COMSSA

UNIX & C PROGRAMMING

REVISION SESSION 1

2025 Semester 1

Acknowledgement of Country

We at ComSSA would like to acknowledge the traditional owners of the land on which Curtin University Perth is located, the Whadjuk people of the Nyungar Nation. We pay our respects to Nyungar elders, past, present and emerging, and acknowledge their wisdom and guidance in our teaching environments.

Today's Agenda

01 Basics

02 Environments

03 Pointers

04 Function Pointers

05 Arrays & Strings

Icebreaker

In groups, compare C and the last language you learned. Pick out differences & things you like about the languages.





C Programming Basics

- Data Types & Declarations
- Control Structures
- C89 Specifics

Data Types & How to Declare Them

INTEGER & LONG

Whole Numbers

```
int myNum = 2;
int otherNum = -25;
long num = 32;
```

FLOAT & DOUBLE

Decimal Numbers

```
float y = 2.6;
double z = 3.14;
```

CHAR

A Single ASCII Character

```
char letter = 'A';
char number = '7';
char symbol = '+';
```

FOR THOSE COMING FROM JAVA

No Inherent Boolean!

However using #define in a header file, we can effectively have the same thing

```
#define FALSE 0
#define TRUE !FALSE /* Anything other than 0 */
```

Control Structures

IF-ELSE

```
if (Boolean condition) {
    statements;
} else if (Boolean condition) {
    statements;
} else {
    statements;
}
```

WHILE AND DO-WHILE LOOPS

```
while (Boolean condition) {
    statements;
}

do { /* always executes at least once */
    statements;
} while (Boolean condition);
```

SWITCH-CASE

```
switch (expression) {
  case literal1:
    statements;
    break;
  case literal2:
    statements;
    break;
  default:
    statements;
    break;
}
```

FOR LOOPS

```
for (initialisation; condition; action) {
    statements;
}
```

Note! C89 requires you to initialise the counter *Outside* the for loop

C89 Specifics

-ansi -pedantic

These enforce C89 standards for your code. Must use these for your assignments.

-Wall -Werror

Displays all compilation warnings and errors.

