# CPSC 313 C Refresher

2023W2

# Contents

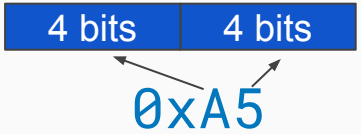
- Basics (number representation, data types, bit shifting, type casting)
- Memory (structs, alignment, endianness)
- Pointers (segfaults, dangling ptrs, mem leaks, stack vs heap memory)
- C programs (arguments, debugging)

# Part 1:
# Basics (number representation, data types, bit shifting, type casting)

# Bytes, Bits and Hexits

Remember your conversions: decimal <-> binary <-> hexadecimal

- ex. $(101)_2$ = 2^2 + 0 + 2^0 = $(5)_{10}$ (binary <-> decimal)
- 1 “hexit” = 1 hex digit = 4 bits, (ex. `0xF` = `0b1111`)
- 1 byte = 8 bits, so 1 byte = 2 hexits



# Addressing Bytes

- Each memory address (ex. `0xDEADBEEF`) houses 1 byte of data
  - note: the memory address is not one byte in size, the value housed in each memory address is 1 byte in size

Q: What is the size of the memory address `0xDEADBEEF`?

# Addressing Bytes

- Each memory address (ex. `0xDEADBEEF`) houses 1 byte of data
  - note: the memory address is not one byte in size, the value housed in each memory address is 1 byte in size

Q: What is the size of the memory address `0xDEADBEEF`?

A: 4 bytes

# Data Types

*Try typing "long long long i;" into gcc*

In this course, assume a **64 bit** CPU (since y86) if unspecified

- so assume for now: `char` is 1 byte, `short` is 2 bytes
- TIP: can always use `sizeof()` to double check sizes
- If you `#include <stdint.h>` you may see `uint32_t` explicitly defines an unsigned 32 bit (4 byte) integer
  - similar for `int32_t`, `uint8_t`, etc.

Q: How many bytes is a `uint64_t`?

# Data Types

*Try typing "long long long i;" into gcc*

In this course, assume a **64 bit** CPU (since y86) if unspecified

- so assume for now: `char` is 1 byte, `short` is 2 bytes
- TIP: can always use `sizeof()` to double check sizes
- If you `#include <stdint.h>` you may see `uint32_t` explicitly defines an unsigned 32 bit (4 byte) integer
  - similar for `int32_t`, `uint8_t`, etc.

Q: How many bytes is a `uint64_t`?

A: 8 bytes

# Signed vs Unsigned

- **What is the difference between signed and unsigned binary numbers?**
- Recall: use two's complement representation for signed binary numbers (flip all bits, add 1)
- Helpful tip: if we know the number is signed the MSB (most significant bit) is the sign bit

# Ranges we can represent with n bits

Given n bits, the range of ints we can represent is:

Unsigned : $[0, 2^n - 1]$
Signed : $[-2^{n-1}, 2^{n-1} -1]$

**Why?** (Hint: how many permutations of n bits are there?)

So if we have an 32 bit integer, plug in for respective ranges :
**unsigned int**: 0 to 4294967295,  **int**: −2,147,483,648 to 2,147,483,647

# Bit Shifting

<<: left shift operator
>>: right shift operator

```
a = a << 2
b = b >> 2
```

```
a <<= 2
b >>= 2
```

```
char c =    0b00000001;
c <<= 6; // 0b01000000
c >>= 7; // 0b00000000
```

**Edge cases**

```
char c =    0b10000000;
c >>= 7; // 0b00000001 or 0b11111111
```

Specified, Implementation-dependent

```
char c = 0b00000001;
c <<= 8 or above; // ?
```

Not well defined!

# Type Casting (Integer types)

**Truncation** (casting to smaller type)

```
int16_t i = 0x000A;
char c = (char) i;  // 0x0A

i = 0xFFFF;
c = (char) i;       // 0xFF
```

**Signed extension** (casting to larger type)

```
char c = 0x0A;
int16_t i = (int16_t) c; // 0x000A

c = 0xFF;
i = (int16_t) c;         // 0xFFFF
```

# Bit Masking

Q: How do you get the value of bit number 3 and bit number 5?

```
char c = 0b11011011; // note you can also define in hex using 0x…
char mask = ???
```

# Bit Masking

Q: How do you get the value of bit number 3 and bit number 5?

```
char c = 0b11011011; // note you can also define in hex using 0x...
char mask = (0b01 << 3) | (0b01 << 5);
char result = ???
```

# Bit Masking

Q: How do you get the value of bit number 3 and bit number 5?

```
char c = 0b11011011; // note you can also define in hex using 0x...
char mask = (0b01 << 3) | (0b01 << 5);
char result = c & mask; // 0b00001000
```

# Strings

- In C, we represent a string using a null terminated char array
  - This means the end of a string is marked by ‘\0’ or 0x0
- String functions are defined in <string.h>
  - strlen counts length of null terminated string (excluding null char)
  - memset fills memory with the same value (e.g. useful for zeroing data)
  - strcpy/strncpy/memcpy/memmove copies data
  - strcmp/strncmp/memcmp compares data

