Large Program Management: Make; Ant
Motivation

- single-file programs do not work well when code gets large
  - compilation can be slow
  - hard to collaborate between multiple programmers
  - more cumbersome to edit

- larger programs are split into multiple files
  - each file represents a partial program or module
  - modules can be compiled separately or together
  - a module can be shared between multiple programs

- but now we have to deal with all these files just to build our program...
Compiling: Java

- What happens when you compile a Java program?

$ javac Example.java

- Example.java is compiled to create Example.class
- The class file is then run with java: java Example
## Compiling: C

<table>
<thead>
<tr>
<th>command</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>GNU C compiler</td>
</tr>
</tbody>
</table>

- to compile a program, type:
  
  ```
  gcc -o target source.c
  ```

  (where `target` is the name of the executable program to build)
  
  - the compiler builds an actual executable file, not a `.class` like Java
  - example: `gcc -o hi hello.c`

- to run your program, just execute that file
  
  - example: `./hi`
Object files (.o)

- A .c file can be compiled into an object (.o) file with `-c`:

  ```
  $ gcc -c part1.c
  $ ls
  part1.c  part1.o  part2.c
  ```

  - A .o file is a binary blob of compiled C code that cannot be directly executed, but can be directly inserted into a larger executable later.

- You can compile a mixture of .c and .o files:

  ```
  $ gcc -o combined part1.o part2.c
  ```

  - Avoids recompilation of unchanged partial program files.
Header files (.h)

• **header**: A C file whose only purpose is to be included (java import)
  ▪ generally a filename with the .h extension
  ▪ holds shared variables, types, and function declarations
  ▪ similar to a java interface: contains function declarations but not implementations

• key ideas:
  ▪ every *name*.c intended to be a module (not a stand alone program) has a *name*.h
  ▪ *name*.h declares all global functions/data of the module
  ▪ other .c files that want to use the module will #include *name*.h
Compiling large programs

• compiling multi-file programs repeatedly is cumbersome:

  $ gcc -o myprogram file1.c file2.c file3.c

• retyping the above command is wasteful:
  ▪ for the developer (so much typing)
  ▪ for the compiler (may not need to recompile all; save them as .o)

• improvements:
  ▪ use up-arrow or history to re-type compilation command for you
  ▪ use an alias or shell script to recompile everything
  ▪ use a system for compilation/build management, such as make
**make**

- **make**: A utility for automatically compiling ("building") executables and libraries from source code.
  - a very basic compilation manager
  - often used for C programs, but not language-specific
  - primitive, but still widely used due to familiarity, simplicity
  - similar programs: ant, maven, IDEs (Eclipse), ...

- **Makefile**: A script file that defines rules for what must be compiled and how to compile it.
  - Makefiles describe which files depend on which others, and how to create / compile / build / update each file in the system as needed.
• **dependency** : When a file relies on the contents of another.
  - can be displayed as a *dependency graph*
  - to build `main.o`, we need `data.h`, `main.c`, and `io.h`
  - if any of those files is updated, we must rebuild `main.o`
  - if `main.o` is updated, we must update `project1`
make demo

• **figlet**: program for displaying large ASCII text (like banner).
  - [http://freshmeat.net/projects/figlet](http://freshmeat.net/projects/figlet)

• Let's download a piece of software and compile it with make:
  - download `.tar.gz` file
  - un-tar it
  - (optional) look at README file to see how to compile it
  - (sometimes) run `./configure`
    * for cross-platform programs; sets up make for our operating system
  - run `make` to compile the program
  - execute the program
Makefile rule syntax

target : source1 source2 ... sourceN
    command
    command
    ...

- source1 through sourceN are the dependencies for building target
- Example:

  myprogram : file1.c file2.c file3.c
  gcc -o myprogram file1.c file2.c file3.c

- The command line must be indented by a single tab
  - not by spaces; NOT BY SPACES! SPACES WILL NOT WORK!
Running make

$ make target
  * uses the file named Makefile in current directory
  * finds rule in Makefile for building target and follows it
    - if the target file does not exist, or if it is older than any of its sources,
      its commands will be executed

• variations:

  $ make
  * builds the first target in the Makefile

  $ make -f makefilename
  $ make -f makefilename target
  * uses a makefile other than Makefile
Exercise: Create a basic Makefile to build \{hello.c, file2.c, file3.c\}

- Basic works, but is wasteful. What happens if we change file2.c?
  - everything is recompiled. On a large project, this could be a huge waste
Making a Makefile

THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."

HEY! GET BACK TO WORK!

COMPILING!