-Werror treats warnings as errors so they must be resolved

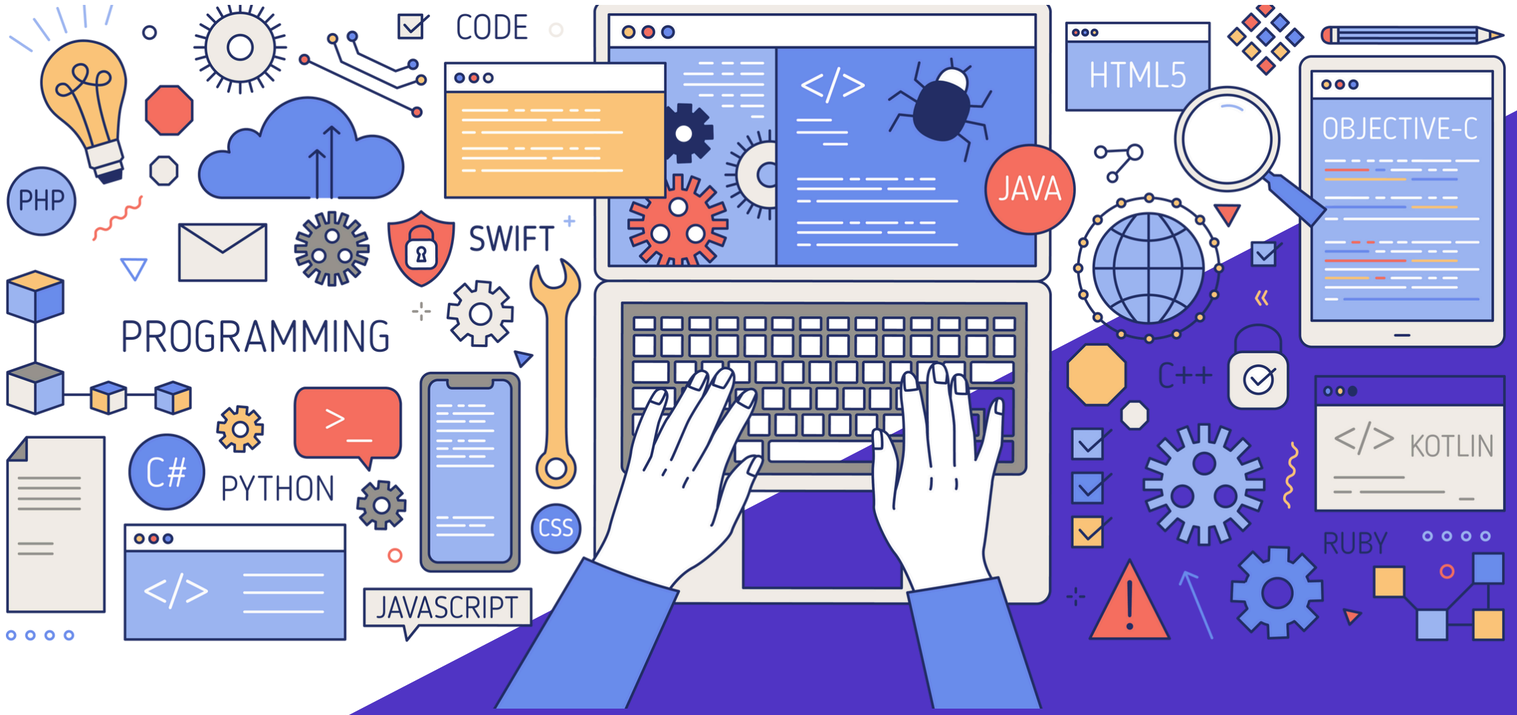
Additional Rules for Code Quality

- No-no's
  - Global Variables
  - break outside of switch statements
  - goto (ever)
  - continue (ever)

- Only a single return statement in non-void functions.

- Functions themselves should be no longer than 50 lines

- No single line should exceed 80 characters



Environments

Header Files

- Used to organise your external declarations for functions used in other files
- also for declaring macros with #define
- use <> for built-in functions (<stdio.h>), “” for your own functions (“one.h”)

```
#define CONDITIONAL 1
...
#ifdef CONDITIONAL
conditionallyRunThisCode();
#endif
codeThatRunsRegardless();
```

```
gcc -c somefile.c -D CONDITIONAL = 1
```

ONE.C

```
#include “one.h”

int func1( int n ) { return n++; }
double funct2( int n ) { return n*n; }
```

ONE.H (HEADER FILE)

```
int func1( int n );
double funct2( int n );
```

MAIN.C

```
#include “one.h”
...
func1( 3 );
```

Makefile

Compile every time with one command!

WITHOUT MAKEFILE

```
$ gcc -c -g -ansi -Wall -Werror -pedantic foo.c
$ gcc -c -g -ansi -Wall -Werror -pedantic one.c
$ gcc -c -g -ansi -Wall -Werror -pedantic two.c
$ gcc -c -g -ansi -Wall -Werror -pedantic three.c
$ gcc foo.o one.o two.o three.o -o program
```

Have to do each time you recompile or find the commands for only the edited files and re-link

WITH MAKEFILE

```
$ make
```

```
CC = gcc
CFLAGS = -ansi -pedantic -Wall -Werror -g
OBJ = foo.o one.o two.o three.o
EXEC = program

$(EXEC): $(OBJ)
	$(CC) $(OBJ) -o $(EXEC)

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

one.o : one.c one.h two.h three.h
	$(CC) -c -g $(CFLAGS) one.c

two.o : two.c two.h macros.h
	$(CC) -c -g $(CFLAGS) two.c

three.o : three.c three.h macros.h
	$(CC) -c -g $(CFLAGS) three.c

clean :
	rm -f $(OBJ) $(EXEC)
```

Question Time

Which of the following is true?

01 Reference a variable with $variable

02 You need to include all header files in every file

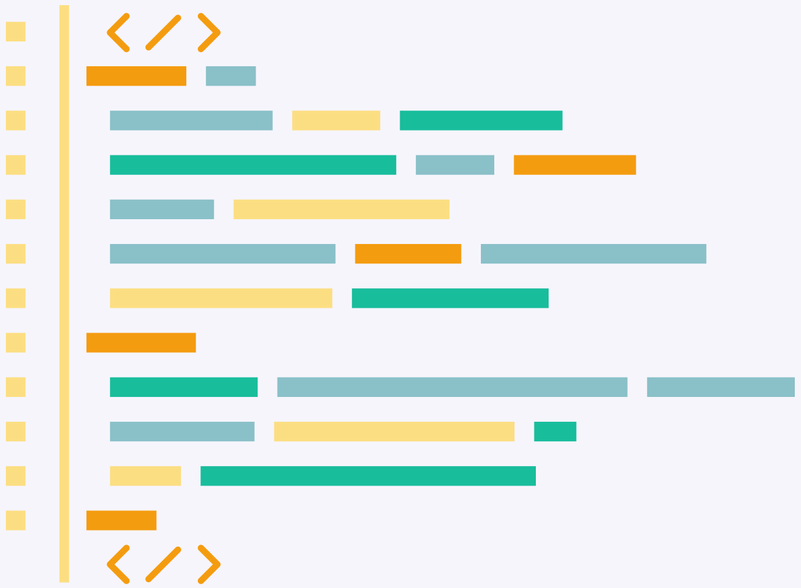
03 A makefile only helps you compile files

04 You only need to include relevant header files

05 A header file can't exist without a corresponding .c file

Question Time

Which of the following is true?



