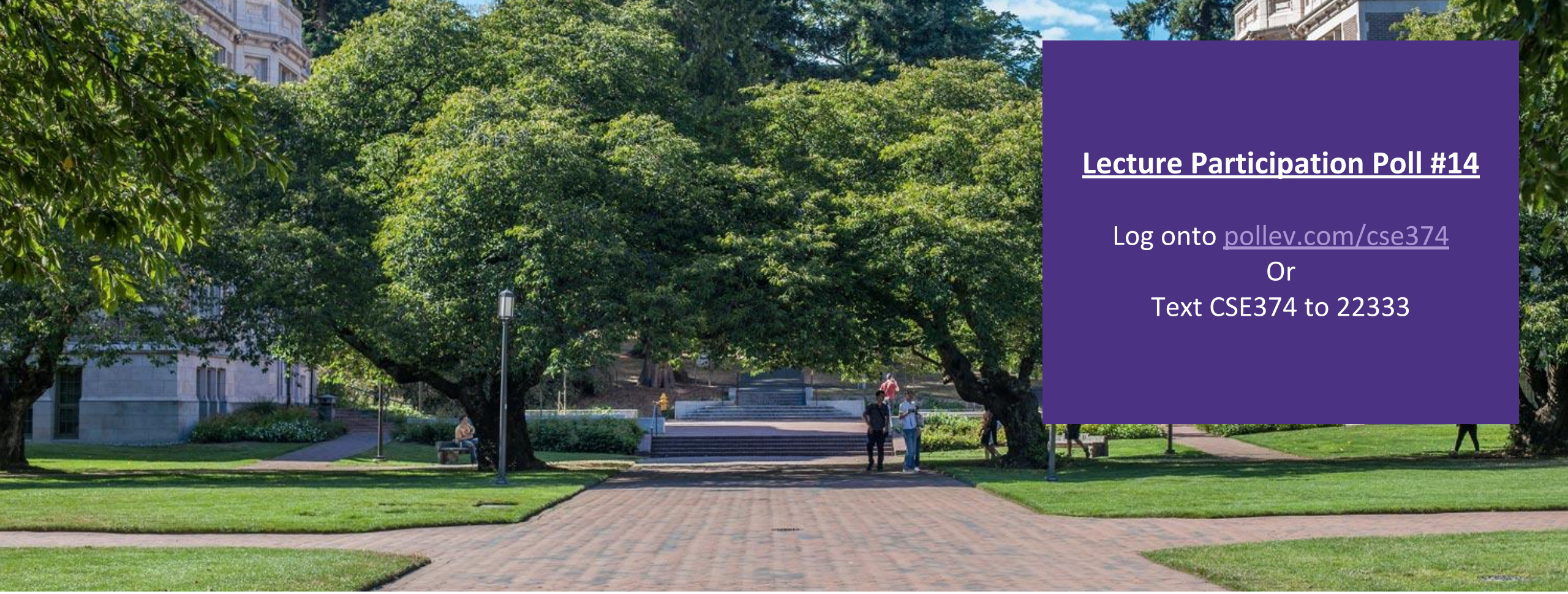


Lecture Participation Poll #14

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

Lecture 14: makefiles

CSE 374: Intermediate Programming Concepts and Tools

Administrivia

Assignments

HW2 Live - Soft Deadline **EXTENDED** Friday October 30th at 9pm PST

- Autograder updated

Reminder: Midpoint Deadline Friday November 6th at 9pm PST

Review Assignment Live – Due Date **EXTENDED** Thursday October 29th at 9pm PST

- 24 hrs late 20% penalty
- 48 hrs late 50% penalty
- Not accepted more than 48hrs late
- Combination of autograding & handchecking

Reminder: Midpoint Deadline Friday November 6th at 9pm PST

- Will post grades to canvas sometime the week after

Make Files

- **Make** is a program which automates building **dependency trees**
  - List of **rules** written in a **Makefile** declares the commands which build each intermediate part
  - Helps you avoid manually typing gcc commands, easier and less prone to typos
  - Automates build process
- Makefiles are a list of with **Make rules** which include:
  - **Target** - An output file to be generated, dependent on one or more sources
  - **Source** – Input source code to be built
  - **Recipe** - command to generate target

tab not spaces! →

```
target: source
    recipe
```

```
ll.o: ll.c ll.h
    gcc -c ll.c
```

