Java and C tips

Intro to Computer Systems, Fall 2022

Instructor: Travis McGaha

TAs:

Ali Krema	Andrew Rigas	Anisha Bhatia
Audrey Yang	Craig Lee	Daniel Duan
David LuoZhang	Eddy Yang	Ernest Ng
Heyi Liu	Janavi Chadha	Jason Hom
Katherine Wang	Kyrie Dowling	Mohamed Abaker
Noam Elul	Patricia Agnes	Patrick Kehinde Jr.
Ria Sharma	Sarah Luthra	Sofia Mouchtaris

Upcoming Due Dates

- HW10/11 (J compiler) is due Friday December 9th
  - HW10 & 11 make up a 2-part assignment that take a while to complete.
  - Recitation for this assignment has been VERY helpful
  - Can grant extensions on this, but there will be reduced office hours and Ed activity after a bit
  - **Took some students a long time in Fall 2021**
- Final Exam: Thursday December 15th
  - Cumulative
  - More info coming soon

J Compiler Common Mistakes

- DON’T FORGET TO ADD HEADER GUARDS

- next_token
  - When you read a comment, don’t forget to read till the rest of the line

- ASM generation:
  - Some 16-bit LITERALs require both CONST and HICONST to load that value into a register
  - The prologue/epilogue is wrong, you can mostly copy this off of the slides though.
  - Generating unique labels/handling nested control structures

Lecture Outline

None of this is on the final exam or HW10/HW11

- **Java vs C**
  - **Java Datatypes**
  - Java Compilation
  - Java Garbage Collector
- C tips & Practice

Comparing Java and C

- Perquisite to this course: CIS 1100
  - You all have experience programming in Java
  - Java the first language for most of you

- "The Hardest programming language you learn is the second one that you learn."
  - May not fully be true, but it is common to struggle with the differences between the languages
  - Doesn't help that C and Java look VERY similar

- Hopefully this comparison gives you a better understanding of both Java and C

Disclaimer

- Java and C both can have multiple implementations.
  - Some things we discuss in this lecture may not be guaranteed, but instead may vary.
- C: Leaves some details that can vary from machine to machine and/or compiler to compiler
  - Example: what is the size and sign of the **`char`** datatype?
  - Example: What happens when we return the address of data in the current stack frame?
- Java: the language specification provides an abstraction
  - We can understand how the code should behave, but it may do things differently when actually compiled & run

Java Data Types: Primitives

- Primitive types are pretty much the same as C
  - **`int`**, **`float`**, **`double`**, etc.
  - Java doesn't have **`unsigned`** types to avoid issues with converting & comparing between signed/unsigned types

- `char`:
  - **`char`** in C is 1-byte which represents an ASCII character
  - **`char`** in Java is 2-bytes for 2-byte Unicode characters

- Primitive Size:
  - Java is designed to be portable, primitives are fixed in size
  - C primitive sizes can vary from machine to machine
  - Example: `int` **is 4-bytes** in Java, and **is usually 4-bytes** in C

Java Data Types: Pointers

- Pointers are a type of primitive in C. Can be used to access memory but we can also deal with the address directly (pointer arithmetic, get address of with `&`)

- Java has references, which are almost like "protected" or "hidden" pointers.
  - All Object variables are actually Object References
  - Much more constrained in how you use them to try and minize possible memory usage errors

- Both have `NULL` or `null` to indicate an unused/empty pointer/references. (NULL typically represented as 0)

Java Data Types: Objects

- C doesn't have true Objects, but code can have "objects" or structs. This data can exist in many places in memory.

- Java has Object support. All objects in Java are stored on the heap. The "`new`" keyword allocates memory dynamically, like how `malloc` allocates space.

Java Data Types: Objects Example

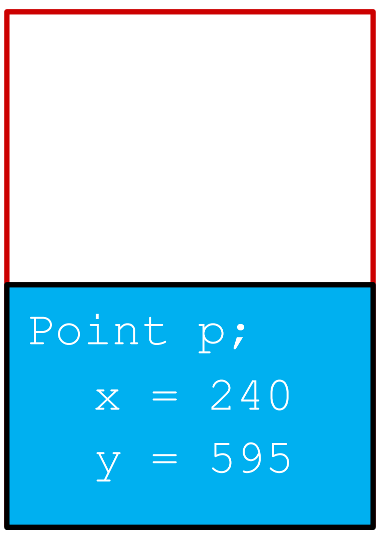
- Consider we have a struct **`Point`** in C and object **`Point`** in Java. Each contains two integers, an X and a Y.