01 Reference a variable with $variable
($(variable))

02 You need to include all header files in every file
(Some header files aren't relevant in the c files.)
(Remember, header files reference things from other files)

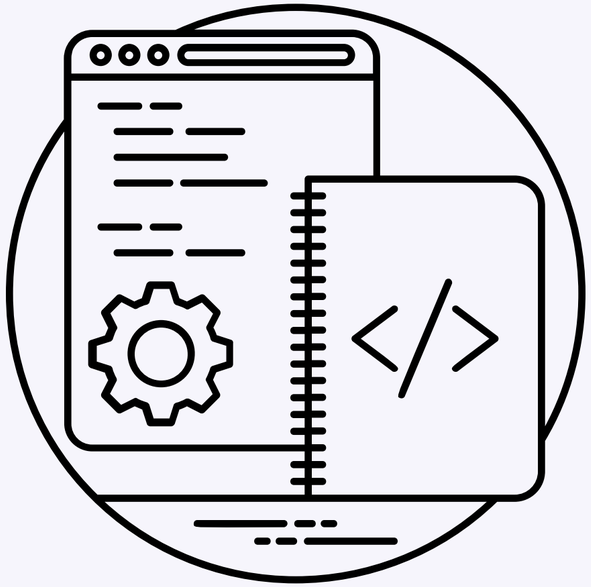
03 A makefile only helps you compile files
(It can do other things, like remove files too)

04 You only need to include relevant header files

05 A header file can't exist without a corresponding .c file
(Macros.h is a file where there are no corresponding c files and it still works)

Question Time

Which of the following is false?



01 When making a makefile, you must use hard tabs instead of spaces

02 You get an object file after compiling a c file

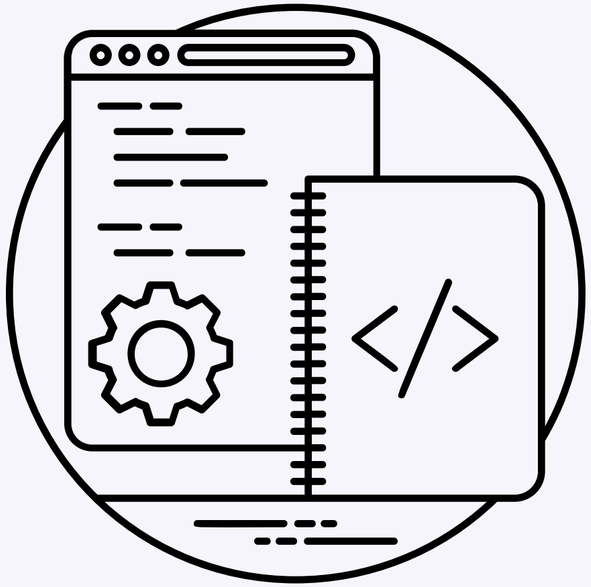
03 Constants can be stored in the header file

04 Header files are important to reference methods

05 You need to recompile every file each time you make a change

Question Time

Which of the following is false?



01 When making a makefile, you must use hard tabs instead of spaces

02 You get an object file after compiling a c file

03 Constants can be stored in the header file

04 Header files are important to reference methods

05 You need to recompile every file each time you make a change
(You only need to recompile the files you've made changes to)

Makefile Activity

In groups of 2 - 3, write a makefile to compile these .c files

MAIN.C

```
#include <stdio.h>
#include "setup.h"

void main(int argc, char** argv)
{/* code that does cool stuff here */ }
```

GAME.C

```
#include <stdio.h>
#include "game.h"
#include "map.h"
#include "macros.h"

void startGame(/* parameters */)
{ /* cool game code */ }
```

SETUP.C

```
#include <stdio.h>
#include "setup.h"
#include "game.h"
#include "map.h"
#include "macros.h"

void extract(/* parameters */)
{/* code that does some stuff here */ }
```