- Makefile logic
  - Make builds based on structural organization of how code depends on other code as defined by includes
  - Recursive – if a source is also a target for other sources, must also evaluate its dependencies and remake as required
  - Make can check when you've last edited each file, and only build what is needed!
    - Files have "last modification date". make can check whether the sources are more recent than the target
  - Make isn't language specific: recipe can be any valid shell command
- run `make` command from within same folder
  - `$make [ -f makefile] [ options ] … [ targets ] ../`
  - Starts with first rule in file then follows dependency tree
  - -f specifies makefile name, if non provided will default to "Makefile"
  - if no target is specified will default to first listed in file

https://www.gnu.org/software/make/manual/make.html#Introduction

Makefile Example: Linked List

```c
#include "ll.h"

int main() {
    Node *n1 = make_node(4, NULL);

    // rest of main…
```

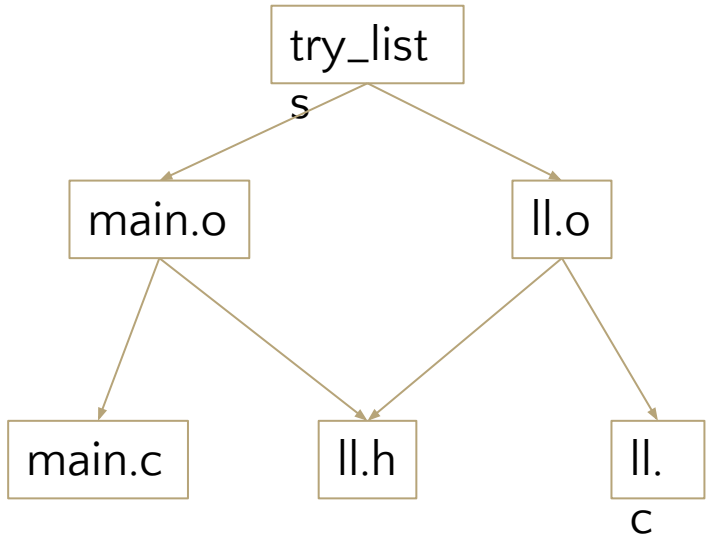
main.c

```c
#ifndef LL_H
#define LL_H

typedef struct Node {

    // rest of Node def…
```

ll.h

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

#include "ll.h"

Node *make_node(int value, Node *next) {
    // rest of linked list code…
```

ll.c




```
try_lists: ll.o main.o
   gcc -o try_lists ll.o main.o

ll.o: ll.c ll.h
    gcc -c ll.c -o ll.o

main.o: main.c ll.h
   gcc -c main.c -o main.o
```

Makefile

More Make Tools

- make variables help reduce repetitive typing and make alterations easier
  - can change variables from command line
  - enables us to reuse Makefiles on new projects
  - can use conditionals to choose variable settings
- ifdef checks if a given variable is defined for conditional execution
  - ifndef checks if a given variable is NOT defined
- Special characters:
  - $@ for target
  - $^ for all sources
  - $< for left-most source
  - \ enables multiline recipies
  - * functions as wildcard (use carefully)
  - % enables implicit rule definition by using % as a make specific wildcard

```
CC = gcc
CGLAGS = -Wall

foo.o: foo.c foo.h bar.h
    $(CC) $(CFLAGS) -c foo.c -o foo.o

make CFLAGS=-g

EXE=
ifdef WINDIR #defined on Windows
   EXE=.exe
endif
widget$(EXE): foo.o bar.o
    $(CC) $(CFLAGS) -o widget$(EXE)\
        foo.o bar.o

OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
    gcc -o widget $(OBJFILES)

