

Makefiles

Nolan Bard

Department of Computing Science
University of Alberta

CMPUT 201, Fall 2005

Overview

- 1 What is a Makefile?
 - Explanation
 - Motivation
- 2 Creating and Using Makefiles
 - Makefile Creation
 - Using make
 - Simple Example
- 3 Try it out!
 - My First Makefile
- 4 Makefile Variables
 - Defining Variables
 - Using Variables

Overview

- 5 Other Uses for make
 - "Phony" Targets
 - Automate This!
- 6 Now Let's Try This!
 - A More Advanced Makefile
- 7 Concluding Notes
 - Conclusion
 - Resources

Part I

What is a Makefile?

Makewhat? Whatfile? Whatwhat?

- Makefiles are text files used by a program called 'make' that help automate tasks
- One common use of make is automating the compilation of programs
- But make can automate a variety of tasks aside from just compiling

Meh! Why bother!

- As programming projects grow in size, compiling the programs often becomes very complicated. An example...

```
$ gcc -I./poker-eval/include -O3 -mcpu=pentiumpro -march=pentiumpro  
-fomit-frame-pointer -ffast-math -static ByteArray.c queue.c  
access.c message.c bot.c util.c read_solution.c hand_strength.c  
util_game_def.c hash.c translate_solution.c make_bucket_game.c  
new_poker.c sequence_form.c tableau.c avl_tree.c  
sequence_form_solution.c -L../game-theory/poker-eval/lib -lpoker  
-lm -o psopti
```

Meh! Why bother!

- Imagine having to type this command every time you wanted to compile this project: and things can get even worse than this...
- Parts of code become interdependent and recompiling everything can take a long time.
- Makefiles save you time in typing complicated commands and allow other people to compile your code with no knowledge of how your code works.

Part II

Creating and Using Makefiles

Making make Makefiles

- First, when creating a makefile, it should generally be called either 'Makefile' or 'makefile' because make automatically looks for files with those names.
- If the file is not called one of these, you must tell make which file to use with 'make -f filename'.
- Within the makefile, you can have any number of rules that follow this format:

```
target ... : dependencies ...  
TAB construction command  
:  
:
```

Making make Makefiles

- First, when creating a makefile, it should generally be called either 'Makefile' or 'makefile' because make automatically looks for files with those names.
- If the file is not called one of these, you must tell make which file to use with 'make -f filename'.
- Within the makefile, you can have any number of rules that follow this format:

```
target ... : dependencies ...  
TAB construction command  
:  
:
```

Making make Makefiles

- Each rule specifies the dependencies needed to run the construction commands that build the target
- IMPORTANT NOTE: The lines of commands must start with a tab
- Keep in mind that dependencies can not only be files but also other targets in your makefile

Making make Makefiles

- Each rule specifies the dependencies needed to run the construction commands that build the target
- **IMPORTANT NOTE:** The lines of commands must start with a tab
- Keep in mind that dependencies can not only be files but also other targets in your makefile

Making make Makefiles

- Each rule specifies the dependencies needed to run the construction commands that build the target
- IMPORTANT NOTE: The lines of commands must start with a tab
- Keep in mind that dependencies can not only be files but also other targets in your makefile

Makefile Basics: Conveniences

```
$ cat Makefile
#This is a comment
#This is a comment that goes on and on and on. But I can make it look \
  nicer by splitting it over multiple lines using a backslash.
```

- Makefiles also allow for comments in the file. Simply use a `#` and everything from it to the end of the line will be ignored by make.
- Sometimes you will have long lines in a makefile. To keep this more legible you can use a backslash to continue the current line on the next line.

Makefile Basics: Conveniences

```
$ cat Makefile
#This is a comment
#This is a comment that goes on and on and on. But I can make it look \
  nicer by splitting it over multiple lines using a backslash.
```

- Makefiles also allow for comments in the file. Simply use a `#` and everything from it to the end of the line will be ignored by make.
- Sometimes you will have long lines in a makefile. To keep this more legible you can use a backslash to continue the current line on the next line.