C

```c
int main() {
   Point p;
   p.x = 240;
   p.y = 595;
}
```

Java

```java
public static void main(String args[]) {
   Point p;
   p.x = 240;
   p.y = 595;
}
```



Stack frame for `main()`

NULL POINTER EXCEPTION

Point p is an uninitialized references (an uninitialized pointer)

Java Data Types: Objects Example

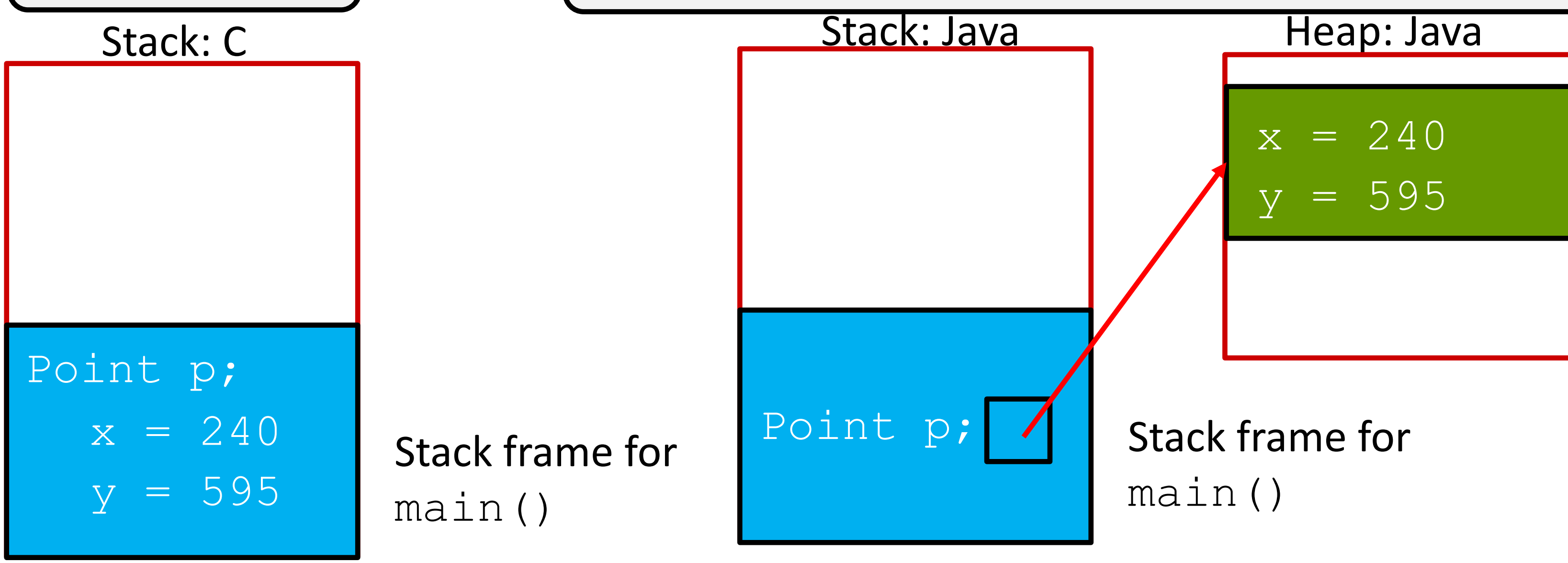
- Consider we have a struct **`Point`** in C and object **`Point`** in Java. Each contains two integers, an X and a Y.

C

```c
int main() {
  Point p;
  p.x = 240;
  p.y = 595;
}
```

Java

```java
public static void main(String args[]) {
  Point p = new Point();
  p.x = 240;
  p.y = 595;
}
```

