# Applied Programming

# Make

# Make

- **make**
  - A language used to automate building and testing programs http://www.gnu.org/software/make/
  - We will **use make extensively in the homework**
  - Here is a make tutorial:

    http://www.opussoftware.com/tutorial/TutMakefile.htm

# Make

- A dependency-tracking software build utility:
  - Built into Unix/Linux
  - Widely available on most platforms
    - PC, Mac, Android, etc.
  - Created by Stuart Feldman in 1976

- Allows people to build and install software without knowing anything about the software.
  - **GNU Make** is one of the most popular versions

# Make

- "Automatically*" figures out which files need to be update
  - * By checking file date stamps
- "Automatically" determines the proper order for updating files
  - Only rebuilds the minimum set of files necessary
- Not limited to any particular language
  - Can be use to deinstall software or do anything else you want to do.

- Really a rules based language
  - Based on rules **you** provide

# Makefiles

- Make is a **rule** based language
  - **Not** a **procedural** language
  - No order, does not "read" from the top down.

- Each **rule** begins with a **target** followed by a colon (:) and a list of optional **components** (files or other targets), followed by one or more lines of "commands"

# Rules

- A **rule** tells Make how to execute commands in order to build a target
  - It specifies a target and a list of dependencies
  - This list includes all files (source files and/or other targets) necessary

- A Make **target** is something you want to produce or do
  - E.g.  An obj file, plot file, etc

Note: Make “commands” are just the command line instructions you would **execute by hand** to build the code

# Basics

- "Makefiles" should be called "Makefile"
  - With the a capital M and NO file type!
  - The default and standard most people use
    - BTW: You can run: make –f <name>

- General rule form:

```
target: components ……
    <tab> commands
```

- Note: Commands MUST be indented with a **TAB** character NOT SPACES!!

# Rules

- General form: **target**: **components ……**
  Example:

  **code.exe: code.c include.h other.obj**

  **….**

  **other.obj: other.c include.h**

  – Code.exe depends on any changes in code.c, include.h and other.obj
  – Other.obj depends on other.c and include.h

- If other.c changes, then other.obj would execute which would then cause code.exe to execute, in that order.
  – Rule base language

# Full example

```
code.exe: code.c include.h other.obj
<tab>     @echo "Compiling code"
<tab>     gcc -std=c99 –Wall code.c –O1 –o code.exe
```

- If make determines code.exe must be updated then it will run the command(s) AFTER the rule.
  - These command(s) can do anything, not just compile.
  - Commands are just normal terminal command lines
- If a command generates an error (bad RC) then make STOPS!
  - Make executes rules NOT lines, they will be out of order

# Make command line options

- Start with:
  - `-` Errors are ignored; - gcc file.c –o file
  - `@` Command is not printed to standard output @echo “Compiling code”
  - `+` Command is executed even if Make is in a "do not execute" mode (not used)

- Alternatively, use the commands:
  - .IGNORE - ignore all errors, not a good idea
  - .SILENT - don’t echo everything, a good idea

# Macros & Variables

- Variables are just simple macros with a constant string:
  - Traditionally uses CAPITAL letters
    - Required in this class
  - E.g.: CC = gcc -std=c99 –Wall –O1 –pedantic -g
  - The “variable” CC contains the compiler string

- Using variables and macros
  - Enclose in $(..) then use as the string
  - E.g.: echo $(CC) prints the compiler line

# More Macros

- Macros are only evaluated when used
- Macros can be created from shell commands.
  - DAYDATE = ` date `
  - Note: The back tick (`) NOT a tick (‘)

- Macros can be made from other macros
  - V1 = Good
  - V2 = morning
  - V3 = $(V1) $(V2)
  - echo $(V3) Prints “Good morning”

# More Macros

- Macros can use string substitution
  - "%" matches zero or more characters.

```
SOURCE = one.c    two.c
OBJS   = $(patsubst %.c, %.o, $(SOURCE))
```

- Replace all the ".c file types" with ".o"
  - The macro OBJS holds: **one.o two.o**

# Quick Rules

- We want ALL .c files to compile to .o files

```
.c.o :
    echo "My compile"
    $(CC) $(CFLAGS) –c $(SOURCE)
```

Note: Make has "built in" rules that might be used instead of yours. Always verify YOUR rules are running with a handy echo statement

# Conditionals

- Make supports: If .. Then .. Else
  - Not used much
- The *if-condition* can be:
  - ifdef *variable-name*
  - ifndef *variable-name*
  - The *variable-name* should **not** be surrounded by $()
- or
  - ifeq *test*
  - ifneq *test*
  - the *test* can be expressed as: *"a" "b"* or *(a,b)*

# Standards

- Programmers expect certain standard targets
  - **make all** (or just make),
    - Compiles everything
  - **make clean**
    - Cleans the applications up, gets rid of the executables, object files, plots, temporary files, etc.
  - **make install**
    - Installs applications in the right place
    - Not used in this class.