%.o: %.c
    $(CC) -c $(CFLAGS) $< -o $@
clean:
   rm *.o widget
```

Makefile

http://cslibrary.stanford.edu/107/UnixProgrammingTools.pd

Phony Targets

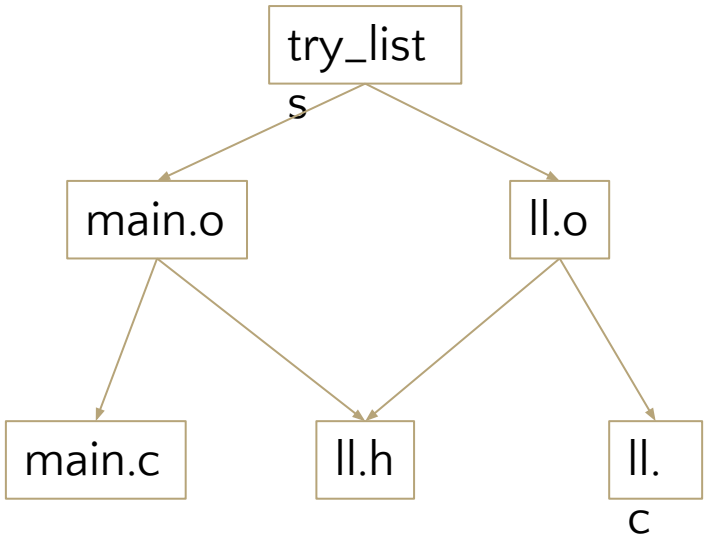
- A target that doesn't create the listed output
- A way to force run commands regardless of dependency tree
- Common uses:
  - all – used to list all top notes across multiple dependency trees
  - clean – cleans up files after usage
  - test – specifies test functionality
  - printing messages or info

```
all: try_lists test_suite
clean:
    rm objectfiles
test: test_suite
    ./test_suite
```

```
CC = gcc
CGLAGS = -Wall

all: my_program your_program

my_program: foo.o bar.o
    $(CC) $(CFLAGS) -o my_program foo.o bar.o

your_program: bar.o baz.o
    $(CC) $(CFLAGS) -o your_program foo.o baz.o

#not shown: foo.o, bar.o, baz.o targets

clean:
    rm *.o my_program your_program
```

Makefile

Example Makefile



variable definitions

must include rules for each file

rules define dependency hierarchy

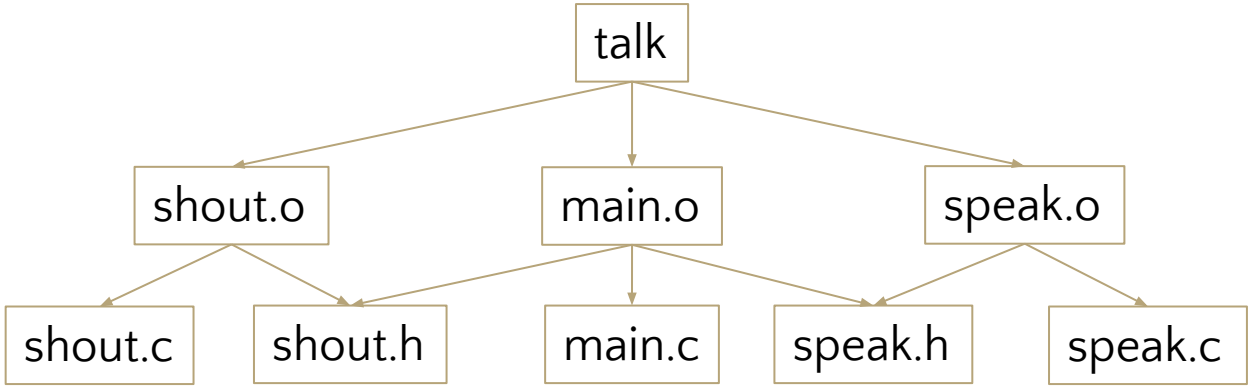
```
CC = gcc
CGLAGS = -g -Wall -std=c11

try_lists: main.o ll.o
    $(CC) $(CFLAGS) -o try_lists main.o ll.o

main.o: main.c ll.h
    $(CC) $(CFLAGS) -c main.c

ll.o: ll.c ll.h
    $(CC) $(CFILES) -c ll.c

clean:
    rm *.o
```

Makefile

Example

```c
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "speak.h"
#include "shout.h"
/* Write message m in uppercase to
stdout */
void shout(char m[])
{
   int len; /* message length */
   char *mcopy; /* copy of original
message */
   int i;
   len = strlen(m);
   mcopy = (char
*)malloc(len*sizeof(char)+1);
   strcpy(mcopy,m);
   for (i = 0; i < len; i++)
      mcopy[i] = toupper(mcopy[i]);
   speak(mcopy); free(mcopy);
}
```

shout.c