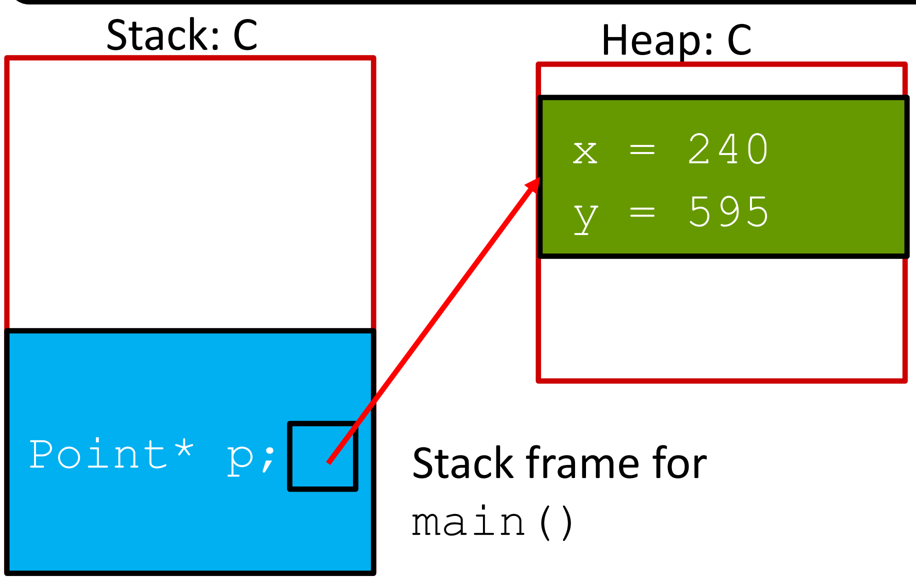


C Objects Heap Example

- C can also have "references" to things on the heap, but it is more explicit in the code

C

```
int main() {
  Point* p = malloc(sizeof(Point));
  p->x = 240;
  p->y = 595;
}
```



Java Data Types: Arrays

- C Arrays:
  - elements are garbage by default
  - Length not stored
  - Does not check bounds when accessing array

- Java Arrays:
  - elements are initialized to 0 or `null`
  - Length stored as an immutable field at start of the array
  - Every access to the array does a bounds check, throwing an exception if the index is illegal

Java Data Types: Arrays in Memory

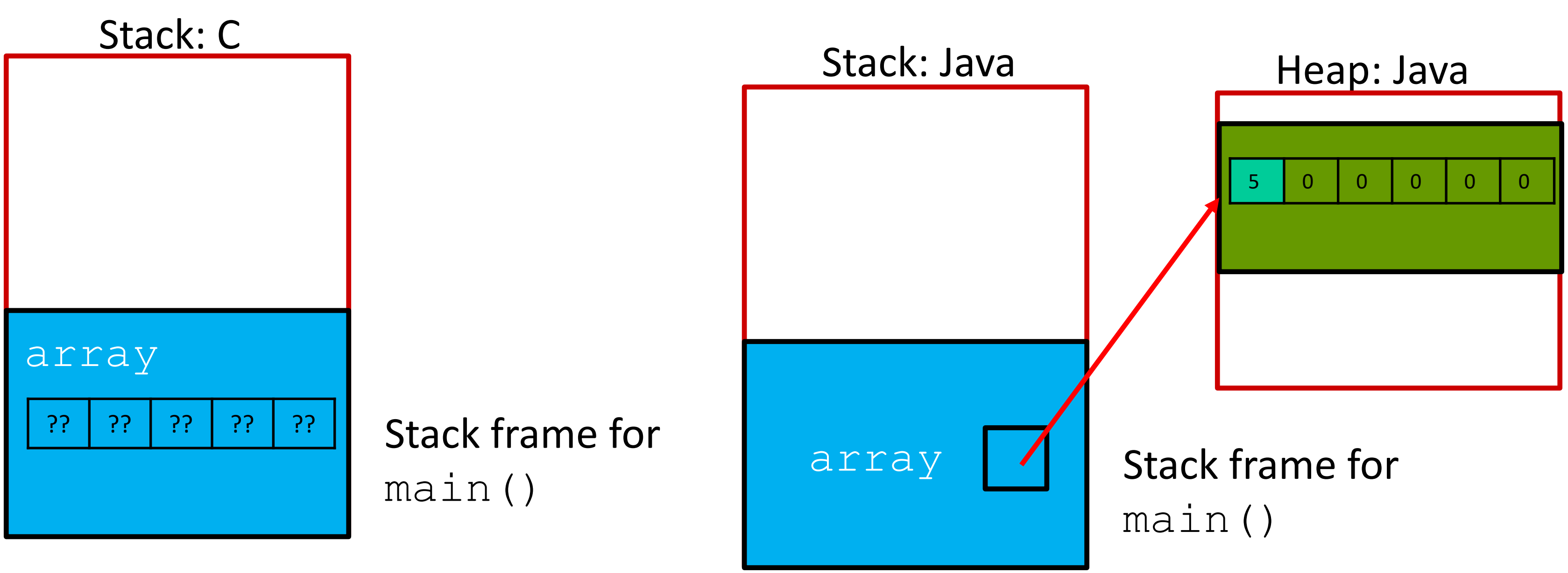
- Example Code:

C

```
int main() {
  int array[5];
}
```

Java

```
public static void main(String args[]) {
  int[] array = new int[5];
}
```



Java Data Types: Strings

- C strings:
  - ASCII Characters
  - Pretty much an array of characters
  - Null terminated
  - Can be modified

- Java strings:
  - Unicode Characters
  - An Object
  - Bounded by length like arrays in Java (with a 4-byte int field)
  - Are immutable

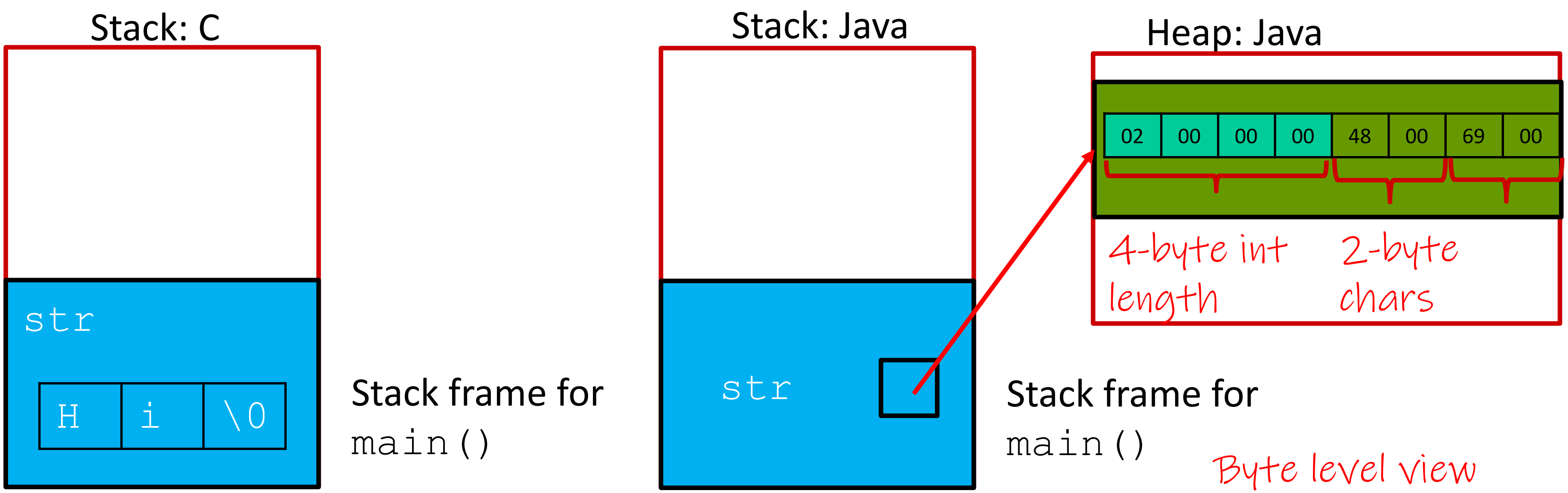
Java Data Types: Strings in Memory

C

```c
int main() {
  char str[3] = "Hi";
}
```

Java

```java
public static void main(String args[]) {
  String str = new String("Hi");
}
```