# Phony Targets

- Targets without prerequisites
  - Really shell scripts in the make file
    - E.g: make clean
  - Normally, phony targets will always be executed, unless the name of a phony target exists as a file
    - E.g : If there was a file called “clean” then make clean would only execute if you changed the clean file!
- To avoid this problem use .PHONY

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

# Special Macros

| Value | Comment |
|---|---|
| $@ | The target name<br>$(CC) $@.cpp -o $@ |
| $? | The name(s) of **all** the changed dependents<br>$(CC) $? -o $@ |
| $< | The name of the related file that caused the action $(CC) -c $< |
| $* | The prefix shared by target and dependent files.<br>$(CC) -c $*.c |
| # | Use the hashtag "#" for comments |
| \ | Continuation character, not normally used |

# Special Macro $@

| Value | Comment |
|---|---|
| $@ | The target name<br>$(CC) $@.cpp -o $@ |

| | Example | Comment |
|---|---|---|
| Bin: | One.c two.c<br>$(CC) $@.cpp -o $@ | $@ contains "Bin"<br>Really:<br>$(CC) Bin.cpp –o Bin |
| Plot: | One.c two.c<br>$(CC) $@.cpp -o $@ | $@ contains "Plot"<br>Really<br>$(CC) Plot.cpp –o Plot |

# Special Macro $<

| Value | Comment |
|---|---|
| $< | The name of the related file that caused the action $(CC) -c $< |

| | Example | Comment |
|---|---|---|
| Bin: | One.c two.c<br>$(CC) $<.cpp -c $< | Assume One.c changed<br>$< contains "One.c" |
| Bin: | One.c two.c<br>$(CC) $<.cpp -c $< | Assume Two.c changed<br>$< contains "Two.c" |
| Bin: | One.c two.c<br>$(CC) $<.cpp -c $< | Assume One.c AND Two.c changed<br>$< contains "One.c" |

# Dependency Chain Example

```
all: $(PROG)

$(PROG): $(OBJS)
        echo "linking $(PROG)"

test: $(PROG)
        echo "Testing GCD Solver"

#This works like: all obj's depend on C's
.c.o:   echo "Compile $<"
```

**All depends on PROG**

**PROG depends on OBJS**

**TEST depends on PROG**

**OBJS depends on C, end of the line**

# Example 1/5

```
# Simple demonstration make file
SOURCE   = gcd_driver.c gcd_module.c
INCLUDE = gcd_module.h
PROG       = gcd
RESULTS = results.txt


# Compiler and Directives
CC           = gcc
CFLAGS = -Wall -std=c99 –O1 -pedantic –g
LFLAGS = -lm
```

Just declaring a bunch of "variables"

# Example 2/5

```
# Build a list of OBJ files from the source
OBJS    = $(patsubst %.c, %.o, $(SOURCE))

# Don't print out each executed line
.SILENT:

# The default is to build the app
all: $(PROG)
```

Make a list of .o files for later

Disable printing executed lines

Our first real rule!

Comment says it all!

# Example 3/5

```
# Execute the program – aka: make test
.PHONY: test
test: $(PROG)
	echo "Testing GCD Solver“
	./$(PROG) 22 > $(RESULTS)
	./$(PROG) 25 >> $(RESULTS)
	echo "The answers are:“
	@cat $(RESULTS)
```

Our 2nd rule Test depends on $(PROG)

<Tabs> not spaces

Does stuff, saves the results in a file, then prints them out

# Example 4/5

```
# Convert each .c file to a .o file
.c.o:  echo "compile $<“
        $(CC) $(CFLAGS) -c $<

# Rebuild the solution if any .o file
$(PROG): $(OBJS)
        echo "linking $(PROG)“
        $(CC) $(CFLAGS) $(LFLAGS) $(OBJS) -o $(PROG)
```

Our 3rd rule tells make what to do with .c files

The 4th rule tells make what to do with .o files

# Example 5/5

```
.PHONY: help
help:
        echo "make options:  all, test, clean, help"


# Remove everything we build
.PHONY: clean
clean:
        -rm -f $(PROG)
        -rm -f *.o
        -rm -f $(RESULTS)
```