```c
#include <stdio.h>
#include "speak.h"
/* Write message m to stdout */
void speak(char m[])
{
   printf("%s\n", m);
}
```

speak.c

```c
#include "speak.h"
#include "shout.h"
/* Say HELLO and goodbye */
int main(int argc, char* argv[])
{
   shout("hello");
   speak("goodbye");
   return 0;
}
```

main.c

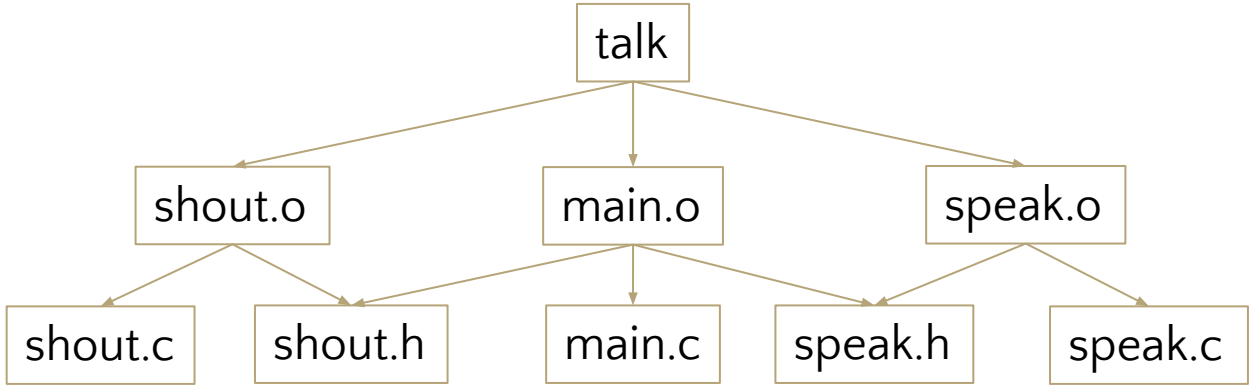
```c
#ifndef SPEAK_H
#define SPEAK_H
/* Write message m to stdout */
void speak(char m[]);
#endif /* ifndef SPEAK_H */
```

speak.h

```c
#ifndef SHOUT_H
#define SHOUT_H
/* Write message m in uppercase to stdout */
void shout(char m[]);
#endif /* ifndef SHOUT_H */
```

shout.h

Example



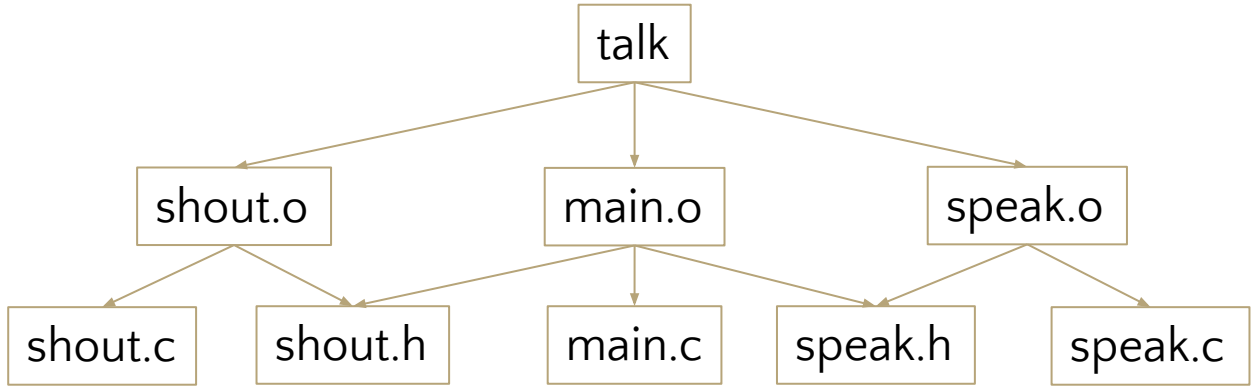

```
all: talk
# The executable
talk: main.o speak.o shout.o
   gcc -Wall -std=c11 -g -o talk main.o speak.o shout.o