Lecture Outline

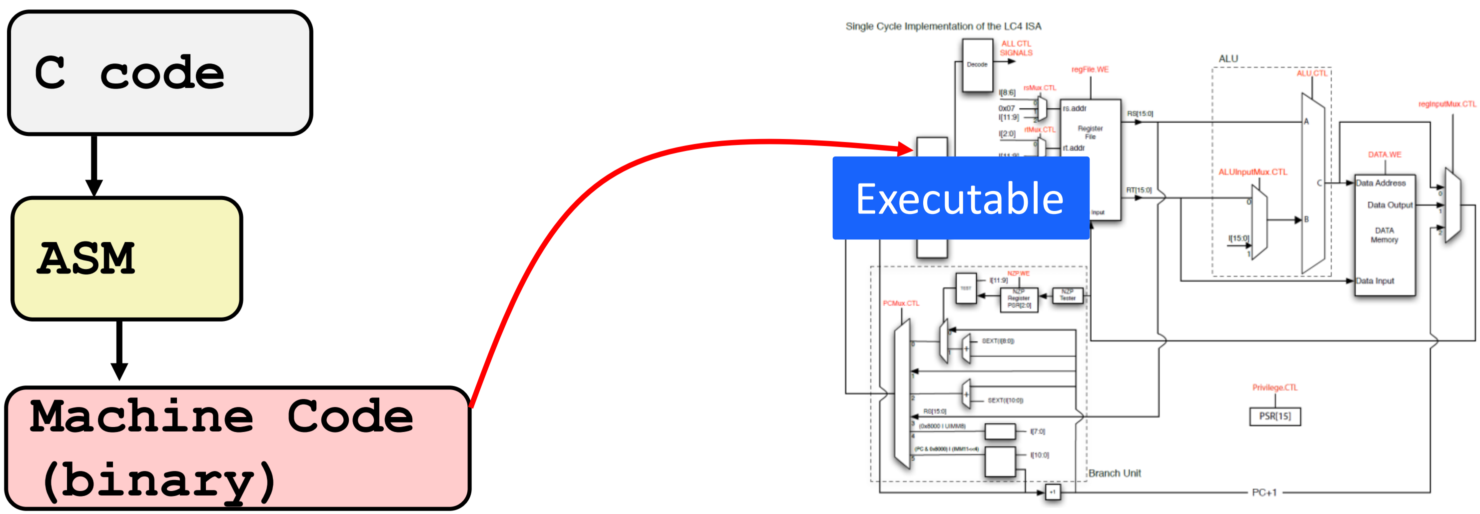
None of this is on the final exam or HW10/HW11

- **Java vs C**
  - Java Datatypes
  - **Java Compilation**
  - Java Garbage Collector
- C tips & Practice

Compilation

- In this class, we've walked through the C compilation Process
  - C codes is compiled into assembly instructions
  - Assembly instructors are assembled into machine code

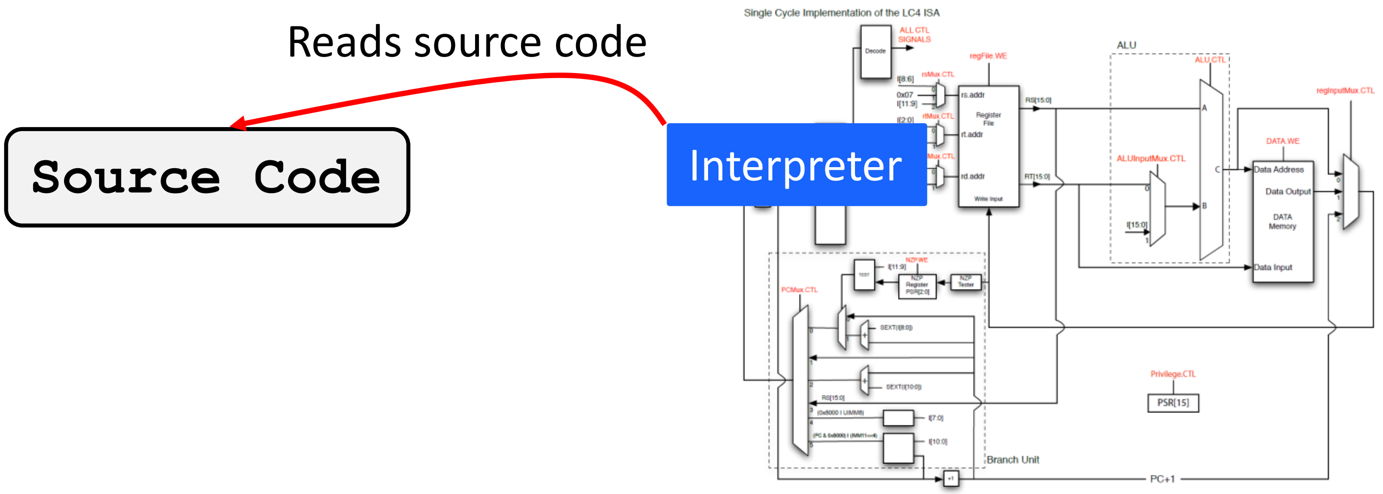
- At run-time, machine Code Is loaded directly into program memory and run directly on the processor



Interpreters

- There exist other ways for programming languages to run on a computer. A common method is using interpreters
  - Python, Lisp, Javascript, etc.