How to use your Makefile

- Typing 'make target_name' will cause make to check if the target is newer than the dependencies.
- If the target is older, it gets created by executing the target's construction commands.
- Running make without an argument (ie. just type 'make') causes make to run the first rule it encounters in the makefile.
- A good practice for makefiles is for the first rule, usually called 'all', to have dependencies of all the programs you want the makefile to build.
- This prevents other rules from being accidentally invoked when make is run without arguments.

How to use your Makefile

- Typing `'make target_name'` will cause make to check if the target is newer than the dependencies.
- If the target is older, it gets created by executing the target's construction commands.
- Running make without an argument (ie. just type `'make'`) causes make to run the first rule it encounters in the makefile.
- A good practice for makefiles is for the first rule, usually called `'all'`, to have dependencies of all the programs you want the makefile to build.
- This prevents other rules from being accidentally invoked when make is run without arguments.

How to use your Makefile

- Typing 'make target_name' will cause make to check if the target is newer than the dependencies.
- If the target is older, it gets created by executing the target's construction commands.
- Running make without an argument (ie. just type 'make') causes make to run the first rule it encounters in the makefile.
- A good practice for makefiles is for the first rule, usually called 'all', to have dependencies of all the programs you want the makefile to build.
- This prevents other rules from being accidentally invoked when make is run without arguments.

When make Goes Wrong

- make can fail to build a target for a number of reasons. Usually it fails when:
 - there is a dependency it does not know how to create
 - one of the construction commands for a target returns an error
- If make does not have a rule for a dependency, you will have to fix this by adding it to the makefile.
- Different commands will return an error for different reasons.
 - Compilers return an error value if they cannot compile the code.
 - Other programs like 'rm' may return an error if you try to remove a file that does not exist.
- If we want to ignore the return value of a command, then we tell make this by putting a '-' in front of the command (eg. '-rm hello').

When make Goes Wrong

- make can fail to build a target for a number of reasons. Usually it fails when:
 - there is a dependency it does not know how to create
 - one of the construction commands for a target returns an error
- If make does not have a rule for a dependency, you will have to fix this by adding it to the makefile.
- Different commands will return an error for different reasons.
 - Compilers return an error value if they cannot compile the code.
 - Other programs like 'rm' may return an error if you try to remove a file that does not exist.
- If we want to ignore the return value of a command, then we tell make this by putting a '-' in front of the command (eg. '-rm hello').

When make Goes Wrong

- make can fail to build a target for a number of reasons. Usually it fails when:
 - there is a dependency it does not know how to create
 - one of the construction commands for a target returns an error
- If make does not have a rule for a dependency, you will have to fix this by adding it to the makefile.
- Different commands will return an error for different reasons.
 - Compilers return an error value if they cannot compile the code.
 - Other programs like 'rm' may return an error if you try to remove a file that does not exist.
- If we want to ignore the return value of a command, then we tell make this by putting a '-' in front of the command (eg. '-rm hello').

When make Goes Wrong

- make can fail to build a target for a number of reasons. Usually it fails when:
 - there is a dependency it does not know how to create
 - one of the construction commands for a target returns an error
- If make does not have a rule for a dependency, you will have to fix this by adding it to the makefile.
- Different commands will return an error for different reasons.
 - Compilers return an error value if they cannot compile the code.
 - Other programs like 'rm' may return an error if you try to remove a file that does not exist.
- If we want to ignore the return value of a command, then we tell make this by putting a '-' in front of the command (eg. '-rm hello').

Simple Example

- If we wanted to compile code in myProgram.c, we could use the following makefile to do so:

```
$ cat Makefile
#This is a comment
#This is a comment that goes on and on and on. But I can make it look \
  nicer by splitting it over multiple lines using a backslash.

all: myProgram

myProgram: myProgram.c
  gcc -Wall -ansi myProgram.c -o myProgram
```

Part III

Try it out!

Let's Try this out

First, let's make a quick "Hello World" Program in hello.c

```
#include <stdio.h>
int main(int argc, char * argv[])
{
    printf("Hello World\n");
    return 0;
}
```

Let's Try this out

- Now let's make a makefile to compile our project. Open a file called 'Makefile' in the same directory as hello.c
 - We want to compile hello.c into a program called 'hello'
 - 'hello' depends on the source code in hello.c
 - 'gcc -Wall -ansi hello.c -o hello' will make 'hello' from hello.c
- Using this information, write a Makefile that can create 'hello'
- Now add a rule above this that reads "all: hello"
- Now we can build our project by typing 'make hello' (or just 'make')
- Keep what you just did around. We will use it later in the slides.