OH. CARRY ON.

courtesy XKCD
Making a Makefile

- Exercise: Create a basic Makefile to build \{hello.c, file2.c, file3.c\}
  - Basic works, but is wasteful. What happens if we change file2.c?
    - everything is recompiled. On a large project, this could be a huge waste

- Augment the makefile to make use of precompiled object files and dependencies
  - by adding additional targets, we can avoid unnecessary re-compilation
Rules with no dependencies

myprog: file1.o file2.o file3.o
  gcc -o myprog file1.o file2.o file3.o

**clean:**
  rm file1.o file2.o file3.o myprog

• make assumes that a rule's command will build/create its target
  ▪ but if your rule does not actually create its target, the target will still not exist the next time, so the rule will always execute (**clean** above)
  ▪ **make clean** is a convention for removing all compiled files
Rules with no commands

\textbf{all: myprog myprog2}

\texttt{myprog: file1.o file2.o file3.o}
\hspace{1em} \texttt{gcc -o myprog file1.o file2.o file3.o}

\texttt{myprog2: file4.c}
\hspace{1em} \texttt{gcc -o myprog2 file4.c}
\hspace{1em} ...

- all rule has no commands, but depends on myprog and myprog2
  - typing \texttt{make all} will ensure that myprog, myprog2 are up to date
  - all rule often put first, so that typing \texttt{make} will build everything
- Exercise: add “clean” and “all” rules to our hello Makefile
Variables

\begin{itemize}
  \item \texttt{NAME = value} \hspace{1cm} (declare)
  \item \texttt{$(NAME)$} \hspace{1cm} (use)
  \end{itemize}

\texttt{OBJFILES = file1.o file2.o file3.o}
\texttt{PROGRAM = myprog}

\texttt{$(PROGRAM)$: $(OBJFILES)$}
  \texttt{gcc -o $(PROGRAM) $(OBJFILES)}

\texttt{clean:}
  \texttt{rm $(OBJFILES) $(PROGRAM)}

• variables make it easier to change one option throughout the file
  ▪ also makes the makefile more reusable for another project
More variables

OBJFILES = file1.o file2.o file3.o
PROGRAM = myprog
CC = gcc
CCFLAGS = -g -Wall

$(PROGRAM): $(OBJFILES)
  $(CC) $(CCFLAGS) -o $(PROGRAM) $(OBJFILES)

• many makefiles create variables for the compiler, flags, etc.
  ▪ this can be overkill, but you will see it "out there"
Special variables

$@
the current target file

$^
all sources listed for the current target

$<
the first (left-most) source for the current target

(there are other special variables)

myprog: file1.o file2.o file3.o
  gcc $(CCFLAGS) -o $@ $^

file1.o: file1.c file1.h file2.h
  gcc $(CCFLAGS) -c $<

• Exercise: change our hello Makefile to use variables for the object files and the name of the program
Auto-conversions

• rather than specifying individually how to convert every .c file into its corresponding .o file, you can set up an implicit target:

  ```
  # conversion from .c to .o
  .c.o:
    gcc $(CCFLAGS) -c $<
  ```

  "To create filename.o from filename.c, run gcc -g -Wall -c filename.c"

• for making an executable (no extension), simply write .c :

  ```
  .c:
    gcc $(CCFLAGS) -o $@ $<
  ```

• Exercise: simplify our hello Makefile with a single .c.o conversion
What about Java?

- Create Example.java that uses a class MyValue in MyValue.java
  - Compile Example.java and run it
    - javac automatically found and compiled MyValue.java
  - Now, alter MyValue.java
    - Re-compile Example.java... does the change we made to MyValue propagate?
      - Yep! javac follows similar timestamping rules as the makefile dependencies. If it can find both a .java and a .class file, and the .java is newer than the .class, it will automatically recompile
      - But be careful about the depth of the search...
  - But, this is still a simplistic feature. Ant is a commonly used build tool for Java programs giving many more build options.
Ant

• Similar idea to Make, though Ant uses build.xml instead of Makefile:

```xml
<project>
  <target name="name">
    tasks
  </target>

  <target name="name">
    tasks
  </target>
</project>
```

• Tasks can be things like:
  - `<javac ... />`
  - `<mkdir ... />`
  - `<delete ... />`
  - A whole lot more... [http://ant.apache.org/manual/tasksoverview.html](http://ant.apache.org/manual/tasksoverview.html)
Ant Example

- Create an Ant file to compile our Example.java program

- Running ant (assuming build.xml in current directory):
  
  $ ant **targetname**
Ant Example

• Create an Ant file to compile our Example.java program

```xml
<project>
  <target name="clean">
    <delete dir="build"/>
  </target>

  <target name="compile">
    <mkdir dir="build/classes"/>
    <javac srcdir="src" destdir="build/classes"/>
  </target>
</project>
```
Automated Build Systems

• Fairly essential for any large programming project

  ▪ Why? Shell scripts instead? What are these tools aiming to do?
  ▪ Is timestamping the right approach for determining “recompile”?
  ▪ What about dependency determination?
  ▪ What features would you want from an automated build tool?

  ▪ Should “building” your program also involve non-syntactic checking?
    • Ant can run JUnit tests...