Q: What would printf("%s", str) output?

```
char str[] = “hello, world!”;
str[5] = ‘\0’;
```

# Strings

- In C, we represent a string using a null terminated char array
  - This means the end of a string is marked by ‘\0’ or 0x0
- String functions are defined in <string.h>
  - strlen counts length of null terminated string (excluding null char)
  - memset fills memory with the same value (e.g. useful for zeroing data)
  - strcpy/strncpy/memcpy/memmove copies data
  - strcmp/strncmp/memcmp compares data

Q: What would printf("%s", str) output?

```
char str[] = “hello, world!”;
str[5] = ‘\0’;
```

A: hello

# Part 2:
# Memory (structs, alignment, endianness)

# Structs

```
struct myStruct {
    char a;     // 1 byte
    int32_t i; // 4 bytes
    char b[3]; // 3 bytes
};
```

Initialize a struct called ms

```
struct myStruct ms;
```

Set the "a" field to 0x123

```
ms.a = 0x123;
```

Write a function that takes a struct

```
void func(struct myStruct p) { ...
```

Q: What is the size of `ms` in memory? We will find out...

# Alignment Rules

*Structs in C may be larger than they appear*

1. Whole struct must be aligned by the **alignment of its largest field**

   Ex. if the largest thing in the struct is 8 bytes, the size of the entire struct must be a multiple of 8. (8, 16, 24, 32...)

   We say "the **alignment of the struct** is 8" or "the struct is **8-aligned**"

2. Arrays are aligned by the size of their elements
   e.g. a 100000-long array of int32_t (4 bytes) must start at an address that is a multiple of 4. It is 4-aligned.

3. Anything not a struct or an array is aligned by its raw size
   e.g. a short (2 bytes) must start at an address that is a multiple of 2. It is 2-aligned.

If any rules are not satisfied, add padding (empty space) to the struct until they are.

# The answer

```
struct myStruct {
    char a;    // 1 byte
    int32_t i; // 4 bytes
    char b[3]; // 3 bytes
};
```

# The answer

```
struct myStruct {
    char a;    // 1 byte
    int32_t i; // 4 bytes
    char b[3]; // 3 bytes
};
```

```
 0:   a
 1:   padding
 2:   padding
 3:   padding    (Rule 2: sizeof(i) = 4, so i must be 4-aligned)
 4:   i
 5:   i
 6:   i
 7:   i
 8:   b[0]
 9:   b[1]
10:   b[2]
11:   padding    (Rule 1: sizeof(s) must be a multiple of 4)
```

# The answer

```
struct myStruct {
    char a;    // 1 byte
    int32_t i; // 4 bytes
    char b[3]; // 3 bytes
};
```

A: This struct takes 12 bytes of memory! (not 8)

```
 0:   a
 1:   padding
 2:   padding
 3:   padding    (Rule 2: sizeof(i) = 4, so i must be 4-aligned)
 4:   i
 5:   i
 6:   i
 7:   i
 8:   b[0]
 9:   b[1]
10:   b[2]
11:   padding    (Rule 1: sizeof(s) must be a multiple of 4)
```

# Why do we need alignment?

https://stackoverflow.com/questions/38875369/what-is-data-alignment-why-and-when-should-i-be-worried-when-typecasting-pointe

Taken from the above link (note: this is a simplification)

For example I have two different pieces data to store: data1: "ab" data2: "cdef"

So without alignment the memory will look like : |a b c d| |e f 0 0|

**How many read cycles to read data1? How about data 2?**

Instead, if we use alignment : |a b 0 0 | | c d e f |. **How many cycles for data1? And how many for data 2?**

# Endianness

The **endianness** of an architecture is the way it represents data.
Big: Most significant byte on the lowest address. Makes more sense to humans.
Little (default in CPSC 313): Most significant digit on the highest address. More commonly seen.

**0x1000:** 01 23 45 67 89 AB CD EF
**0x1008:** 03 69 BE 25 8C F1 47 AD

| | Big-endian | Little-endian |
|---|---|---|
| What is the 1-byte integer at location 0x100A? | | |
| What is the 4-byte integer at location 0x1004? | | |

# Endianness

The **endianness** of an architecture is the way it represents data.
Big: Most significant digit on the lowest address. Makes more sense to humans.
Little (default in CPSC 313): Most significant digit on the highest address. More commonly seen.

**0x1000:** 01 23 45 67 89 AB CD EF
**0x1008:** 03 69 BE 25 8C F1 47 AD

| | Big-endian | Little-endian |
|---|---|---|
| What is the 1-byte integer at location 0x100A? | | |
| What is the 4-byte integer at location 0x1004? | | |

# Endianness

The **endianness** of an architecture is the way it represents data.
Big: Most significant digit on the lowest address. Makes more sense to humans.
Little (default in CPSC 313): Most significant digit on the highest address. More commonly seen.

