COMPSCI 1XC3 - Computer Science Practice and Experience: Development Basics

Topic 9 - Makefiles, Testing and Static Analysis

Zheng Zheng

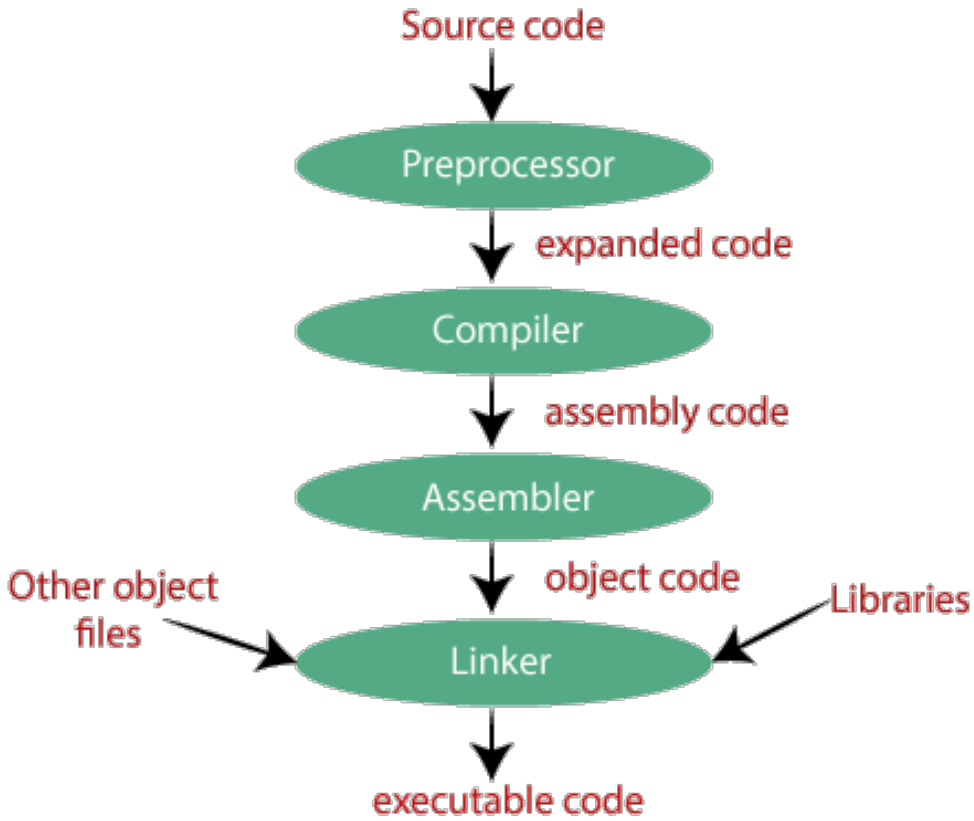
McMaster University

Winter 2023

Stages of Compilation

Recall the stages of compilation:

- Preprocessor directives (`include` and `define` etc) are resolved.
- Parsing & AST generation
- Assembly of object (machine) code
- Linking with other object files (if applicable).



Object Files

So what's the point of object files?

- An object file is machine code that was compiled from it's source code file, but *without* the code for any called functions in other source code files.
- Instead, object files contain *links to* the needed functions in the needed files and libraries.
- Object files are binary encoded, not character encoded, so there's no point trying to look at them with `cat` or `less`.

By compiling an object file for each source code file *separately*, the object files only need to be updated *if the source code is updated!* This can produce a massive speed boost in compilation time!

OBJECTion!!!

Using object files correctly requires effort. We can compile a batches of source code files to object files using glob patterns and the -c flag:

```
$ gcc -c *.c
```

But then when we go to compile our program from object code:

```
$ gcc -o main main.c lib1.o lib2.o lib3.o etc.o
```

We have to manually include our object files, which is tedious and time consuming. So we're saving compilation time, but losing time to longer commands.

There has to be a better way!

`make` me a sandwich!

`make` is a program which automates the compilation process.