Let's Try this out

- Now let's make a makefile to compile our project. Open a file called 'Makefile' in the same directory as hello.c
 - We want to compile hello.c into a program called 'hello'
 - 'hello' depends on the source code in hello.c
 - 'gcc -Wall -ansi hello.c -o hello' will make 'hello' from hello.c
- Using this information, write a Makefile that can create 'hello'
- Now add a rule above this that reads "all: hello"
- Now we can build our project by typing 'make hello' (or just 'make')
- Keep what you just did around. We will use it later in the slides.

Let's Try this out

- Now let's make a makefile to compile our project. Open a file called 'Makefile' in the same directory as hello.c
 - We want to compile hello.c into a program called 'hello'
 - 'hello' depends on the source code in hello.c
 - 'gcc -Wall -ansi hello.c -o hello' will make 'hello' from hello.c
- Using this information, write a Makefile that can create 'hello'
- Now add a rule above this that reads "all: hello"
- Now we can build our project by typing 'make hello' (or just 'make')
- Keep what you just did around. We will use it later in the slides.

Part IV

Makefile Variables

Defining Variables

- To define a variable, type 'variable_name = text'
- Variables in makefiles can eliminate a lot of duplication, allowing you to change things in only one place.

```
$ cat Makefile
#This is how we define a variable called 'CC'
CC = gcc

all: myProgram

myProgram: myProgram.o
    gcc -Wall -ansi myProgram.o -o myProgram
```

Using Variables

- To use a variable, type '\$(variable_name)'
- This will substitute the text specified at the variable's definition in place of '\$(variable_name)'.

```
$ cat Makefile  
CC = gcc  
CFLAGS = -Wall -ansi
```

```
all: myProgram
```

```
myProgram: myProgram.o  
    $(CC) $(CFLAGS) myProgram.o -o myProgram
```

Using Variables

- Some common variables to define are:
 - CC - the name of the compiler you are using
 - CFLAGS - the flags/options you pass to the C compiler
 - LIBS - any libraries that you are linking to (eg. '-lm')
 - DEFS - any defines (eg. -DMEMWATCH -DMW_STDIO)
- If you are defining variables, generally they should go at the beginning of the Makefile. This way they are easy to find.
- If you use variables, then changing the variable definition changes the entire makefile! This means less work for you!

Using Variables

- Some common variables to define are:
 - CC - the name of the compiler you are using
 - CFLAGS - the flags/options you pass to the C compiler
 - LIBS - any libraries that you are linking to (eg. '-lm')
 - DEFS - any defines (eg. -DMEMWATCH -DMW_STDIO)
- If you are defining variables, generally they should go at the beginning of the Makefile. This way they are easy to find.
- If you use variables, then changing the variable definition changes the entire makefile! This means less work for you!

Using Variables

- Some common variables to define are:
 - CC - the name of the compiler you are using
 - CFLAGS - the flags/options you pass to the C compiler
 - LIBS - any libraries that you are linking to (eg. '-lm')
 - DEFS - any defines (eg. -DMEMWATCH -DMW_STDIO)
- If you are defining variables, generally they should go at the beginning of the Makefile. This way they are easy to find.
- If you use variables, then changing the variable definition changes the entire makefile! This means less work for you!

Part V

Other Uses for make

Make More of make

- Phony targets are targets that do not actually create a file
- They can be used for a variety of things like:
 - cleaning up temporary files in your project (ie. .o, core, emacs backup files)
 - creating a tarball for your assignment (ie. submit.tar)
 - submitting your assignment
- The idea of a 'make clean' command is widely used and also required for your assignment
- Commands for submitting can be especially useful to create ahead of time so there is less "Ack! I need to submit!" panic when assignments are due.

