Make – Scripting Multi-File Builds

Dr. Robert Lowe

Division of Mathematics and Computer Science
Maryville College

Maryville COLLEGE

The Problem

- Most software applications require dozens, or even hundreds, of source files!

The Problem

- Most software applications require dozens, or even hundreds, of source files!
- The building of applications requires many complex commands.

The Problem

- Most software applications require dozens, or even hundreds, of source files!
- The building of applications requires many complex commands.
- This can become very unwieldy very quickly.

The Problem

- Most software applications require dozens, or even hundreds, of source files!
- The building of applications requires many complex commands.
- This can become very unwieldy very quickly.
- We solve this by using some sort of build system.

Multi-Stage Compilation

- Compiling the entire source every time is quite time consuming.

Multi-Stage Compilation

- Compiling the entire source every time is quite time consuming.
- Instead we split the compilation into two parts:

Multi-Stage Compilation

- Compiling the entire source every time is quite time consuming.
- Instead we split the compilation into two parts:
  1. Compile cpp files.

Multi-Stage Compilation

- Compiling the entire source every time is quite time consuming.
- Instead we split the compilation into two parts:
  1. Compile cpp files.
  2. Link cpp files together.

Multi-Stage Compilation

- Compiling the entire source every time is quite time consuming.
- Instead we split the compilation into two parts:
  1. Compile cpp files.
  2. Link cpp files together.
- We can do this by adding the `-c` option to `g++`

Let's Try It!

1. Change into the `examples/19-Classy` directory.

Let's Try It!

1. Change into the `examples/19-Classy` directory.
2. Compile each of the cpp files like this:

Let's Try It!

1. Change into the `examples/19-Classy` directory.
2. Compile each of the cpp files like this:
   - `g++ -c soda-machine.cpp`

Let's Try It!

1. Change into the `examples/19-Classy` directory.
2. Compile each of the cpp files like this:
   - `g++ -c soda-machine.cpp`
   - `g++ -c sodasim.cpp`

Let's Try It!

1. Change into the `examples/19-Classy` directory.
2. Compile each of the cpp files like this:
   - `g++ -c soda-machine.cpp`
   - `g++ -c sodasim.cpp`
3. List the directory. Note the **object files** this created.

Let's Try It!

1. Change into the `examples/19-Classy` directory.
2. Compile each of the cpp files like this:
   - `g++ -c soda-machine.cpp`
   - `g++ -c sodasim.cpp`
3. List the directory. Note the **object files** this created.
4. Now we will link the executable with the following:
   `g++ soda-machine.o sodasim.o -o sodasim`

Enter Make

- Linking object files is faster than compiling source files.

Enter Make

- Linking object files is faster than compiling source files.
- We only need to recompile the object files when the source file changes.

Enter Make

- Linking object files is faster than compiling source files.
- We only need to recompile the object files when the source file changes.
- This is still a heavy workload!

Enter Make

- Linking object files is faster than compiling source files.
- We only need to recompile the object files when the source file changes.
- This is still a heavy workload!
- This where the tool `make` comes in.

Enter Make

- Linking object files is faster than compiling source files.
- We only need to recompile the object files when the source file changes.
- This is still a heavy workload!
- This where the tool `make` comes in.
- `make` lets us script the build process in an intelligent way.

Enter Make

- Linking object files is faster than compiling source files.
- We only need to recompile the object files when the source file changes.
- This is still a heavy workload!
- This where the tool `make` comes in.
- `make` lets us script the build process in an intelligent way.
- `make` works by processing “recipes”.

Maryville COLLEGE

Enter Make

- Linking object files is faster than compiling source files.
- We only need to recompile the object files when the source file changes.
- This is still a heavy workload!
- This where the tool `make` comes in.
- `make` lets us script the build process in an intelligent way.
- `make` works by processing “recipes”.
- Recipes are either implicit or explicitly.

Maryville COLLEGE

Implicit Recipes

- Make is “smart enough” to build some things without extra input.