- `make` compiles only those source code files that **need** to be recompiled to produce the executable.
- It does this by checking the time at which your source code files were last saved, and the time when the program was last compiled.
- `make` requires a special configuration file, called a `makefile`, which tells it which files to compile, which compiler to use, etc.

Makefiles are *not* Bash Scripts.

sudo make me a sandwich!

Makefiles are composed of **rules**.

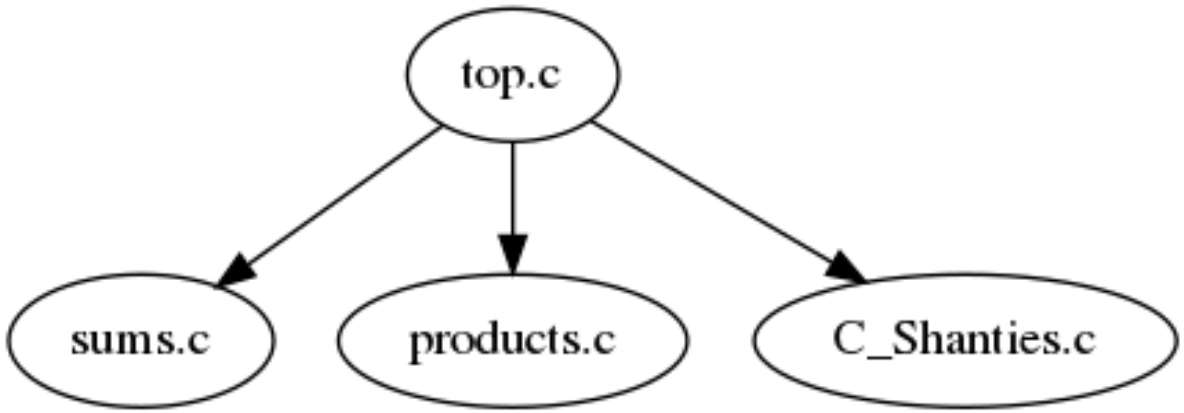
```
<target> : <prerequisites>
  <recipe>
```

- A `target` can be one of three things:
  - An executable file
  - An object file
  - A "Phony Target", which functions as a special command (i.e, `clean`).
- A `prerequisites` are the files that are used as input to create the target.
- A `recipe` is a Bash command that `make` performs to create the target.

Each recipe line must be started with a **tab** (`\t`) character. Spaces are invalid syntax!

A Simple Example

Recall the files we used in Lab 6 to create static and dynamic libraries.



We used lots of fancy compilation methods to create static and dynamic libraries, but you can also compile these using more simple invocations of gcc, such as

```
$ gcc -Wall -o top top.c sums.o products.o C_Shanties.o
```

Makefile-ification

If we write rules to produce all the object files and the final executable into a makefile, we would get the following:

```
top : top.c sums.o products.o C_Shanties.o
  gcc -Wall -o top top.c sums.o products.o C_Shanties.o

C_Shanties.o : C_Shanties.c
  gcc -Wall -c -o C_Shanties.o C_Shanties.c

products.o : products.c
  gcc -Wall -c -o products.o products.c

sums.o : sums.c
  gcc -Wall -c -o sums.o sums.c
```

How to use `make`

Now that we have our makefile in place, we can start compiling!
To compile the "default goal", simply type:

```
$ make
```

To compile any of the targets in the file, just specify the target:

```
$ make sums.o
$ make top
```

To invoke a phony target, again, just type the name:

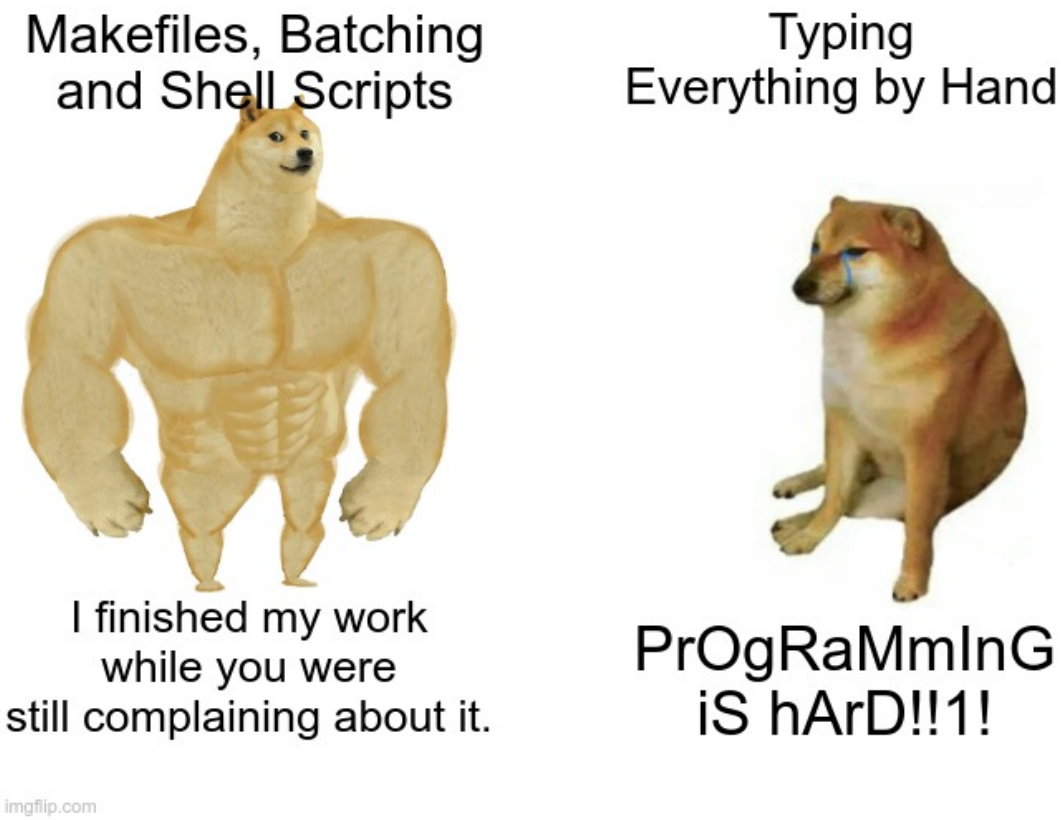
```
$ make cleanup
```

Nothing could be more simple!

So how does this Work?

- On a default invokation (i.e., just calling `make` with no arguments), `make` will use the first rule as its compilation target.
  - Therefore, your top file should be at the top!
- Whether the target is default or specified, `make` recursively (!) processes the rules for all prerequisites of the target rule.
- If the source file is newer than its corresponding object file, or if the object file doesn't exist, `make` will produce it.
  - For testing purposes, `touch` updates the timestamp on a file without modifying the contents.
- Any object file which needs to be linked to other object files will be regenerated as well, if any of the object files it needs are newer than itself.

Variables!


Makefiles, Batching
and Shell Scripts
I finished my work
while you were
still complaining about it.
imgflip.com
Typing
Everything by Hand
PrOgRaMmInG
iS hArD!!1!

Variables <3

As with so many things in life, we can make things once again simpler in the long term by complicating things in the short term.

- Makefiles support variables which are similar in many ways to the variables in shell scripting.
- Variables once again need the variable substitution operator `$()` to work.
  - This time, enclosing the variable name in round braces is considered good etiquette.
- You also don't have to worry about not leaving some whitespace this time around. (But don't go overboard).

When set up correctly, variables let control many things about our compilation processes from a few lines at the top of the file.