Make More of make

- Phony targets are targets that do not actually create a file
- They can be used for a variety of things like:
 - cleaning up temporary files in your project (ie. .o, core, emacs backup files)
 - creating a tarball for your assignment (ie. submit.tar)
 - submitting your assignment
- The idea of a 'make clean' command is widely used and also required for your assignment
- Commands for submitting can be especially useful to create ahead of time so there is less "Ack! I need to submit!" panic when assignments are due.

Make More of make

- Phony targets are targets that do not actually create a file
- They can be used for a variety of things like:
 - cleaning up temporary files in your project (ie. .o, core, emacs backup files)
 - creating a tarball for your assignment (ie. submit.tar)
 - submitting your assignment
- The idea of a 'make clean' command is widely used and also required for your assignment
- Commands for submitting can be especially useful to create ahead of time so there is less "Ack! I need to submit!" panic when assignments are due.

Make More of make

- Phony targets are targets that do not actually create a file
- They can be used for a variety of things like:
 - cleaning up temporary files in your project (ie. .o, core, emacs backup files)
 - creating a tarball for your assignment (ie. submit.tar)
 - submitting your assignment
- The idea of a 'make clean' command is widely used and also required for your assignment
- Commands for submitting can be especially useful to create ahead of time so there is less "Ack! I need to submit!" panic when assignments are due.

Make More of make

```
$ cat Makefile
CC = gcc
CFLAGS = -Wall -ansi

TARBALL_NAME = submit.tar

all: myProgram

myProgram: myProgram.o
    $(CC) $(CFLAGS) myProgram.o -o myProgram

$(TARBALL_NAME): Makefile README myProgram.c
    tar -cvf $(TARBALL_NAME) Makefile README myProgram.c

handin: $(TARBALL_NAME)
    astep -c c201 -p assn1 $(TARBALL_NAME)

clean:
    -rm -f core *.o
```


But wait, there's more!

- Using makefiles to generate experimental results can also be very handy.
- Imagine a piece of code that creates a number of files of data that need to be graphed.
- Suppose you use UNIX tools like gnuplot to create graphs and you do it all by hand.
- If data in the files change (say there was a mistake in your code) you have to remake all the graphs by hand.
- Automating things like this in a makefile can greatly reduce the time it takes to analyze results.

But wait, there's more!

- Using makefiles to generate experimental results can also be very handy.
- Imagine a piece of code that creates a number of files of data that need to be graphed.
- Suppose you use UNIX tools like gnuplot to create graphs and you do it all by hand.
- If data in the files change (say there was a mistake in your code) you have to remake all the graphs by hand.
- Automating things like this in a makefile can greatly reduce the time it takes to analyze results.

But wait, there's more!

- Using makefiles to generate experimental results can also be very handy.
- Imagine a piece of code that creates a number of files of data that need to be graphed.
- Suppose you use UNIX tools like gnuplot to create graphs and you do it all by hand.
- If data in the files change (say there was a mistake in your code) you have to remake all the graphs by hand.
- Automating things like this in a makefile can greatly reduce the time it takes to analyze results.

But wait, there's more!

- Using makefiles to generate experimental results can also be very handy.
- Imagine a piece of code that creates a number of files of data that need to be graphed.
- Suppose you use UNIX tools like gnuplot to create graphs and you do it all by hand.
- If data in the files change (say there was a mistake in your code) you have to remake all the graphs by hand.
- Automating things like this in a makefile can greatly reduce the time it takes to analyze results.

But wait, there's more!

- Using makefiles to generate experimental results can also be very handy.
- Imagine a piece of code that creates a number of files of data that need to be graphed.
- Suppose you use UNIX tools like gnuplot to create graphs and you do it all by hand.
- If data in the files change (say there was a mistake in your code) you have to remake all the graphs by hand.
- Automating things like this in a makefile can greatly reduce the time it takes to analyze results.

Part VI

Now Let's Try This!

Let's try this out again...

- Using the makefile and the Hello World program you already made, we will now try out some of the other makefile techniques.
- In your `hello.c` file, add `' #include "memwatch.h" '`
- Open your makefile. We want to:
 - compile `hello.c` with `memwatch.c`
 - remove duplication in our makefile
 - be able to clean up our files
 - be able to submit easily

Let's try this out again...