Implicit Recipes

- Make is “smart enough” to build some things without extra input.
- For instance, try the following:

Implicit Recipes

- Make is “smart enough” to build some things without extra input.
- For instance, try the following:
  1. Change into your `examples/01-Intro-C++` directory

Implicit Recipes

- Make is “smart enough” to build some things without extra input.
- For instance, try the following:
  1. Change into your `examples/01-Intro-C++` directory
  2. Run the command: `rm hello` to erase the old binary (if you have one)

Implicit Recipes

- Make is “smart enough” to build some things without extra input.
- For instance, try the following:
  1. Change into your `examples/01-Intro-C++` directory
  2. Run the command: `rm hello` to erase the old binary (if you have one)
  3. Type `make hello` and press enter.

Implicit Recipes

- Make is “smart enough” to build some things without extra input.
- For instance, try the following:
  1. Change into your `examples/01-Intro-C++` directory
  2. Run the command: `rm hello` to erase the old binary (if you have one)
  3. Type `make hello` and press enter.
  4. Now run `make hello` again.

Implicit Recipes

- Make is “smart enough” to build some things without extra input.
- For instance, try the following:
  1. Change into your `examples/01-Intro-C++` directory
  2. Run the command: `rm hello` to erase the old binary (if you have one)
  3. Type `make hello` and press enter.
  4. Now run `make hello` again.
- This is make’s implicit compilation rules. It is smart enough to convert a single file program into an executable.

Implicit Recipes

- Make is “smart enough” to build some things without extra input.
- For instance, try the following:
  1. Change into your `examples/01-Intro-C++` directory
  2. Run the command: `rm hello` to erase the old binary (if you have one)
  3. Type `make hello` and press enter.
  4. Now run `make hello` again.
- This is make’s implicit compilation rules. It is smart enough to convert a single file program into an executable.
- One more thing to try:

  ```
  make hello.o
  ```

Implicit Recipes

- Make is “smart enough” to build some things without extra input.
- For instance, try the following:
  1. Change into your `examples/01-Intro-C++` directory
  2. Run the command: `rm hello` to erase the old binary (if you have one)
  3. Type `make hello` and press enter.
  4. Now run `make hello` again.
- This is make’s implicit compilation rules. It is smart enough to convert a single file program into an executable.
- One more thing to try:

  `make hello.o`
- Pretty neat, huh?

Makefile – Explicit Recipes

- When we compile multiple files, we need to explicitly tell make how to go about doing it.

Makefile – Explicit Recipes

- When we compile multiple files, we need to explicitly tell make how to go about doing it.
- We do this by creating a file named "`Makefile`"

Makefile – Explicit Recipes

- When we compile multiple files, we need to explicitly tell make how to go about doing it.
- We do this by creating a file named "`Makefile`"
- In the `Makefile` we write a series of **recipes** in the following format:

```
target:  ingredient list
```

Maryville COLLEGE

Makefile – Explicit Recipes

- When we compile multiple files, we need to explicitly tell make how to go about doing it.
- We do this by creating a file named "`Makefile`"
- In the `Makefile` we write a series of **recipes** in the following format:

  ```
  target:  ingredient list
  ```

- For example, try the following:

Makefile – Explicit Recipes

- When we compile multiple files, we need to explicitly tell make how to go about doing it.
- We do this by creating a file named "`Makefile`"
- In the `Makefile` we write a series of **recipes** in the following format:

  ```
  target:  ingredient list
  ```

- For example, try the following:
  1. Change to the `examples/19-Classy` directory.