**0x1000:** 01 23 45 67 89 AB CD EF
**0x1008:** 03 69 BE 25 8C F1 47 AD

| | Big-endian | Little-endian |
|---|---|---|
| What is the 1-byte integer at location 0x100A? | 0xBE | |
| What is the 4-byte integer at location 0x1004? | | |

# Endianness

The **endianness** of an architecture is the way it represents data.
Big: Most significant digit on the lowest address. Makes more sense to humans.
Little (default in CPSC 313): Most significant digit on the highest address. More commonly seen.

**0x1000:** 01 23 45 67 89 AB CD EF
**0x1008:** 03 69 BE 25 8C F1 47 AD

| | Big-endian | Little-endian |
|---|---|---|
| What is the 1-byte integer at location 0x100A? | 0xBE | 0xBE |
| What is the 4-byte integer at location 0x1004? | | |

# Endianness

The **endianness** of an architecture is the way it represents data.
Big: Most significant digit on the lowest address. Makes more sense to humans.
Little (default in CPSC 313): Most significant digit on the highest address. More commonly seen.

**0x1000:** 01 23 45 67 89 AB CD EF
**0x1008:** 03 69 BE 25 8C F1 47 AD

| | Big-endian | Little-endian |
|---|---|---|
| What is the 1-byte integer at location 0x100A? | 0xBE | 0xBE |
| What is the 4-byte integer at location 0x1004? | | |

# Endianness

The **endianness** of an architecture is the way it represents data.
Big: Most significant digit on the lowest address. Makes more sense to humans.
Little (default in CPSC 313): Most significant digit on the highest address. More commonly seen.

**0x1000:** 01 23 45 67 89 AB CD EF
**0x1008:** 03 69 BE 25 8C F1 47 AD

| | Big-endian | Little-endian |
|---|---|---|
| What is the 1-byte integer at location 0x100A? | 0xBE | 0xBE |
| What is the 4-byte integer at location 0x1004? | 0x89ABCDEF | |

# Endianness

The **endianness** of an architecture is the way it represents data.
Big: Most significant digit on the lowest address. Makes more sense to humans.
Little (default in CPSC 313): Most significant digit on the highest address. More commonly seen.

**0x1000:** 01 23 45 67 89 AB CD EF
**0x1008:** 03 69 BE 25 8C F1 47 AD

| | Big-endian | Little-endian |
|---|---|---|
| What is the 1-byte integer at location 0x100A? | 0xBE | 0xBE |
| What is the 4-byte integer at location 0x1004? | 0x89ABCDEF | 0XEFCDAB89 |

# Endianness

**0x1000** 01 23 45 67 89 AB CD EF

**0x1008** 03 69 BE 25 8C F1 47 AD

What is myStruct if read at location 0x1004?

Size of myStruct: 12 bytes

```
struct myStruct {
    char a[2]; // 2 bytes
    int32_t b; // 4 bytes
    int16_t c; // 2 bytes
};
```

Big-endian:

Little-endian:

# Endianness

**0x1000** 01 23 45 67 89 AB CD EF

**0x1008** 03 69 BE 25 8C F1 47 AD

What is myStruct if read at location 0x1004?

Size of myStruct: 12 bytes

```
struct myStruct {
    char a[2]; // 2 bytes
    int32_t b; // 4 bytes
    int16_t c; // 2 bytes
};
```

Big-endian:
a: [0x89, 0xAB], b: 0x0369BE25, C: 0x8CF1

Little-endian:
a: [0x89, 0xAB], b: 0x25BE6903, C: 0xF18C

Endianness does not change how items are placed in struct / array like structures.

# Part 3:
# Pointers (segfaults, dangling ptrs, mem leaks, stack vs heap memory, type casting)

# Stack vs Heap Memory Differences

- Global variables are **allocated** (given a spot) at **compile time** (**static** memory allocation)
- In general, the layout of stack frames are decided at compile time
  - parameters
  - local variables in each function
  - return addresses
- However, the stack itself is not allocated at compile time. We don't know how many stack frames there will be at any time, nor what the heap will look like, until **runtime** (**dynamic** memory allocation)
- On the heap, we use malloc/free to manually manage memory

- Fun fact: a stack is a stack data structure; a heap is not a heap data structure.

# Stack vs Heap Memory Differences (Summary)

| | Static or Global | Stack (Local) | Heap (Dynamic) |
|---|---|---|---|
| Layout (Align, Offset, Size) | Compile Time | Compile Time | Run Time |
| Address | Compile Time | Run Time | Run Time |
| Data | Compile Time/Run Time | Run Time | Run Time |
| Per Thread | No | Yes | No |
| Where? | Executable | OS Provided Memory | OS Provided Memory |

# Pointers Basics

- `*` : dereference operator, which is also used in pointer declaration
- `&` : (ampersand) "address of" operator
- remember: pointers are addresses! (the address of the thing it is pointing to)
  - this is not the address of the pointer

Initialize a pointer:

```
int x = 10;
int * p = &x;  // pointer p points to x (p contains the address of x in memory)
*p = 20;       // now if we ask for the value of x, it will be 20
```

# Pointer Arithmetic

Q: What is the final value of ptr?

```
int32_t* ptr = 0x7fffffe0;
ptr++;
```

# Pointer Arithmetic

Q: What is the final value of ptr?

```
int32_t* ptr = 0x7fffffe0;
ptr++;
```

A: `0x7fffffe4`

Adding and subtracting an integral value to pointers works like a data index!