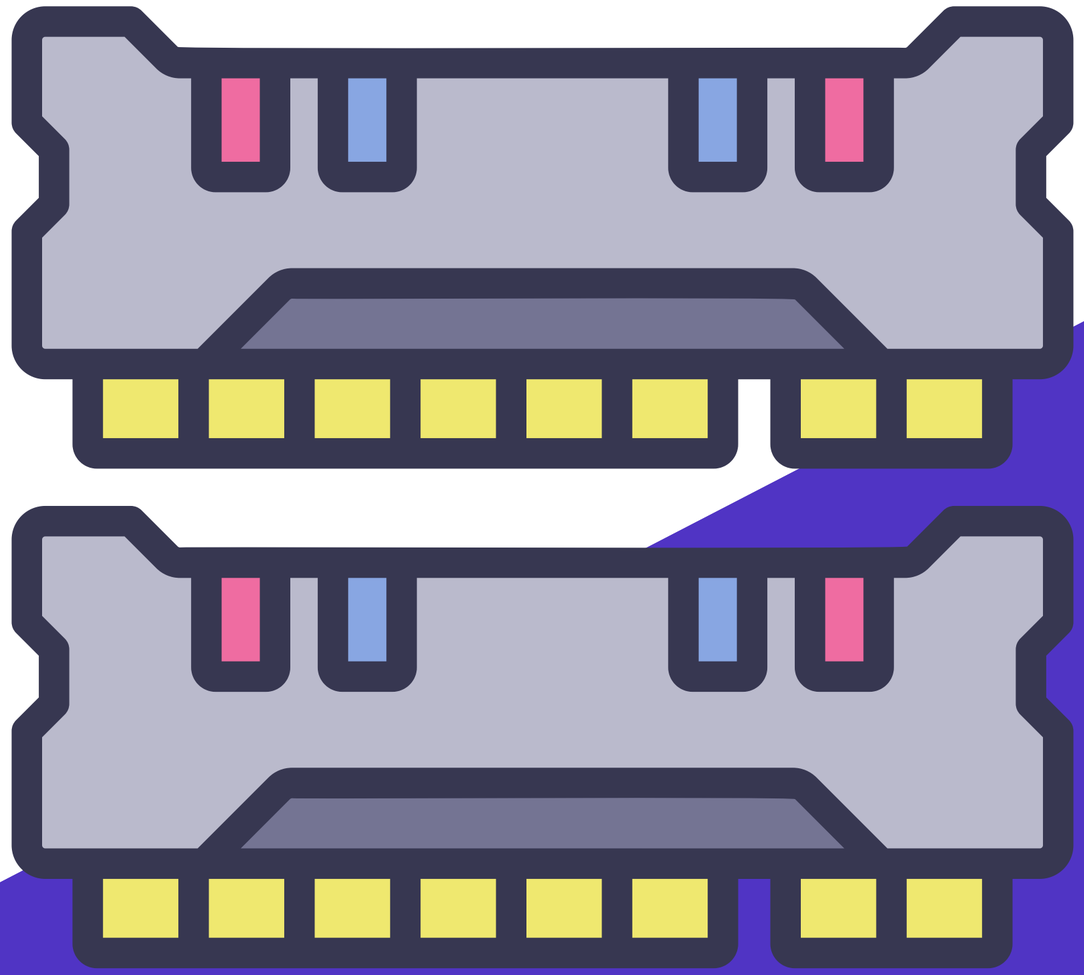
MAP.C

```
#include <stdio.h>
#include "map.h"
#include "macros.h"

void printMap(/* parameters */)
{/* code that does map stuff here */ }
```