Abstract Out the Compiler!

```
CC = gcc

top : top.c sums.o products.o C_Shanties.o
  $(CC) -Wall -o top top.c sums.o products.o C_Shanties
    .o

C_Shanties.o : C_Shanties.c
  $(CC) -Wall -c -o C_Shanties.o C_Shanties.c

products.o : products.c
  $(CC) -Wall -c -o products.o products.c

sums.o : sums.c
  $(CC) -Wall -c -o sums.o sums.c
```

Abstract Out the Compiler Flags!

```
CC = gcc
Cflags = -Wall -o

top : top.c sums.o products.o C_Shanties.o
  $(CC) $(Cflags) top top.c sums.o products.o
    C_Shanties.o

C_Shanties.o : C_Shanties.c
  $(CC) -c $(Cflags) C_Shanties.o C_Shanties.c

products.o : products.c
  $(CC) -c $(Cflags) products.o products.c

sums.o : sums.c
  $(CC) -c $(Cflags) sums.o sums.c
```

Abstract Out the List of Objects!

```
CC = gcc
Cflags = -Wall -o
objects = sums.o products.o C_Shanties.o

top : top.c $(objects)
  $(CC) $(Cflags) top top.c $(objects)

C_Shanties.o : C_Shanties.c
  $(CC) -c $(Cflags) C_Shanties.o C_Shanties.c

products.o : products.c
  $(CC) -c $(Cflags) products.o products.c

sums.o : sums.c
  $(CC) -c $(Cflags) sums.o sums.c

# Makefile Comments Use Octothorpe BTW!
```

clean up your Act!

Although it isn't required, it is conventional to include a "cleanup" phony target, which deletes files used during executable generation, but which are not needed afterwards.

```
clean :
  rm -f $(objects)
```

- Specifying no prerequisites makes this a "stand alone" rule.
- The `-f` flag on `rm` means "force". It stops `rm` complaining about requests to delete non-existent files.

`all` in the Family

Another phony target typically included is "all". This isn't necessary unless a makefile has more than one final target (i.e., if there was more than one top file that could have gone in the top slot).

```
all_targets = executable1 executable2 executable3

all : $(all_targets)
# No recipe!
```

The "all" rule typically doesn't do anything by itself, it just queues up all valid final targets for regeneration by making them prerequisites.

This is Why We Test Things!

Automating things is both efficient and satisfying, so let's automate one of the more tedious programming tasks: **testing**!

- When testing a program, we compare the **expected output** of our program to the **actual output**.
- Of course, this only works if both outputs were produced from the same **input**.
- The expected output can be generated by a number of processes:
  - Human design
  - A different, perhaps less efficient algorithm
  - Customer Requirements

different Strokes for different Folks!

file1.txt

```
1 NUMBER_NODES=100
2 MAX=1024
3 MIN=1
4 SPECIAL_NODE=4
5 KEY_FILE=keys.txt
6 MAX_TIME=45s
7 SEARCH=INSERT
8 DELETE=RECURSION
```

file2.txt

```
1 NUMBER_NODES=100
2 MAX=1024
3 MIN=1
4 SPECIAL_NODE=5
5 KEY_FILE=keys.txt
6 MAX_TIME=45s
7 SEARCH=INSERT
8 DELETE=RECURSION
```

```
$ diff file1.txt file2.txt
4c4
< SPECIAL_NODE=4
---
> SPECIAL_NODE=5
```

Introducing `diff`

Up to now, we've been comparing our expected and actual outputs manually. This, however, is slow, tedious, and a bad use of human resources.

- The `diff` command automatically reports the differences between two files.
- Combined with output capture in Bash (>), we can run a program, collect it's output, and compare it to a pre-existing file that gives us our expected output:

```
$ ./foo > foo_actual.txt
diff foo_expected.txt foo_actual.txt
```

- And naturally, we can integrate this sequence of commands into both bash scripts and makefiles!

Using Exit Codes

Exit codes are used by commands and programs to indicate success or failure.

- In C, the return value of `main` is the exit status of the program.
- In Bash scripting, the exit status of the last command to run is stored in the special variable $?.

```
# cp_check.sh
cp $1 $2
if [ $? -eq 0 ]
then
  echo "Copy Succeeded."
else
  echo "Copy Failed."
fi
```

```
$ ./cp_check.sh none.c x/
cp: cannot stat 'none.c':
  No such file or directory
Copy Failed.
```

The `assert` Library

Output comparison using `diff` is great for automated black-box testing. However, many (even most) of the properties we may wish to test are *internal to the program*!