- Using the makefile and the Hello World program you already made, we will now try out some of the other makefile techniques.
- In your `hello.c` file, add `' #include "memwatch.h" '`
- Open your makefile. We want to:
 - compile `hello.c` with `memwatch.c`
 - remove duplication in our makefile
 - be able to clean up our files
 - be able to submit easily

Let's try this out again...

- Using the makefile and the Hello World program you already made, we will now try out some of the other makefile techniques.
- In your hello.c file, add ' `#include "memwatch.h"` '
- Open your makefile. We want to:
 - compile hello.c with memwatch.c
 - remove duplication in our makefile
 - be able to clean up our files
 - be able to submit easily

Let's try this out again...

- Above "all: hello", define:
 - CC as the compiler command
 - DEFS as a list of define (-Dsomertext) statements
 - CFLAGS as a list of flags/options to pass to the compiler
 - OBJECTS as all the .o files you will need to build your program
- Change the 'hello' rule to use your new variables
- Finally add rules for cleaning up, making a tarball, and submitting the tarball
- Now we can build our project using multiple source files by typing 'make hello' (or just 'make'). And we don't need to worry when it comes to the due date since we can just type a quick make command!

Let's try this out again...

- Above "all: hello", define:
 - CC as the compiler command
 - DEFS as a list of define (-Dsomertext) statements
 - CFLAGS as a list of flags/options to pass to the compiler
 - OBJECTS as all the .o files you will need to build your program
- Change the 'hello' rule to use your new variables
- Finally add rules for cleaning up, making a tarball, and submitting the tarball
- Now we can build our project using multiple source files by typing 'make hello' (or just 'make'). And we don't need to worry when it comes to the due date since we can just type a quick make command!

Let's try this out again...

- Above "all: hello", define:
 - CC as the compiler command
 - DEFS as a list of define (-Dsomertext) statements
 - CFLAGS as a list of flags/options to pass to the compiler
 - OBJECTS as all the .o files you will need to build your program
- Change the 'hello' rule to use your new variables
- Finally add rules for cleaning up, making a tarball, and submitting the tarball
- Now we can build our project using multiple source files by typing 'make hello' (or just 'make'). And we don't need to worry when it comes to the due date since we can just type a quick make command!

Let's try this out again...

- Above "all: hello", define:
 - CC as the compiler command
 - DEFS as a list of define (-Dsomertext) statements
 - CFLAGS as a list of flags/options to pass to the compiler
 - OBJECTS as all the .o files you will need to build your program
- Change the 'hello' rule to use your new variables
- Finally add rules for cleaning up, making a tarball, and submitting the tarball
- Now we can build our project using multiple source files by typing 'make hello' (or just 'make'). And we don't need to worry when it comes to the due date since we can just type a quick make command!

A Final Product

```
CC = gcc
DEFS = -DMEMWATCH -DMW_STDIO
CFLAGS = -Wall -ansi $(DEFS)

OBJECTS = hello.o memwatch.o
BASIC_C = hello.c
BASIC_H =

EXEC_NAME = hello
TARBALL_NAME = submit.tar

all: $(EXEC_NAME)

$(EXEC_NAME): $(OBJECTS)
    $(CC) $(CFLAGS) $(OBJECTS) -o $(EXEC_NAME)
$(TARBALL_NAME): Makefile README $(BASIC_C) $(BASIC_H)
    tar -cvf $(TARBALL_NAME) Makefile README $(BASIC_C) $(BASIC_H)
handin: $(TARBALL_NAME)
    astep -c c201 -p assn1 $(TARBALL_NAME)
clean:
    -rm -f core *.o $(EXEC_NAME)
```

Part VII

Concluding Notes

Conclusion

- make is a very useful tool for automating the compilation of your programs.
- Using just these commands, make can also automate many other tedious tasks
- These features are only the basics. Much more advanced things can be done with makefiles, but this should get you started.

Resources

- Your Linux (Sobool) book has two sections on make!
- Check out page 399 and 715.
- Check out the GNU Make website at:
`http://www.gnu.org/software/make/`
- And the manual at:
`http://www.gnu.org/software/make/manual/make.html`