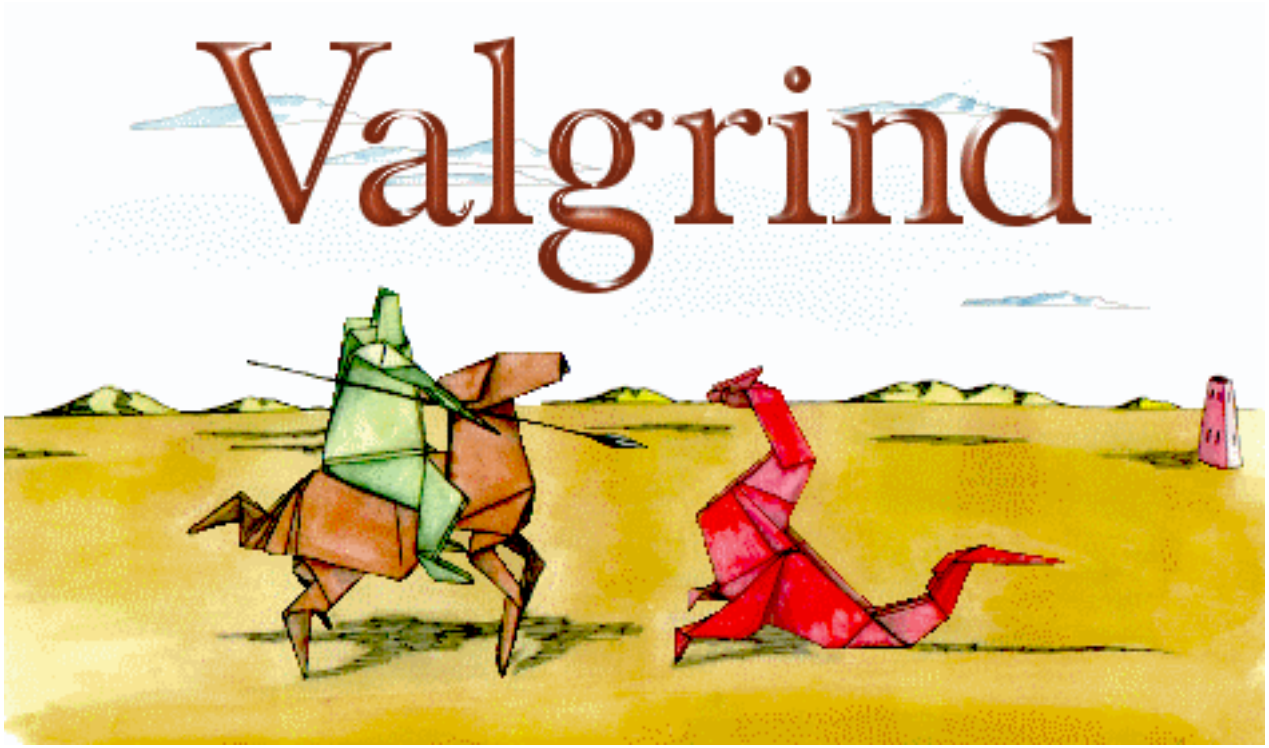
Just being nice

Notice the "-"!

What would happen if rm didn't find a file?

# CMPE 380 Standards

- Our make files will minimally contain
  - **make all**
    - Compiles everything
  - **make clean**
    - Cleans the applications up, gets rid of the executables, object files, plots, temporary files, etc.
  - **make mem**
    - Generates a Valgrind memory report
      - **And depends on the built code!**
  - **make help**
    - List all the key make targets




- A FAMILY of tools to detect *memory management bugs*, *threading bugs* and profile programs in detail.
  - Valgrind.org

# Sample Code Fragment

```
/* Simple program, compiles OK, no errors or warnings */
int main(){
    int *num;
    /* Get memory, more later */
    num = malloc(sizeof(int)*NUM_ELEMS);
    num [0] = 0;
    printf("The first numbers are %d %d\n", num[0], num[1]);

    /* Init the number */
    for (int i = 2; i <= NUM_ELEMS; i++) {
       num[i] = i;
    }
    printf("The next numbers are %d %d\n", num[2], num[3]);
    free(num);
return(0);
}
```