# Individual source files
speak.o: speak.c speak.h
   gcc -Wall -std=c11 -g -c speak.c
shout.o: shout.c shout.h speak.h
   gcc -Wall -std=c11 -g -c shout.c
main.o: main.c speak.h shout.h
   gcc -Wall -std=c11 -g -c main.c

# A "phony" target to remove built files and backups
clean: rm -f *.o talk *~
```

Makefile

Example

Makefile

```
CC = gcc
# Compiler flags: -Wall for debugger warnings
# -std=c11 for updated standards
CFLAGS = -Wall -std=c11

ifdef DEBUG
CFLAGS += -g
endif

# The name of the program that we are producing.
TARGET = talk

# This is a "phony" target that tells
# make what other targets to build.
all: $(TARGET)

# All the .o files we need for our executable.
OBJS = main.o speak.o shout.o

# The executable
$(TARGET): $(OBJS)
   $(CC) $(CFLAGS) -o talk $(OBJS)
```




```
# Individual source files
speak.o: speak.c speak.h
   $(CC) $(CFLAGS) -c speak.c
shout.o: shout.c shout.h speak.h
   $(CC) $(CFLAGS) -c shout.c
main.o: main.c speak.h shout.h
   $(CC) $(CFLAGS) -c main.c

# A "phony" target to remove built files and backups
clean: rm -f *.o talk *~
```

Memory Leak

- A memory leak occurs when code fails to deallocate dynamically allocated memory that is no longer used
  - forgetting to free a malloc—ed block
  - losing or changing the pointer to a malloc-ed block
- Result – program's memory will keep growing – your OS handles this
  - ok for short-lived programs because memory is deallocated when program ends
  - bad repercussions for long-lived programs
    - slow down processing over time
    - could exhaust all available memory -> program crash
    - other programs could get starved for memory

Common Memory Errors

```
x = (int*)malloc(M*sizeof(int));
free(x);
y = (int*)malloc(M*sizeof(int));
free(x);
```

Double free and Forgetting to free memory "memory leak"

```
int x[] = {1, 2, 3};
free(x);
```

x is a local variable stored in stack, cannot be freed

```
char** strings = (char**)malloc(sizeof(char)*5);
free(strings);
```

Mismatch of type - wrong allocation size

```
x = (int*)malloc(M*sizeof(int));
free(x);
y = (int*)malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i];
```

Accessing freed memory

Common Memory Errors

```
#define LEN 8
int arr[LEN];
for (int i = 0; i <= LEN; i++)
   arr[i] = 0;
```

Out of bounds access

```
int* foo()
{
   int val = 0;
   return &val;
}
```

dangling pointer

```
long val;
printf("%d", &val);
```

Dereferencing a non-pointer

```
int foo()
{
   int* arr = (int*)malloc(sizeof(int)*N);
   read_n_ints(N, arr);
   int sum = 0;
   for (int i = 0; i < N; i++)
      sum += arr[i];
   return sum;
}
```

memory leak – failing to free memory

```
int sum_int(int* arr, int len)
{
   int sum;
   for (int i = 0; i < len; i++)
      sum += arr[i];
   return sum;
}
```

Reading memory before allocation

Finding and Fixing Memory Errors

- Valgrind is a tool that simulates your program to find memory errors
  - it can detect all of the errors we've discussed so far!
  - catches pointer errors during execution
  - prints summary of heap usage, including details of memory leaks

```
gcc -o myprogram myprogram.c
valgrind --leak-check=full myprogram arg1 ag
```

- Can show:
  - Use of uninitialized memory
  - Reading/writing memory after it has been free'd
  - Reading/writing off the end of malloc'd blocks
  - Reading/writing inappropriate areas on the stack
  - Memory leaks -- where pointers to malloc'd blocks are lost forever
  - Mismatched use of malloc/new/new [] vs free/delete/delete []
  - Overlapping src and dst pointers in memcpy() and related functions

Valgrind Example

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

int main(int argc, char** argv){
  int i;
  int *a = malloc(sizeof(int) * 10);
  if (!a) return -1; /*malloc failed*/
  for (i = 0; i < 11; i++){
    a[i] = i;
  }
  free(a);
  return 0;
}
```