Maryville COLLEGE

Makefile – Explicit Recipes

- When we compile multiple files, we need to explicitly tell make how to go about doing it.
- We do this by creating a file named "`Makefile`"
- In the `Makefile` we write a series of **recipes** in the following format:

  ```
  target:  ingredient list
  ```

- For example, try the following:
  1. Change to the `examples/19-Classy` directory.
  2. Create a file named `Makefile` with the following content:

```
sodasim: sodasim.o soda-machine.o
	g++ -o sodasim sodasim.o soda-machine.o

sodasim.o: sodasim.cpp soda-machine.h
soda-machine.o: soda-machine.cpp soda-machine.h
```

Maryville COLLEGE

Makefile – Explicit Recipes

- When we compile multiple files, we need to explicitly tell make how to go about doing it.
- We do this by creating a file named "`Makefile`"
- In the `Makefile` we write a series of **recipes** in the following format:

```
target:  ingredient list
```

- For example, try the following:
  1. Change to the `examples/19-Classy` directory.
  2. Create a file named `Makefile` with the following content:

```
sodasim: sodasim.o soda-machine.o
	g++ -o sodasim sodasim.o soda-machine.o

sodasim.o: sodasim.cpp soda-machine.h
soda-machine.o: soda-machine.cpp soda-machine.h
```

- Remember that when indenting, you must use a literal tab character!

Maryville COLLEGE

Some Predefined Variables

- The make syntax is itself a scripting language.

Some Predefined Variables

- The make syntax is itself a scripting language.
- Variables begin with dollar signs $.

Some Predefined Variables

- The make syntax is itself a scripting language.
- Variables begin with dollar signs $.
- There are several pre-defined variables, the two most commonly used ones are:

Some Predefined Variables

- The make syntax is itself a scripting language.
- Variables begin with dollar signs $.
- There are several pre-defined variables, the two most commonly used ones are:
  - `$@` – The name of the target

Some Predefined Variables

- The make syntax is itself a scripting language.
- Variables begin with dollar signs $.
- There are several pre-defined variables, the two most commonly used ones are:
  - `$@` – The name of the target
  - `$^` – The list of all ingredients

Some Predefined Variables

- The make syntax is itself a scripting language.
- Variables begin with dollar signs $.
- There are several pre-defined variables, the two most commonly used ones are:
  - `$@` – The name of the target
  - `$^` – The list of all ingredients
- We could simplify the sodasim `Makefile` like so:

```
sodasim: sodasim.o soda-machine.o
    g++ -o $@ $^

sodasim.o: sodasim.cpp soda-machine.h
soda-machine.o: soda-machine.cpp soda-machine.h
```

Maryville COLLEGE

User Defined Variables

- You can also define your own variables:
  `TARGETS=stock`

User Defined Variables

- You can also define your own variables:
  `TARGETS=stock`
- You refer to your own variables like this:
  `$(TARGETS)`

User Defined Variables

- You can also define your own variables:
  `TARGETS=stock`
- You refer to your own variables like this:
  `$(TARGETS)`
- This allows you to make compact makefiles.

Making The Program 5 Makefile

```
TARGETS=stock

#application builds
all: $(TARGETS)
stock: iofun.o main.o stock.o transaction.o portfolio.o
        g++ -o $@ $^

#object files
iofun.o: iofun.h iofun.cpp
main.o: main.cpp iofun.h stock.h transaction.h portfolio.h
stock.o: stock.h stock.cpp
transction.o: transaction.cpp transaction.h
portfolio.o: portfolio.cpp portfolio.h


#delete all binaries
clean:
        rm -f *.o $(TARGETS)
```

Maryville COLLEGE

Building With Make

- Run `make` to build the first recipe in the `Makefile`

Building With Make

- Run `make` to build the first recipe in the `Makefile`
- Run `make target` to build any other target.

Building With Make

- Run `make` to build the first recipe in the `Makefile`
- Run `make target` to build any other target.
- For example `make clean` runs the clean target.

Lab Activity – Address Book

Using the Program 5 makefile as an example, build a makefile for the address book lab.

```
TARGETS=stock

#application builds
all: $(TARGETS)
stock: iofun.o main.o stock.o transaction.o portfolio.o
        g++ -o $@ $^

#object files
iofun.o: iofun.h iofun.cpp
main.o: main.cpp iofun.h stock.h transaction.h portfolio.h
stock.o: stock.h stock.cpp
transction.o: transaction.cpp transaction.h
portfolio.o: portfolio.cpp portfolio.h


#delete all binaries
clean:
        rm -f *.o $(TARGETS)
```

Maryville COLLEGE