- &ptr[i] == ( ptr + i ) == ( ((char*)ptr) + i * sizeof(*ptr) )
- Useful for iterating through data without having to store index separately
- If you need byte arithmetic use something like char* and cast back to original type to access

# Pass by Value vs Pass by Reference

```
void my_swap1(int* i, int* j) {
    int* temp = i;
    i = j;
    j = temp;
}
```

```
void my_swap2(int i, int j) {
    int temp = i;
    i = j;
    j = temp;
}
```

Q: What function will correctly swap a and b?

```
int a = 3;
int b = 4;
my_swapN(&a, &b);
```

# Pass by Value vs Pass by Reference

```
void my_swap1(int* i, int* j) {
    int* temp = i;
    i = j;
    j = temp;
}
```

```
void my_swap2(int i, int j) {
    int temp = i;
    i = j;
    j = temp;
}
```

Q: What function will correctly swap a and b?

```
int a = 3;
int b = 4;
my_swapN(&a, &b);
```

A: Neither! We are assigning local variables in either case. We need to dereference the pointers!

# Correct Swap Function

```
void my_swap3(int* i, int* j) {
    int temp = *i;
    *i = *j;
    *j = temp;

}

int a = 3;
int b = 4;
my_swap3(&a, &b);
```

😊

# Segmentation Faults

- When you try to access memory that isn't supposed to be accessed
- Common case: you forget to initialize your pointers and dereference NULL

```
char *p = NULL;
*p = ‘a’;           // Segmentation fault: core dumped
```

# Memory Leaks

*Technically memory-safe*

When you allocate memory on the heap but do not deallocate it

```
int* ptr = (int *) malloc(sizeof(int));

*ptr = 10;

return 0; // return without freeing ptr
          // we should call free(ptr); before returning
```

# Memory Leaks

*Technically memory-safe*

When you allocate memory on the heap but do not deallocate it

Q: Why is this a problem?

```
int* ptr = (int *) malloc(sizeof(int));

*ptr = 10;

return 0; // return without freeing ptr
          // we should call free(ptr); before returning
```

# Memory Leaks

*Technically memory-safe*

When you allocate memory on the heap but do not deallocate it

Q: Why is this a problem? A: It's wasteful

```
int* ptr = (int *) malloc(sizeof(int));

*ptr = 10;

return 0; // return without freeing ptr
          // we should call free(ptr); before returning
```

# Memory Leaks

*Technically memory-safe*

When you allocate memory on the heap but do not deallocate it

Q: Why is this a problem? A: It's wasteful

Q: Can we have memory leaks using stack memory?

```
int* ptr = (int *) malloc(sizeof(int));

*ptr = 10;

return 0; // return without freeing ptr
          // we should call free(ptr); before returning
```

# Memory Leaks

*Technically memory-safe*

When you allocate memory on the heap but do not deallocate it

Q: Why is this a problem? A: It's wasteful

Q: Can we have memory leaks using stack memory?  A: No.

```
int* ptr = (int *) malloc(sizeof(int));

*ptr = 10;

return 0; // return without freeing ptr
          // we should call free(ptr); before returning
```

# Dangling Pointers

*Anyone remember lab_debug from 221?*

When a pointer points to something that "no longer exists"

Q: Why is this a problem?

```
char *p = NULL;
{                   // local scope
    char c;
    p = &c;
}                   // dangling ptr: c does not exist outside the {braces}

//  here
```

# Dangling Pointers

*Anyone remember lab_debug from 221?*

When a pointer points to something that "no longer exists"

Q: Why is this a problem?  A: Allows undefined behaviour, use-after-free, or (best case) segfault

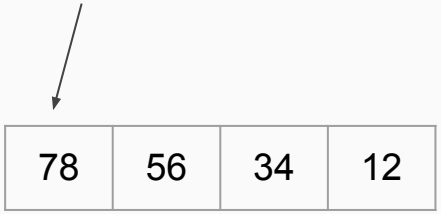
```
char *p = NULL;
{                   // local scope
    char c;
    p = &c;
}                   // dangling ptr: c does not exist outside the {braces}

//  here
```

# Type Casting (Pointers)

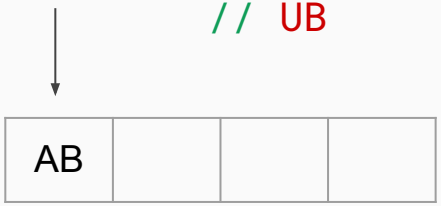
```
int32_t i = 0x12345678;
char* cp = (char *)&i;

*cp = ? //  0x78
```




```
char c = 0xAB;
int32_t* ip=(int32_t *)&c;

*ip = ?   // 0x??????AB
          // UB
```



# Part 4:
# C Programs (arguments, debugging)

# Writing a Main Function

https://stackoverflow.com/questions/3024197/what-does-int-argc-char-argv-mean

**argv** and **argc** are how command line arguments are passed to main() in C.

**argc** will (in practice) be 1 plus the number of actual arguments in **argv**, as virtually all implementations will prepend the name of the program to the array.