EXTRA CHALLENGE

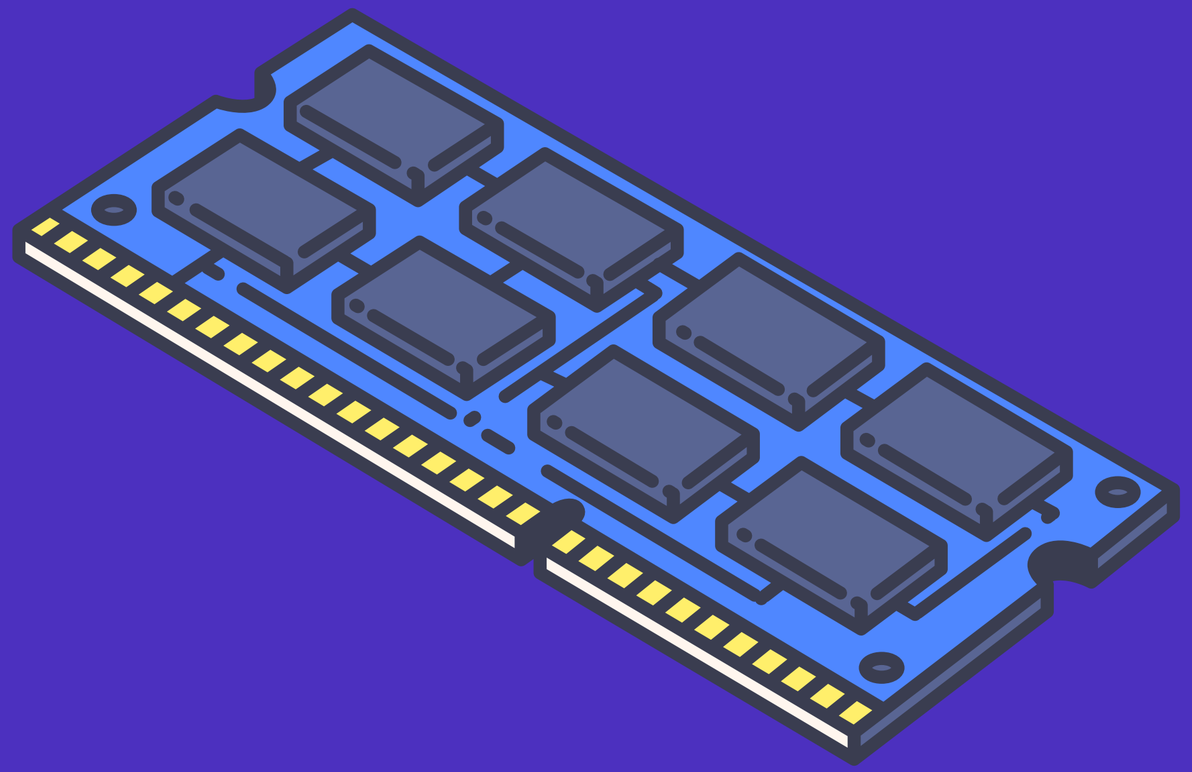
Add in conditional compilation for a BORDERLESS flag



Pointers

Pointers

- Pointers are special data types
- They store memory addresses ***not values***
- Declare pointers using the '*' operator
- Cannot assign a value to a pointer.
  - Can assign the memory address of the value or null
- To access the ***value*** the memory address is pointing to, use '*'
  - Do not try and access a pointer containing null (your program will crash)



0x7ffe5367e044

Pointer Basics

MEMORY ADDRESSES

```
int num = 64;
printf(" %p", &num);
```

Print the memory address of '*num*' using the '&' operator

Your paragraph text

ACCESSING VALUES

```
char letter = 'b';
char* letterPtr = &letter;
printf(" %c", *letterPtr);
```

Declare a pointer, assign it the memory address of *'letter*'.
When printing, access the value 'letterPtr' is pointing to using the '*' operator.

DECLARING & ASSIGNING POINTERS

```
float* fltPtr = null;
float flt = 3.14;

fltPtr = &flt;
```

Declare a pointer using '*datatype * var_name*' assign it null then, assign it the memory address containing 'flt'

Discuss in Groups

What is the output of the following?

1

```
int* num1, num2;
num2 = 5;
num1 = &num2;

printf("%d", *num1);
```

2

```
char* letterPtr, letter;
letter = 'z';
letterPtr = &letter;

letter = 'h';

printf("%c ", letter);
printf("%c", *letterPtr);
```

3

```
char* letterPtr = null;
char** ptrPtr = &letterPtr;

printf("%p", *ptrPtr);
printf("%c", *letterPtr);
```

Discuss in Groups

What is the output of the following?

1

```
int* num1, num2;
num2 = 5;
num1 = &num2;

printf("%d", *num1);
```

Output: '5'

2

```
char *letterPtr, letter;
letter = 'z';
letterPtr = &letter;

letter = 'h';

printf("%c ", letter);
printf("%c", *letterPtr);
```

Output: 'h h'

3

```
char* letterPtr = null;
char** ptrPtr = &letterPtr;

printf("%p", *ptrPtr);
printf("%c", *letterPtr);
```

Output: 'Segmentation Fault'

Activity!