example1.c

```
$ gcc -Wall -pedantic -g example1.c -o example
$ valgrind ./example
==23779== Memcheck, a memory error detector
==23779== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==23779== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==23779== Command: ./example
==23779==
==23779== Invalid write of size 4
==23779==    at 0x400548: main (example1.c:9)
==23779==  Address 0x4c30068 is 0 bytes after a block of size 40 alloc'd
==23779==    at 0x4A05E46: malloc (vg_replace_malloc.c:195)
==23779==    by 0x40051C: main (example1.c:6)
==23779==
==23779==
==23779== HEAP SUMMARY:
==23779==     in use at exit: 0 bytes in 0 blocks
==23779==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==23779==
==23779== All heap blocks were freed -- no leaks are possible
==23779==
==23779== For counts of detected and suppressed errors, rerun with: -v
==23779== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

terminal

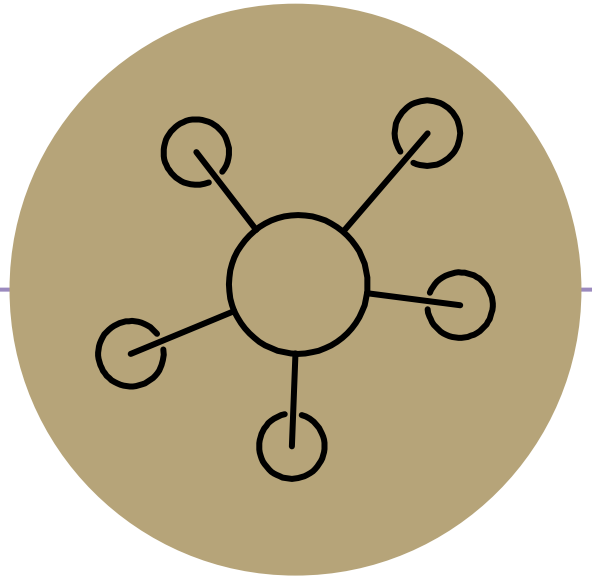
Valgrind EX2

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

int main(int argc, char** argv){
  int i;
  int a[10];
  for (i = 0; i < 9; i++)
    a[i] = i;

  for (i = 0; i < 10; i++){
    printf("%d ", a[i]);
  }
  printf("\n");
  return 0;
}
```

example2.c

```
$ gcc -Wall -pedantic -g example2.c -o example2
$ valgrind ./example2
==24599== Memcheck, a memory error detector
==24599== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==24599== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==24599== Command: ./example2
==24599==
==24599== Conditional jump or move depends on uninitialised value(s)
==24599==    at 0x33A8648196: vfprintf (in /lib64/libc-2.13.so)
==24599==    by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
==24599==    by 0x400567: main (example2.c:11)
==24599==
==24599== Use of uninitialised value of size 8
==24599==    at 0x33A864484B: _itoa_word (in /lib64/libc-2.13.so)
==24599==    by 0x33A8646D50: vfprintf (in /lib64/libc-2.13.so)
==24599==    by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
==24599==    by 0x400567: main (example2.c:11)
==24599==
==24599== Conditional jump or move depends on uninitialised value(s)
==24599==    at 0x33A8644855: _itoa_word (in /lib64/libc-2.13.so)
==24599==    by 0x33A8646D50: vfprintf (in /lib64/libc-2.13.so)
==24599==    by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
==24599==    by 0x400567: main (example2.c:11)
==24599==
0 1 2 3 4 5 6 7 8 7
==24599==
==24599== HEAP SUMMARY:
==24599==     in use at exit: 0 bytes in 0 blocks
==24599==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==24599==
==24599== All heap blocks were freed -- no leaks are possible
==24599==
==24599== For counts of detected and suppressed errors, rerun with: -v
==24599== Use --track-origins=yes to see where uninitialised values come from
==24599== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 6 from 6)
```

terminal



Appendix

File IO – working with strings

- **FILE *fopen(const char *path, const char *mode);**
  - opens the file whose name is the string pointed to by path and associates a stream with it.
- **char *fgets(char *s, int size, FILE *stream);**
  - reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.
- **int fprintf(FILE *stream, const char *format, ...);**
  - It's printf, but to a file.
  - **int fputc(int** c, **FILE** *stream); *// print a single character*
  - **int fputs(const char** *s, **FILE** *stream); *// print a string*