# Example of a Main Function

```
int main(int argc, char* argv[]) {
   printf("Have %d arguments:\n", argc);
   for (int i = 0; i < argc; ++i) {
      printf("%s\n", argv[i]);
   }
}
```

```
gcc test.c -o test
```

# Example of a Main Function

```
int main(int argc, char* argv[]) {
   printf("Have %d arguments:\n", argc);
   for (int i = 0; i < argc; ++i) {
      printf("%s\n", argv[i]);
   }
}
```

```
gcc test.c -o test
```

Input:

```
./test a1 b2 c3
```

Q: What are in argc/argv?

# Example of a Main Function

```
int main(int argc, char* argv[]) {
   printf("Have %d arguments:\n", argc);
   for (int i = 0; i < argc; ++i) {
      printf("%s\n", argv[i]);
   }
}
```

```
gcc test.c -o test
```

Input:

```
./test a1 b2 c3
```

Q: What are in argc/argv?

A:

```
argc = 4
argv[0] = "./test"
argv[1] = "a1"
argv[2] = "b2"
argv[3] = "c3"
```

# Printf Statements

**%d** : Integer

**%f** : Float

**%c** : Character

**%s** : String

**%p** : Pointer

**%x** : Hexadecimal

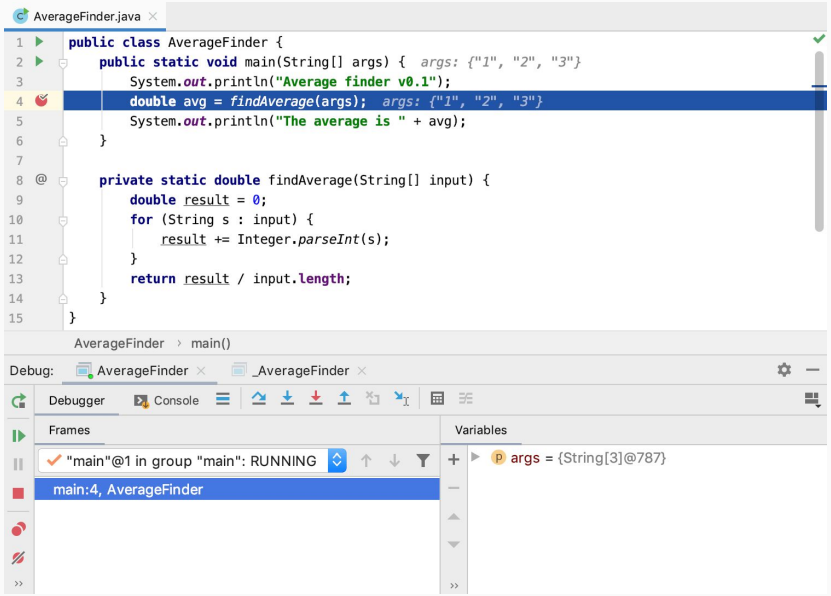
**Basic Syntax:**

```
char stringVar[] = “Hello World!”;
printf(“Value of stringVar: %s\n”, stringVar)
```