Write a method in C that calculates:

- The quantity of pizza required for events
- The total cost of ordering the pizza

Its parameters are:

- A pointer for the total number of pizzas.
- A pointer for the total cost of pizza.
- The number of attendees

Assume:

- Each attendee eats 3 slices of pizza
- Each pizza has 8 slices
- Each pizza is 10 dollars

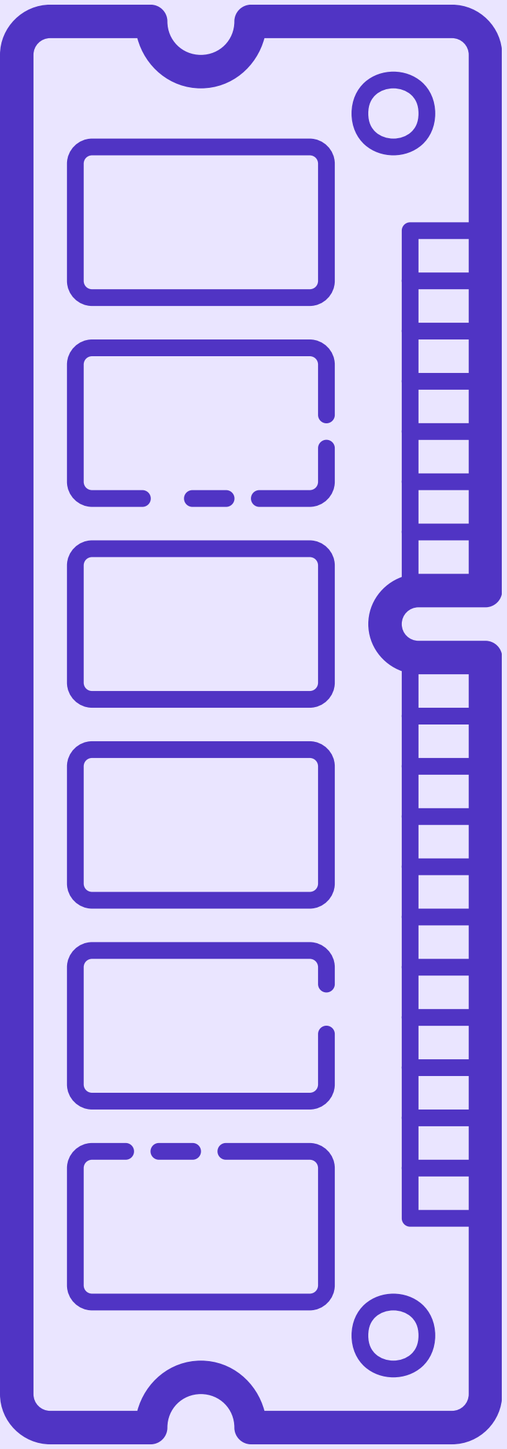
Write the function pointer tags for these two groups of functions

double add(double numOne, double numTwo)
double subtract(double numOne, double numTwo)
double multiply(double numOne, double numTwo)
double divide(double numOne, double numTwo)

int moveUp(char** map, int* pv, int* mv)
int moveDown(char** map, int* pv, int* mv)
int moveLeft(char** map, int* pv, int* mv)
int moveRight(char** map, int* pv, int* mv)

BONUS: Write a function that returns one of these function pointers

Arrays & Strings



Arrays

Static Vs Dynamic

- Both arrays & pointers refer to 'addresses'.
- int* and int[10] are not interchangeable!

- Fixed arrays are allocated locally (on the stack)
  - **Like this:** int foo[3];
  - When you exit the function where they were declared, they stop existing!
- Dynamic “arrays” are allocated on the heap
  - **Like this:** int *foo = (int*)malloc(10 * sizeof (int));
  - Persistent until freed

Dynamic Arrays

- malloc() manually allocates space on the heap
- This space must be manually freed! If there is no free or the program terminates early it causes a memory leak.

