **make** uses **gcc** to generate **units.o**
  • If **units.cpp** exists, **make** uses **g++** to generate **units.o**

• **make** has quite a few built-in implicit build rules!
  • Read **make** documentation for more details
Using Implicit Build Rules

• Implicit build rules make your makefiles **much** shorter
  
  ```
  CONVERT_OBJS = units.o convert.o
  ```

  ```
  all : convert hw3testunits
  ```

  ```
  convert : $(CONVERT_OBJS)
  g++ -std=c++14 -Wall $(CONVERT_OBJS) \ 
  -o convert
  ```

  ```
  clean :
  rm -f convert hw3testunits *.o *~
  ```

  ```
  .PHONY: all clean
  ```

• Can leave out the rules for all the object files!
Definitions of Implicit Rules

• Example definitions of implicit build rules:
  
  # C compilation implicit rule
  %.o : %.c
  $(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@

  # C++ compilation implicit rule
  %.o : %.cpp
  $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@

• Variables are used for compiler and options!
  • **CC** is the C compiler to use, **CXX** is the C++ compiler to use
  • **CFLAGS** are C compiler options, **CXXFLAGS** are C++ compiler options
  • **CPPFLAGS** are the preprocessor flags
  • Default values are for **gcc** and **g++**
Leveraging Variables in Implicit Rules

• We want to use the implicit-rule variables in our makefiles! 😊

• Example: specify `-Wall` and `-std=c++14` for compilation
  
  ```
  CXXFLAGS = -Wall -std=c++14
  CONVERT_OBJS = units.o convert.o
  
  all : convert hw3testunits
  
  convert : $(CONVERT_OBJS)
    $(CXX) $(CXXFLAGS) $(CONVERT_OBJS) \ 
   -o convert $(LDFLAGS)
  
  clean :
    rm -f convert hw3testunits *.o *
  
  .PHONY : all clean
  ```
Definitions of Implicit Rules (2)

• Examples of implicit build rules:
  
  # C++ compilation implicit rule
  
  %.o : %.cpp

  $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@

• Special syntax for pattern-matching
  
  • % matches the filename
  
  • $< is the first prerequisite in the dependency list
  
  • @$ is the filename of the target

• These $... values are called **automatic variables**
  
  • Other automatic variables too!
  
  • e.g. $^ is list of all prerequisites in the dependency list
Using Automatic Variables

- Can use automatic variables to link our program
  
  \[
  \text{CXXFLAGS} = -Wall \ -\text{std}=c++14 \\
  \text{CONVERT\_OBJJS} = \text{units}\.o \ \text{convert}\.o
  \]

  \[
  \text{all} : \ \text{convert} \ \text{hw3testunits}
  \]

  \[
  \text{convert} : \ \$(\text{CONVERT\_OBJJS}) \\
  \quad \$(\text{CXX}) \ $(\text{CXXFLAGS}) \ ^\^ \ \text{-o} \ \$\@ \ $(\text{LDFLAGS})
  \]

  \[
  \text{clean} : \\
  \quad \text{rm} \ -f \ \text{convert} \ \text{hw3testunits} \ *.o \ *~
  \]

  \[
  \text{.PHONY} : \ \text{all} \ \text{clean}
  \]
make Reference

• For more details, see the GNU make manual
  • http://www.gnu.org/software/make/manual/
Automatic Document Generation

- Automating API-doc generation is a very powerful technique
  - Comment your code according to a specified style
  - Run a documentation-generator on your code
  - Produces API documentation of your code, in HTML, PDF, etc. formats, ready for distribution!
- The documentation is in one place – your source
  - Tools can use the code as well as your comments in the generated output
- Several different options for doc-generation
- We will use doxygen: [http://www.doxygen.org](http://www.doxygen.org)
Doxygen Configuration

• Doxygen is driven by a config file
  • It will generate a template file for you:
    \texttt{doxygen -g \ [filename]}
  • Default filename is \texttt{Doxyfile}

• Customize the config file for your project
  • Set different configuration parameters as needed
  • Parameters are well documented in the config file

• Parameter names are \texttt{ALL\_CAPS}
  • (just like makefile variables)
  • Parameter-value can extend to next line, if current line ends with \ (backslash) character
  • Switches are specified with \texttt{YES} or \texttt{NO}
Doxygen Config Tips

• You should set:
  • INPUT (input files/directories)
  • OUTPUT_DIRECTORY (where results go)
  • PROJECT_NAME

• Other good settings to use:
  • JAVADOC_AUTOBRIEF = YES
  • EXTRACT_ALL = YES
  • EXTRACT_PRIVATE = YES
  • EXTRACT_STATIC = YES
Commenting Your Code

• Several different formats are recognized
  
  /**
   * This is a comment for my class. It is spiffy.
   */
  
  class MyClass { ... };

• /** starts the comment (javadoc style)
• Can also start with /*! (Qt style)
• Also several other options (see doxygen manual)

• Classes, types, functions have a brief comment, and a detailed comment
  
  • If JAVADOC_AUTOBRIEF is defined in doxygen config, first sentence is used as brief comment.
  • Otherwise, must use \brief keyword in your comments
Structural Commands

• “Structural commands” specify what a comment is associated with
  • “This is a comment for the source file.”
  • “This is a comment for class C.”
  • “This is a comment for parameter x of the function.”
  • etc.

• Allows Doxygen comments to be separated from entities that are being commented. (Not always recommended...)

• Two different formats for structural commands
  • Doxygen format: \cmd
  • Javadoc format: @cmd
  • Can use either format, but be consistent! 😊
What Can Be Commented?

• Files can be given comments
  • Must do this for doxygen to pick up certain comments
  • Examples:
    /*!
    \file ... */
    \file ... */
    (Qt/Doxygen format)
    (Javadoc format)

• Any type can be given a doxygen comment
  • Classes, structs, enums, typedefs, unions, namespaces

• Comment should immediately precede the type
  • ...unless you are using structural commands

• Preprocessor definitions can also be commented!
  • #define symbols, macros
Commenting Variables and Functions

• Global/static variables, and member variables
  • Comments can precede the variable:
    ```
    /** My special widget. */
    SpecialWidget sw;
    ```
  • Or they can follow the variable, on the same line:
    ```
    SpecialWidget sw; /**< My special widget. */
    ```
  • (Note the `<` character)

• Functions and their parameters/return values
  • Parameters follow this pattern:
    ```
    @param name Description
    \param name Description
    ```
  • Return value is documented with `\return` or `@return`
Running Doxygen

• Doxygen is simple to run:
  ```
doxygen [filename]
  ```
  • `doxygen` uses `Doxyfile` if no config file is given
  • Basically no command-line arguments; config file contains all the details!

• Results are stored in output directory
  • Each format gets its own subdirectory
  • `html` for HTML output, `latex` for LaTeX, etc.
  • Can specify alternate output directories if desired.
Doxygen References

• For more details, see the doxygen manual
  • http://www.stack.nl/~dimitri/doxygen/manual.html
  • http://www.doxygen.org
This Week’s Homework

• Write a **Makefile** for your project
  • Build *convert* and *hw3testunits* from their sources
  • Create an *all* target and a *clean* target
  • Create a *test* target that runs *hw3testunits*
  • Make sure that everything works properly

• Update your documentation to use Doxygen style comments
  • Create a **Doxyfile** configuration file
  • Add a *docs* build rule that generates HTML documentation