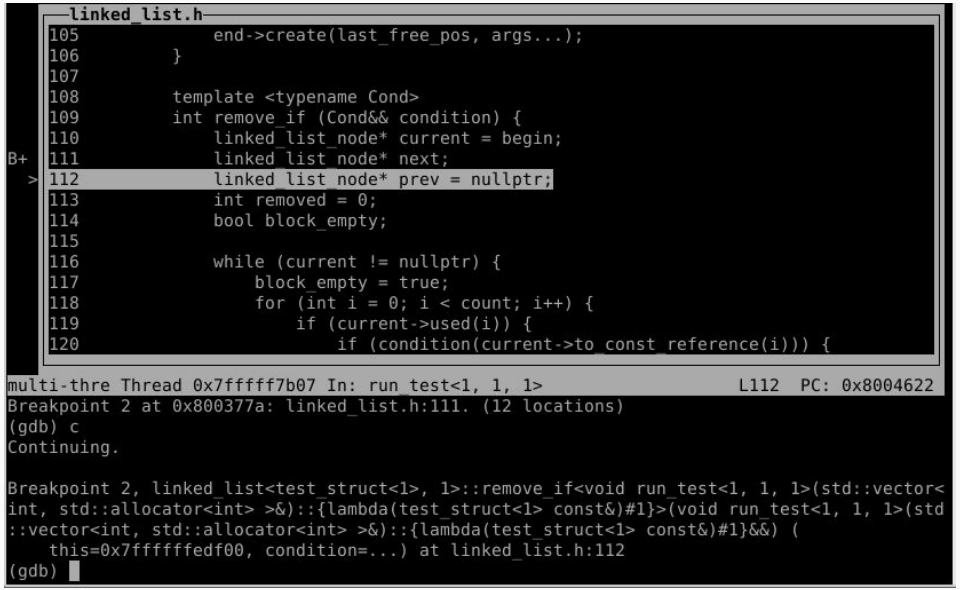
# Debugging C programs

1. You should compile your program with `-g` flag for easy debugging
   a. `gcc -g main.c -o main`
   b. `make` (if there's already a makefile)
2. One way to run gdb is by passing the your executable's name as its argument:
   a. `gdb main`
3. Then, you can run your program using `run`
   a. It'd be helpful to run `layout next` to have code view in your terminal

# Debugging C programs




IntelliJ built-in debugger for Java




C program debugger: GDB

# GDB Common Commands

`start` / `r`(un)

`b`(reak)

`watch`

`bt` backtrace

`clear`

`p`(rint)

`c`(ontinue)

`n`(ext)

`s`(tep)

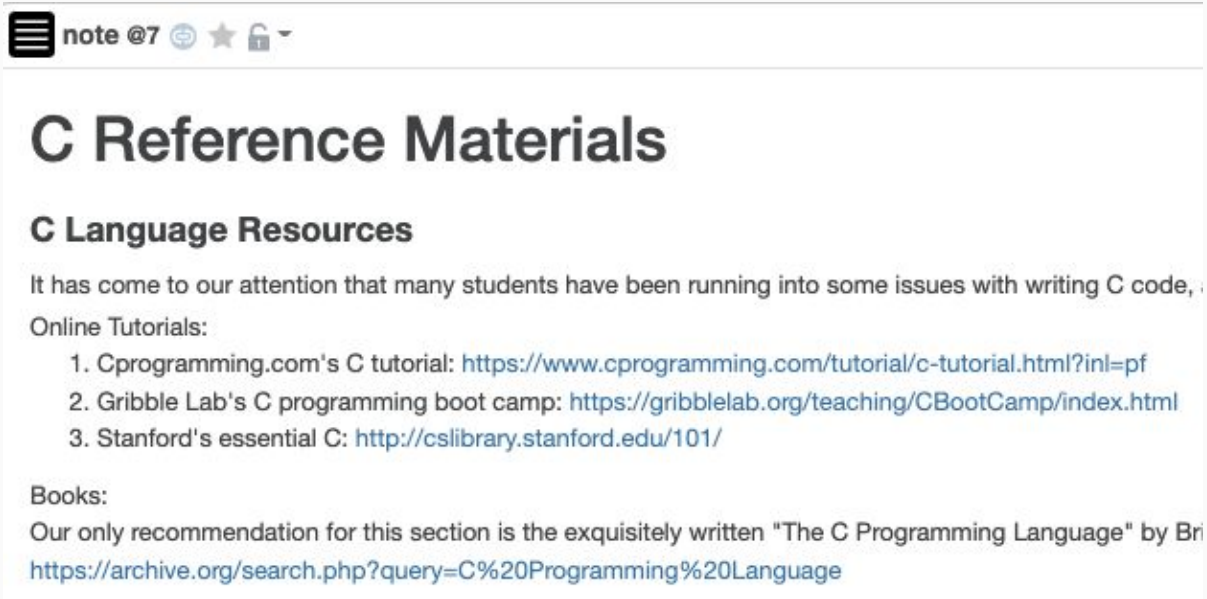
`quit`

Use GDB Reference (Google) for more commands!

You will not learn these commands by looking at the slide. You will learn them by using them.

Demonstration time! (with code snippet)

# For Other Resources, See Piazza!



note @7

## C Reference Materials

### C Language Resources

It has come to our attention that many students have been running into some issues with writing C code,

Online Tutorials:

1. Cprogramming.com's C tutorial: https://www.cprogramming.com/tutorial/c-tutorial.html?inl=pf
2. Gribble Lab's C programming boot camp: https://gribblelab.org/teaching/CBootCamp/index.html
3. Stanford's essential C: http://cslibrary.stanford.edu/101/

Books:

Our only recommendation for this section is the exquisitely written "The C Programming Language" by Br

https://archive.org/search.php?query=C%20Programming%20Language

Also:

- https://en.cppreference.com/w/c/language (language reference, standard C)
- https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html (GNU C reference)
- https://www.onlinegdb.com/online_c_compiler (minimal online compiler for C)

# Bonus: Right-left rule for reading C declarations

*See https://cseweb.ucsd.edu/~gbournou/CSE131/rt_lt.rule.html*

```
int *(*mystery())();
```

What is this? A function pointer? A function pointer pointer? What does it return?

