Make Tutorial

Dr. Edward Brash

September 23, 2022

CPSC 256

Creating an Executable

- Several options:
- At the command line
  - Works ok for small source code distributions
  - Cumbersome – need to keep track of changes by hand – can be confusing even for small codes
- Within an IDE (Eclipse, NetBeans, XCode, etc.)
  - These will use some sort of underlying system, probably Make
- Building a Make system
  - Probably still the most popular option, and you will undoubtedly have to use it at some point!

What is Make?

- Make is a utility that automatically builds executable programs and libraries from source code by reading Makefiles which specify how to derive the target program
- On most Unix systems, Make is actually GNU Make, which can be used in conjunction with the GNU build system (which in turn allows some amount of autoconfiguration abilities)

What is Make?

- Make has its own internal language
- In my experience, it is rather cryptic, and is somewhat platform dependent (use of tabs)
- “Make is a popular but flawed tool.”
- It has a LOT of problems:
  - Make’s parser does not function in a normal way
  - Special tokens (commas, etc.) are not handled consistently
  - Whitespace (sometimes matters, sometimes doesn’t)
  - Undefined variables do not generate an error
  - Limited support for conditions (if … then … else)
  - No boolean data types
  - No scoping (all variables are global)

Example 1 – main.c

```
#include “stdlib.h”
#include “stdio.h”
#include "functions.h”

int main(){
    print_hello();
    printf("The factorial of 5 is %d\n”,factorial(5));
    return 0;
}
```

Example 1 – hello.c

```
#include “stdlib.h”
#include “stdio.h”
#include "functions.h”

void print_hello(){
    printf(“Hello World!\n”);
}
```

Example 1 – factorial.c

```
#include "functions.h”

int factorial(int n){
    if(n>1){
        return(n * factorial(n-1));
    }
    else return 1;
}
```

Example 1 – functions.h

```
void print_hello();
int factorial(int n);
```

At the command line

```
$   gcc -c main.c
$   gcc -c hello.c
$   gcc -c factorial.c
$   gcc -o hello main.o hello.o factorial.o

$ ./hello
Hello World!
The factorial of 5 is 120
```

How do we do this with Make?

- Create a file called Makefile
- Note the tab characters!!! Yikes!!!

```
all:
	gcc -c main.c
	gcc -c hello.c
	gcc -c factorial.c
	gcc -o hello main.o hello.o factorial.o
```

Problems

- Stupid … it makes EVERYTHING every time whether there have been changes or not … highly inefficient

- Uses four steps every time … gcc is smarter than this!

Modification 1 – Still stupid

```
all:
	gcc main.c hello.c factorial.c -o hello
```

Modification 2 – Separate things

```
all: hello

hello: main.o factorial.o hello.o
	gcc main.o factorial.o hello.o -o hello

main.o: main.c
	gcc -c main.c

factorial.o: factorial.c
	gcc -c factorial.c

hello.o: hello.c
	gcc -c hello.c

clean:
	rm -rf *.o hello
```

Modification 3 – Documentation!

```
#
# I am a comment … students have never seen me before
# I feel so unloved
# I was using # long before Twitter existed
#
all: hello

hello: main.o factorial.o hello.o
	gcc main.o factorial.o hello.o -o hello

main.o: main.c
	gcc -c main.c

factorial.o: factorial.c
	gcc -c factorial.c

hello.o: hello.c
	gcc -c hello.c

clean:
	rm -rf *.o hello
```

Modification 4 – Variables

```
# I am a comment, and I want to say that the variable CC will be
# the compiler to use.
CC=gcc
# Hey!, I am comment number 2. I want to say that CFLAGS will be the
# options I'll pass to the compiler.
CFLAGS=-c –Wall

all: hello

hello: main.o factorial.o hello.o
	$(CC) main.o factorial.o hello.o -o hello

main.o: main.c
	$(CC) $(CFLAGS) main.c

factorial.o: factorial.c
	$(CC) $(CFLAGS) factorial.c

hello.o: hello.c
	$(CC) $(CFLAGS) hello.c

clean:
	rm -rf *.o hello
```

Modification 5 – Lots of Sources

```
CC=gcc
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.c hello.c factorial.c
OBJECTS=$(SOURCES:.c=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LDFLAGS) $(OBJECTS) -o $@
.cpp.o:
	$(CC) $(CFLAGS) $< -o $@
clean:
	rm –rf $(OBJECTS) $(EXECUTABLE)
```

Modification 6 – Dependencies

```
CC=gcc
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.c hello.c factorial.c
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LDFLAGS) $(OBJECTS) -o $@
.cpp.o:
	$(CC) $(CFLAGS) $< -o $@
main.o : functions.h
hello.o : functions.h
factorial.o : functions.h
clean:
	rm –rf $(OBJECTS) $(EXECUTABLE)
```

Modification 7 – Wildcards

```
CC=gcc
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.c hello.c factorial.c
OBJECTS=$(SOURCES:.c=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LDFLAGS) $(OBJECTS) -o $@
.cpp.o:
	$(CC) $(CFLAGS) $< -o $@

$(OBJECTS) : functions.h

clean:
	rm –rf $(OBJECTS) $(EXECUTABLE)
```

Modification 8 – Multiple targets

```
CC=gcc
CFLAGS=-c -Wall
LDFLAGS=
FACSOURCE=factorial.c
SOURCES1=main.cpp hello.c
SOURCES2=states.c
FACOBJS=$(FACSOURCE:.cpp=.o)
OBJECTS1=$(SOURCES1:.cpp=.o)
OBJECTS2=$(SOURCES2:.cpp=.o)
EXECUTABLE1=hello
EXECUTABLE2=states

all: $(SOURCES) $(EXECUTABLE1) $(EXECUTABLE2)

$(EXECUTABLE1): $(FACOBJS) $(OBJECTS1)
        $(CC) $(LDFLAGS) $(OBJECTS1) $(FACOBJS) -o $@

$(EXECUTABLE2): $(FACOBJS) $(OBJECTS2)
        $(CC) $(LDFLAGS) $(OBJECTS2) $(FACOBJS) -o $@

.cpp.o:
        $(CC) $(CFLAGS) $< -o $@

$(OBJECTS1) : functions.h

$(OBJECTS2) : functions.h

$(FACOBJS) : functions.h

clean:
        rm –rf $(OBJECTS1) $(OBJECTS2) $(FACOBJS) $(EXECUTABLE1) $(EXECUTABLE2)
```

Version 0

```
all:
	gcc -c main.c
	gcc -c hello.c
	gcc -c factorial.c
	gcc -o hello main.o hello.o factorial.o
```

Future Crash Course?

- Multiple source, include, install directories
- “bin/src/include” directory model
- Linking with libraries
- Multiple Makefiles (for multiple source directories)
- Other build systems (SCons, cmake, etc.)
- GNU Autoconf