# Valgrind Output

```
Conditional jump or move depends on uninitialized value(s)
   at 0x4E814CE: vfprintf (vfprintf.c:1660)
   by 0x4F43F37: __printf_chk (printf_chk.c:35)
   by 0x40061C: main (in val)
Uninitialized value was created by a heap allocation
  at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
  by 0x4005F7: main (in val)

Use of uninitialized value of size 8
  at 0x4E8099B: _itoa_word (_itoa.c:179)
  by 0x4E84636: vfprintf (vfprintf.c:1660)
  by 0x4F43F37: __printf_chk (printf_chk.c:35)
  by 0x40061C: main (in val)
Uninitialized value was created by a heap allocation
  at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
  by 0x4005F7: main (in val)

Conditional jump or move depends on uninitialized value(s)
  at 0x4E809A5: _itoa_word (_itoa.c:179)   by 0x4E84636: vfprintf
(vfprintf.c:1660)
  by 0x4F43F37: __printf_chk (printf_chk.c:35)
  by 0x40061C: main (in val)
Uninitialized value was created by a heap allocation
  at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
   by 0x4005F7: main (in val)

Conditional jump or move depends on uninitialized value(s)
  at 0x4E84682: vfprintf (vfprintf.c:1660)
  by 0x4F43F37: __printf_chk (printf_chk.c:35)
  by 0x40061C: main (in val)
Uninitialised value was created by a heap allocation
  at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
  by 0x4005F7: main (in val)
```

```
Conditional jump or move depends on uninitialized value(s)
  at 0x4E81599: vfprintf (vfprintf.c:1660)
  by 0x4F43F37: __printf_chk (printf_chk.c:35)
  by 0x40061C: main (in val)
Uninitialized value was created by a heap allocation
  at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
  by 0x4005F7: main (in val)

Conditional jump or move depends on uninitialized value(s)
  at 0x4E8161C: vfprintf (vfprintf.c:1660)
  by 0x4F43F37: __printf_chk (printf_chk.c:35)
  by 0x40061C: main (in val)
Uninitialized value was created by a heap allocation
  at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
  by 0x4005F7: main (in val)

Invalid write of size 4
  at 0x400632: main (in val)
Address 0x5200054 is 0 bytes after a block of size 20 alloc'd
  at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
  by 0x4005F7: main (in val)

HEAP SUMMARY:    in use at exit: 0 bytes in 0 blocks
total heap usage: 1 allocs, 1 frees, 20 bytes allocated

All heap blocks were freed -- no leaks are possible

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 7 errors from 7 contexts
suppressed: 0 from 0)(suppressed: 0 from 0)
```

A lot of errors for a simple program. What does it all Mean?

# Reading Valgrind Output 1

**Conditional jump or move depends on uninitialized value(s)**
  at 0x4E814CE: vfprintf (vfprintf.c:1660)
  by 0x4F43F37: __printf_chk (printf_chk.c:35)
  by 0x40061C: main (in val)
Uninitialised value was created by a heap allocation
  at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
  by 0x4005F7: main (in val)

*This doesn't tell us anything. This is caused by something else. Keep looking!*

**Use of uninitialized value of size 8**
  at 0x4E8099B: _itoa_word (_itoa.c:179)
  by 0x4E84636: vfprintf (vfprintf.c:1660)
  by 0x4F43F37: __printf_chk (printf_chk.c:35)
  by 0x40061C: **main (in val)**
Uninitialised value was created by a heap allocation
  at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
  by 0x4005F7: main (in val)

*Ah! We are using a variable we didn't set (initialize). In Val.c, in function* ***main****, using a printf. Need to look at the source code*

# Valgrind decode 1

- **Use of uninitialized value of size 8**
  - We are reading something 8 bytes long (like a 64 bit integer or double) that was never set.
- **Code fragment**

```
num [0] = 0;
printf("The first numbers are %d %d\n", num[0], num[1]);
```

- **Look in the code for a print statement that uses an uninitialized value 8 bytes long.**

# Reading Valgrind Output 2

Conditional jump or move depends on uninitialized value(s)
  at 0x4E8161C: vfprintf (vfprintf.c:1660)
  by 0x4F43F37: __printf_chk (printf_chk.c:35)
  by 0x40061C: main (in val)
Uninitialised value was created by a heap allocation
  at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
  by 0x4005F7: main (in val)

*This doesn't tell us anything. This is caused by something else. Keep looking!*

**Invalid write of size 4**
  at 0x400632: main (in val)
Address 0x5200054 is 0 bytes after a block of size 20 alloc'd
  at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
  by 0x4005F7: **main (in val)**