# Right-left rule

Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"

1. Start at the identifier
2. Read to the **right** until you hit the end, or a closing paren ")"
3. Read to the **left** until you hit the end, or an opening paren "("

Then repeat steps 2 and 3 until you've read everything

# Right-left rule

Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"

```
int *p[]
```

1. Start at the identifier
2. Read to the **right** until you hit the end, or a closing paren ")"
3. Read to the **left** until you hit the end, or an opening paren "("

Then repeat steps 2 and 3 until you've read everything

# Right-left rule

Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"

```
int *p[]
```

1. Start at the identifier
2. Read to the **right** until you hit the end, or a closing paren ")"
3. Read to the **left** until you hit the end, or an opening paren "("

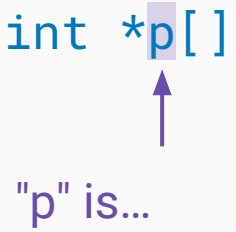
Then repeat steps 2 and 3 until you've read everything

# Right-left rule

Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"




1. Start at the identifier
2. Read to the **right** until you hit the end, or a closing paren ")"
3. Read to the **left** until you hit the end, or an opening paren "("

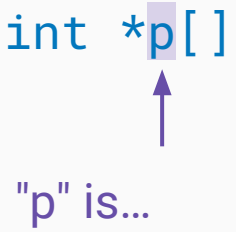
Then repeat steps 2 and 3 until you've read everything

# Right-left rule

Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"




1. Start at the identifier
2. Read to the **right** until you hit the end, or a closing paren ")"
3. Read to the **left** until you hit the end, or an opening paren "("

Then repeat steps 2 and 3 until you've read everything

# Right-left rule

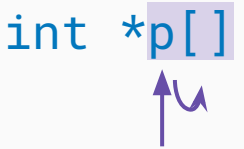
Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"

1. Start at the identifier
2. Read to the **right** until you hit the end, or a closing paren ")"
3. Read to the **left** until you hit the end, or an opening paren "("

Then repeat steps 2 and 3 until you've read everything




"p" is (an)...

... array of...

# Right-left rule

Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"

1. Start at the identifier
2. Read to the **right** until you hit the end, or a closing paren ")"
3. Read to the **left** until you hit the end, or an opening paren "("

Then repeat steps 2 and 3 until you've read everything

```
int *p[]
```



"p" is (an)...

... array of...

# Right-left rule

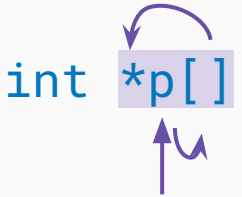
Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"

1. Start at the identifier
2. Read to the **right** until you hit the end, or a closing paren ")"
3. Read to the **left** until you hit the end, or an opening paren "("

Then repeat steps 2 and 3 until you've read everything




"p" is (an)...

... array of...

... pointer to...

# Right-left rule

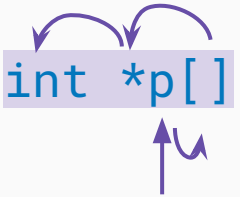
Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"

1. Start at the identifier
2. Read to the **right** until you hit the end, or a closing paren ")"
3. Read to the **left** until you hit the end, or an opening paren "("

Then repeat steps 2 and 3 until you've read everything




"p" is (an)...

... array of...

... pointer to...

int.

# Right-left rule

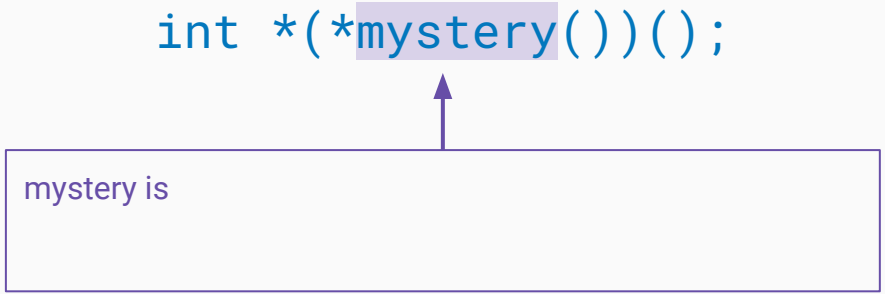
Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"

1. Start at the identifier
2. Read right
3. Read left

// goto 2



# Right-left rule

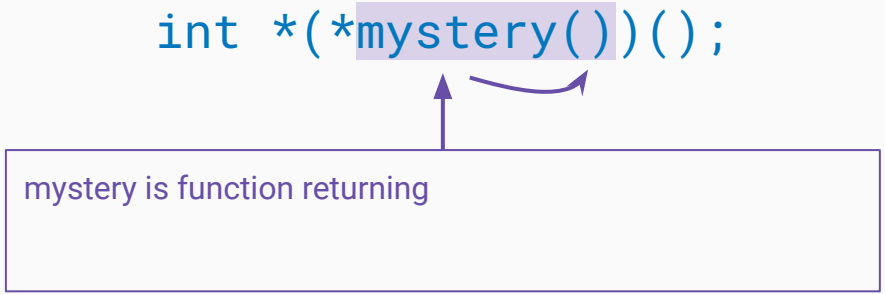
Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"