- The `assert` function, contained in the `assert.h` standard library, allows us to perform testing within a C program.
- If the condition fails, assert invokes `abort()`, which immediately terminates the program and returns a non-zero exit code.
- In addition assert reports to `stderr`:
  - the expression which failed the test
  - the line number of the failed assertion
  - which file the assertion is in

Assertion Example

```c
#include<stdio.h>
#include<assert.h>

int main () {
  char buff[50];
  printf("Enter the letter x... or else! ");
  scanf("%s", buff);
  assert(buff[0] == 'x');
  printf("Good job!\n");
}
```

```
nick:code$ ./a.out
Enter the letter x... or else! Make me!
a.out: an_assertion.c:8: main: Assertion `buff[0] == 'x
    '' failed.
Aborted (core dumped)
```

Application: Autograding

This is how the autograder for the assignments works:

- First, a program replaces your main function with one of our design, that calls your function for certain inputs, and makes assertions about the outputs it receives.
- Then, we compile your program.
  - Any code that fails to compile is reported as a failed test case.
- Your code is then executed.
  - If our inserted assertions fail, that test case is considered failed, and a zero is recorded.
  - If no assertions fail, the test case is passed, and you get however many points that test case was worth.
- We run several test cases per question.
- In this manner, we can process each assignment in about 1.5 seconds!

Looking for Common Problems

You may have noticed that C's compiler errors aren't as *explicit* as Python's. This functionality exists, just not in gcc.
Static Analysis Engines are pieces of software which *analyze* code without *executing* it.

- If you have to run the code to analyze, that's called **Dynamic Analysis**.

During Static Analysis, source code is preprocessed and parsed, but no executable code is generated.

- Instead, the abstract syntax tree itself is analyzed.
- Common patterns which normally indicate bugs or problems can then be identified and are reported to the user.

Introducing cppcheck

To quote the cppcheck manual:

- "Cppcheck is an analysis tool for C/C++ code. It provides unique code analysis to detect bugs and focuses on detecting undefined behaviour and dangerous coding constructs. The goal is to detect only real errors in the code, and generate as few false positives (wrongly reported warnings) as possible. Cppcheck is designed to analyze your C/C++ code even if it has non-standard syntax, as is common in for example embedded projects."

(source: `http://cppcheck.sourceforge.net/manual.pdf`)

Installing Cppcheck

To install cppcheck...

- Debian-family (Ubuntu, Mint, MX, etc)

```
$ sudo apt-get install cppcheck
```

- Fedora-family (Red hat, etc)

```
$ sudo yum install cppcheck
```

- To install on Macintosh:

```
$ brew install cppcheck
```

- If you're using the Pascal server, it's already installed!
- If you're on Windows, you can find an installer here: `http://cppcheck.sourceforge.net`
- Those wishing to have cppcheck installed directly into their brains will have to wait for driver support.

Limitations!

What it does

- Detects common error patterns
- Points out stylistic issues
- Detects when code is dangerous
- Tells you things like:
  - Out-of-bounds Array
  - Division by Zero
  - Useless conditionals
  - Unreachable Code
  - A full listing is available here (link).

What it doesn't

- Detect all bugs
- Compile your code
- Detect anything outside the specified patterns
- Detect semantic errors that are unrelated to "getting the math wrong."
- **Replace testing or careful design.**

For Example...

```c
#include <stdio.h>

int main () {
  int a[5] = {1,2,3,4,5};
  printf("a[10] = %d", a[10]);
  int x = 5 / 0;
  printf("5 / 0 = %d", 5.0/ 0);
}
```

```
$ cppcheck badcode.c
Checking badcode.c ...
[badcode.c:6]: (error) Array 'a[5]' accessed at index
    10, which is out of bounds.
[badcode.c:7]: (error) Division by zero.
```

There are lots of options for refining and customizing the analysis, for more information read the manual (link)!

Acknowledge

The contents of these slides were liberally borrowed (with permission) from slides from the Summer 2021 offering of 1XC3 (by Dr. Nicholas Moore).