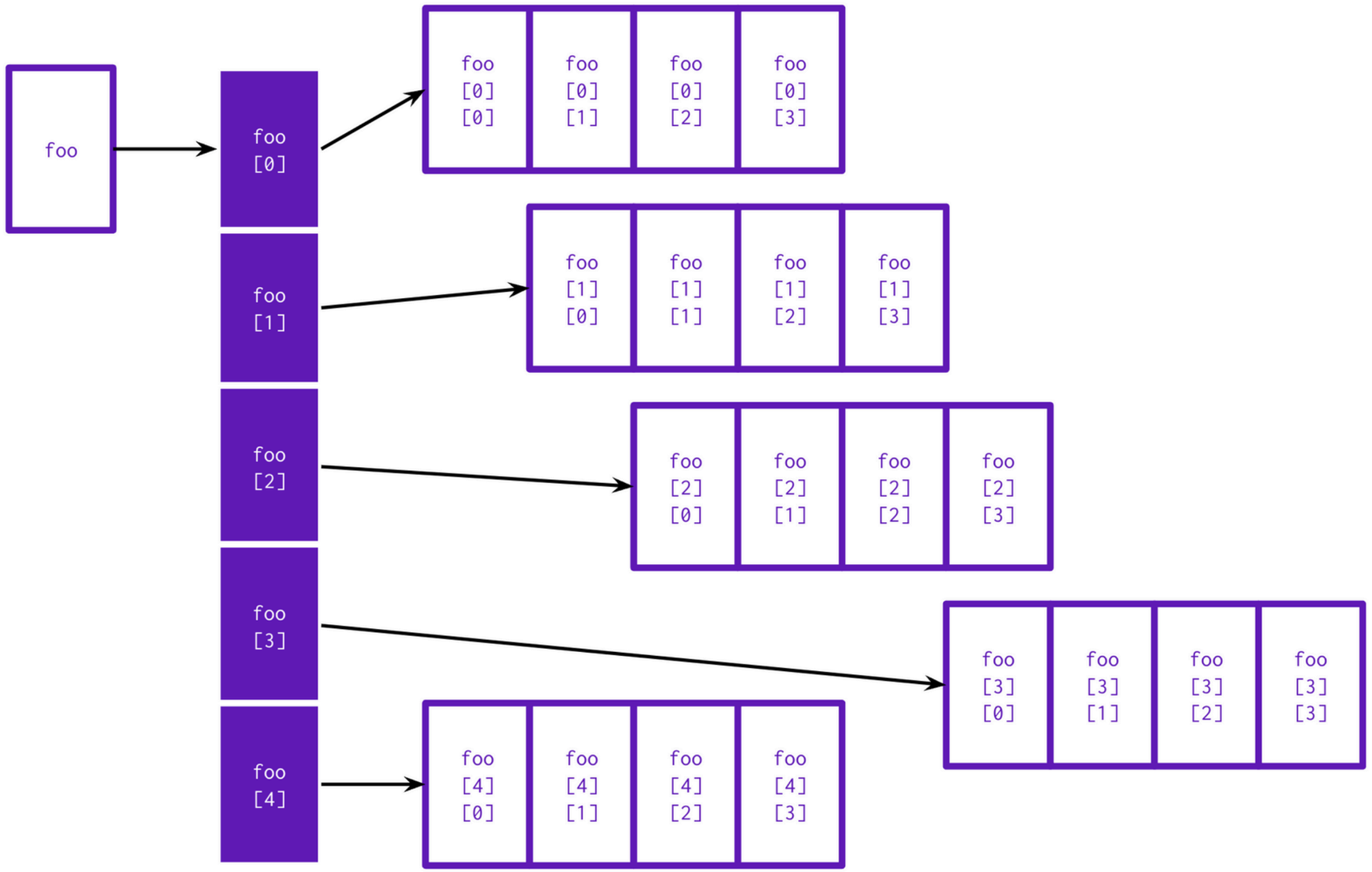
Static Arrays

- Can use sizeof()
- Cannot be too large
- Cannot be multidimensional
- Will not persist outside of the scope it was declared in
- Will not cause memory leaks or errors (unless you manage to REALLY mess up)

Dynamic Arrays

- Declared with malloc()
- Prone to memory leaks & issues if the programmer isn't careful
- Can be huge & multidimensional
- Do not use up space on the stack
- Can be dynamically sized
- Are persistent outside of declared scope
- Must always free()

Dynamic “arrays” of multiple dimensions are essentially pointers to other pointers (a tree of pointers)



Using malloc() for multi-dimensional arrays

```
int i, j;
int*** cubeOfIntegers = (int ***)malloc( 20 * sizeof (int**) );

for (i = 0; i < 20; i++) {
        cubeOfIntegers[i] = (int**)malloc( 20 * sizeof(int*) );
        for (j = 0; j < 20; j++) {
                cubeOfIntegers[i][j] = (int*)malloc( 20 * sizeof(int) );
        }
}
```

**NOTE: The multidimensional arrays must be freed using this method as well! Otherwise you'll have memory leaks & inaccessible memory after your program ends!!

Valgrind

- **USE IT** whenever you are working with pointers and malloc().
  - *valgrind ./example_program*

- Without Valgrind, you'll only know about memory leaks when your PC starts running out of usable memory
- Valgrind also gives more detail about allocation and deallocation errors
- Valgrind can also tell you where the leak/error occurs.
  - *valgrind ./example_program --leak-check=full*
  - *Compile with the -g flag to see line numbers*

- Marks are deducted from assignments for memory leaks & (de)allocation issues!