*Ah! We are writing a variable we don't own someplace in* ***main*** *in val.c.*

# Valgrind decode 2

- **Invalid write of size 4**
  - We are writing something 4 bytes long (like an integer) into memory we don't own.
- **Code fragment**

```
/* Init the number */
  for (int i = 2; i <= NUM_ELEMS; i++) {
    num[i] = i;
  }
```

- **Look in the code for spot where you are setting a variable, typically in a loop.**

# Valgrind decode 3

HEAP SUMMARY: in use at exit: **0 bytes in 0 blocks**
total heap usage: 1 allocs, 1 frees, 20 bytes allocated

All heap blocks were freed **-- no leaks are possible**

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 7 errors from 7 contexts
suppressed: 0 from 0)(suppressed: 0 from 0)

- Even after you fix all the memory access errors, you might still have "leaks".
  - In C the programmer has to manually manage memory usage.
  - A leak is a memory management error

# Valgrind Hints

- Look for:
  - Use of uninitialized value
  - Invalid write of size
  - Initially ignore the rest of the warnings
- The number of bytes and the function name give you a hint where to look
  - Fix one at a time and then recompile
- More on memory management later

# HW Hint 1 - gcc

- Typical command:

**gcc –std=c99 –Wall –pedantic-O1 –g -lm [files.c] –o binFile**

| | |
|---|---|
| -std=c99 | - "using modern C" |
| -Wall | - all warnings |
| -pedantic | - more warning |
| -O1 | - simple optimization, more warning |
| -g | - generate debug information |
| -lm | - include (-l) lib (m) math |
| files.c | - one or more .C files |
| -o binFile | - the name of the output binary is binFile |

- Example:

**gcc –stc=c99 -O1 –Wall –pedantic –g -lm QuadraticSolver.c –o qs**

# HW Hint 2 - Valgrind

- A FAMILY of tools to detect *memory management bugs*, *threading bugs* and profile programs in detail.
  - More in a future class

- Valgrind Sample:

**valgrind --tool=memcheck --leak-check=yes --track-origins=yes ./bin [options]**

- **Where: ./bin** - binary to test
  **options** - any parameters or options for the binary

You always need to make sure that there are no memory leaks AND no memory access errors!

*Important:* You must *compile with* the –g *option* (in gcc)

# HW Hint 3 – memory leaks

- Find memory access and leaks using Valgrind:

**valgrind --tool=memcheck --leak-check=yes --track-origins=yes ./qs 1 -1 1**

- Sample output:

==27024== HEAP SUMMARY:

==27024== in use at exit: **0 bytes in 0 blocks**

==27024== total heap usage: 0 allocs, 0 frees, 0 bytes allocated

==27024== All heap blocks were freed -- **no leaks are possible**

- Goal: No leaks for any execution path in the code.
- Results can CHANGE for different execution paths

# HW Hint 4 – bugs

- Finding other programming bugs:

**valgrind --tool=memcheck --leak-check=yes --track-origins=yes ./val**

- Sample output:

==27024== HEAP SUMMARY:

==27024== in use at exit: **0 bytes in 0 blocks**

==27024== total heap usage: 1 allocs, 1 frees, 20 bytes allocated

==27024== All heap blocks were freed -- **no leaks are possible**

- Goal: No leaks for any execution path in the code.
- Results can CHANGE for different execution paths

# Example 1

- Is there anything wrong with the following make rule?

```
all :
```

- Most likely yes, there are no component dependencies, this is normally wrong. A better solution is something like:

```
all: $(prog)
```

# Example 2

- Which is better makefile programming style?

```
all: myProg
```

Or

```
PROG = myProg
all: $(PROG)
```

- Both makefile rules are functionally identical but the 2nd will be more maintainable over time

# Example 3

- What does this really say? (-o is output)

```
PROG = myProg
$(PROG): $(PROG)_A.c
$(CC)  $<     -o  $@
```

- The myProg binary depends on myProg_A.c. If myProg_A.c is newer than myProg, compile using the file that changed $< (myProg_A.c) and produce the binary $@ (myProg)

# Example 4

- Given: $(PROG): $(PROG)_A.c
  Which is better?

```
$(CC)  $<     -o  $@
```

Or

```
$(CC)  $(PROG)_A.c    -o  $(PROG)
```

- Both makefile rules are functionally identical but the 1st will be more maintainable over time

# Required Reading – next class

- Download from MyCourses

  C for Java Programmers – Maassen.pdf