1. Start at the identifier
2. Read right
3. Read left

// goto 2



# Right-left rule

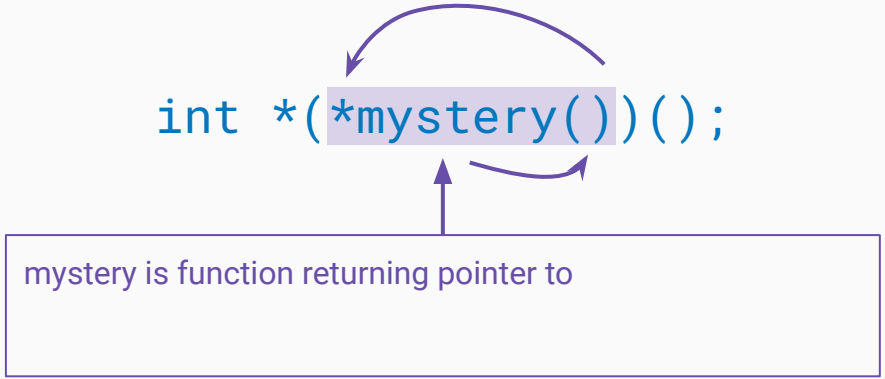
Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"

1. Start at the identifier
2. Read right
3. Read left

// goto 2



# Right-left rule

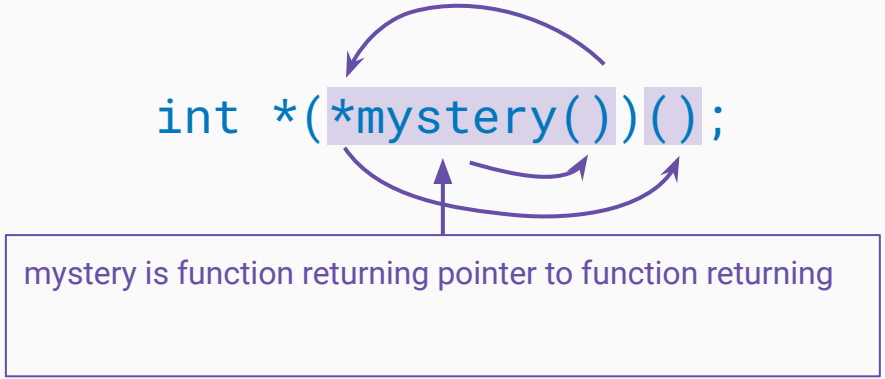
Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"

1. Start at the identifier
2. Read right
3. Read left

// goto 2



# Right-left rule

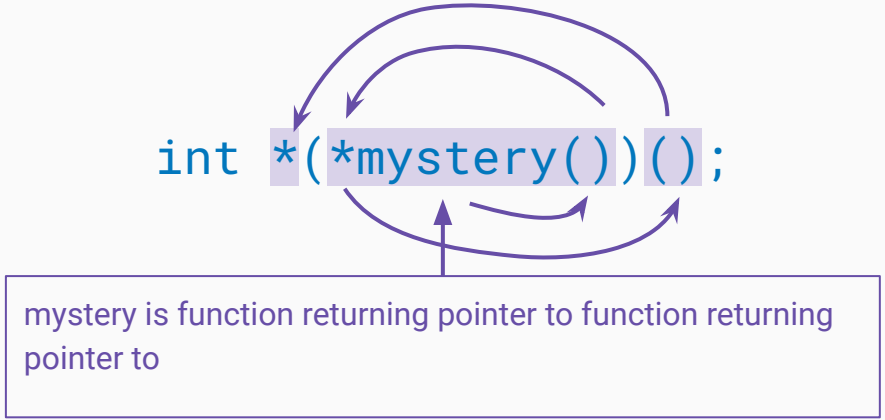
Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"

1. Start at the identifier
2. Read right
3. Read left

// goto 2



# Right-left rule

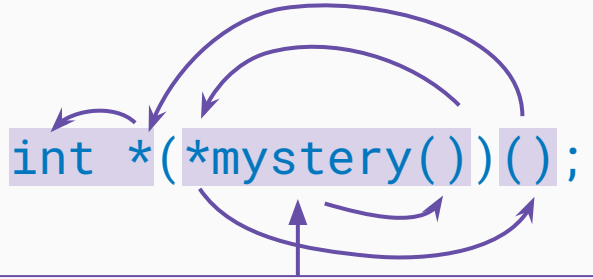
Read * as "pointer to"

Read [ ] as "array of"

Read ( ) as "function returning"

1. Start at the identifier
2. Read right
3. Read left

// goto 2




mystery is function returning pointer to function returning pointer to int!

# Any questions?