What do these outputs mean?

```
==3485== Memcheck, a memory error detector
==3485== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3485== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3485== Command: ./valtest
==3485==
==3485==
==3485== HEAP SUMMARY:
==3485==     in use at exit: 10 bytes in 1 blocks
==3485==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==3485==
==3485== LEAK SUMMARY:
==3485==    definitely lost: 10 bytes in 1 blocks
==3485==    indirectly lost: 0 bytes in 0 blocks
==3485==      possibly lost: 0 bytes in 0 blocks
==3485==    still reachable: 0 bytes in 0 blocks
==3485==         suppressed: 0 bytes in 0 blocks
==3485== Rerun with --leak-check=full to see details of leaked memory
==3485==
==3485== For lists of detected and suppressed errors, rerun with: -s
==3485== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

What do these outputs mean?

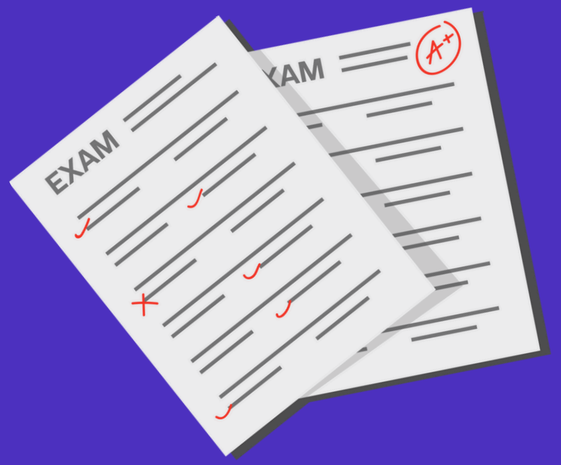
```
==3912== Memcheck, a memory error detector
==3912== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3912== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3912== Command: ./valtest
==3912==
==3912== Invalid free() / delete / delete[] / realloc()
==3912==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==3912==    by 0x109168: main (in /home/nick/Documents/UCP/ForFun/valtest)
==3912==  Address 0x5 is not stack'd, malloc'd or (recently) free'd
==3912==
==3912==
==3912== HEAP SUMMARY:
==3912==     in use at exit: 0 bytes in 0 blocks
==3912==   total heap usage: 0 allocs, 1 frees, 0 bytes allocated
==3912==
==3912== All heap blocks were freed -- no leaks are possible
==3912==
==3912== For lists of detected and suppressed errors, rerun with: -s
==3912== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

What do these outputs mean?

```
==3237== Memcheck, a memory error detector
==3237== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3237== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3237== Command: ./valtest
==3237==
==3237== Invalid write of size 4
==3237==    at 0x1091AB: main (in /home/nick/Documents/UCP/ForFun/valtest)
==3237==  Address 0x4a530a4 is 4 bytes inside an unallocated block of size 4,194,112 in arena "client"
==3237==
==3237== Invalid read of size 4
==3237==    at 0x1091B9: main (in /home/nick/Documents/UCP/ForFun/valtest)
==3237==  Address 0x4a530b8 is 24 bytes inside an unallocated block of size 4,194,112 in arena "client"
==3237==
0
==3237==
==3237== HEAP SUMMARY:
==3237==     in use at exit: 0 bytes in 0 blocks
==3237==   total heap usage: 2 allocs, 2 frees, 1,044 bytes allocated
==3237==
==3237== All heap blocks were freed -- no leaks are possible
==3237==
==3237== For lists of detected and suppressed errors, rerun with: -s
==3237== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

What do these outputs mean?

```
==3639== Memcheck, a memory error detector
==3639== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3639== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3639== Command: ./valtest
==3639==
==3639== Conditional jump or move depends on uninitialised value(s)
==3639==    at 0x109159: main (in /home/nick/Documents/UCP/ForFun/valtest)
==3639==
==3639==
==3639== HEAP SUMMARY:
==3639==     in use at exit: 0 bytes in 0 blocks
==3639==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==3639==
==3639== All heap blocks were freed -- no leaks are possible
==3639==
==3639== Use --track-origins=yes to see where uninitialised values come from
==3639== For lists of detected and suppressed errors, rerun with: -s
==3639== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```



Question / Exam Help

Upcoming Revision Sessions

DSA Revision Session

April 10th

ISE Revision Session

April 11th

Join the Discord!

Check out UniPASS!

Weekly UCP Sessions

Tuesday, 2:00pm - 3:00pm, 407.402

Wednesday, 1:00pm - 2:00pm, 108.117