- The interpreter is a program that runs directly on the processor, reads your code, and interprets how to emulate the execution of your code.



Intermediate Formats

- Some languages are not read directly by the interpreter and instead are translated to some intermediary format
  - When we compile Java code, we are compiling from Java to Java bytecode

- Byte code provides an easier format for the interpreter to read our code

- Java bytecode can be used to implement other programming languages,
  - Kotlin, Scala, etc

Usual Java Compilation

- Java code is first compile to Java bytecode by a java compiler

- Java Byte code is then run on the Java Virtual Machine (JVM) which acts sort of like a Java bytecode interpreter

- There are other ways to compile and run Java and there are many optimizations that can be made to

JIT

- Just In Time (JIT) compilation
  - The interpreter/run-time environment will compile some bytecode into machine code while the program is running to try and execute the code faster.

- Translating to machine code has some overhead cost, especially if the code translation is complex or there are a lot of checks for optimization

- Some interpreters/environments will try to analyze the code to see which parts of bytecode is worth translating to machine code

Interpreters VS Compilers

- Interpreters make it easier to run on different architectures since the environment of the program is controlled by the interpreter
- Interpreters usually have deep connection to a debugger, making development of a debugger easier
- Allow for a garbage collector to implicitly work while the program is running
- Interpreters have more overhead cost than compiled languages and run slower

- Some languages aren't clearly interpreted or compiled

Lecture Outline

None of this is on the final exam or HW10/HW11

- **Java vs C**
  - Java Datatypes
  - Java Compilation
  - **Java Garbage Collector**
- C tips & Practice

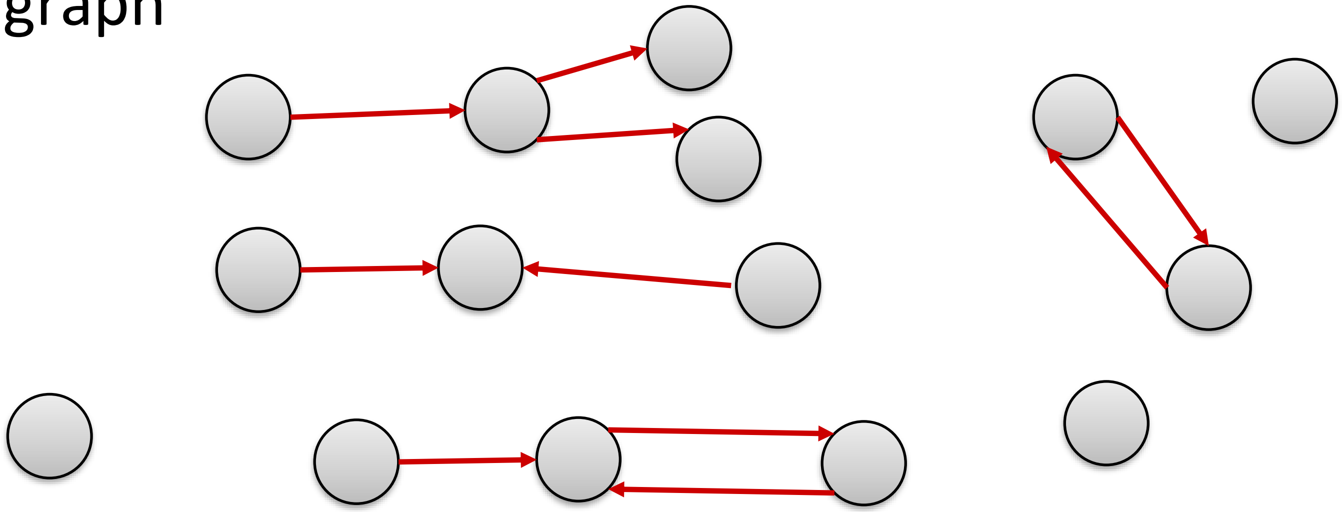
Garbage Collection

- Garbage Collection:
  - automatically deallocates memory on the heap.
- Commonly used in many programming Languages:
  - Java, C#, Go, Javascript, Ruby, Julia, …
- Requires some overhead to check and see what memory can be deallocated and which is still being used
- Many implementations and optimizations on garbage collection

Trace Garbage Collection

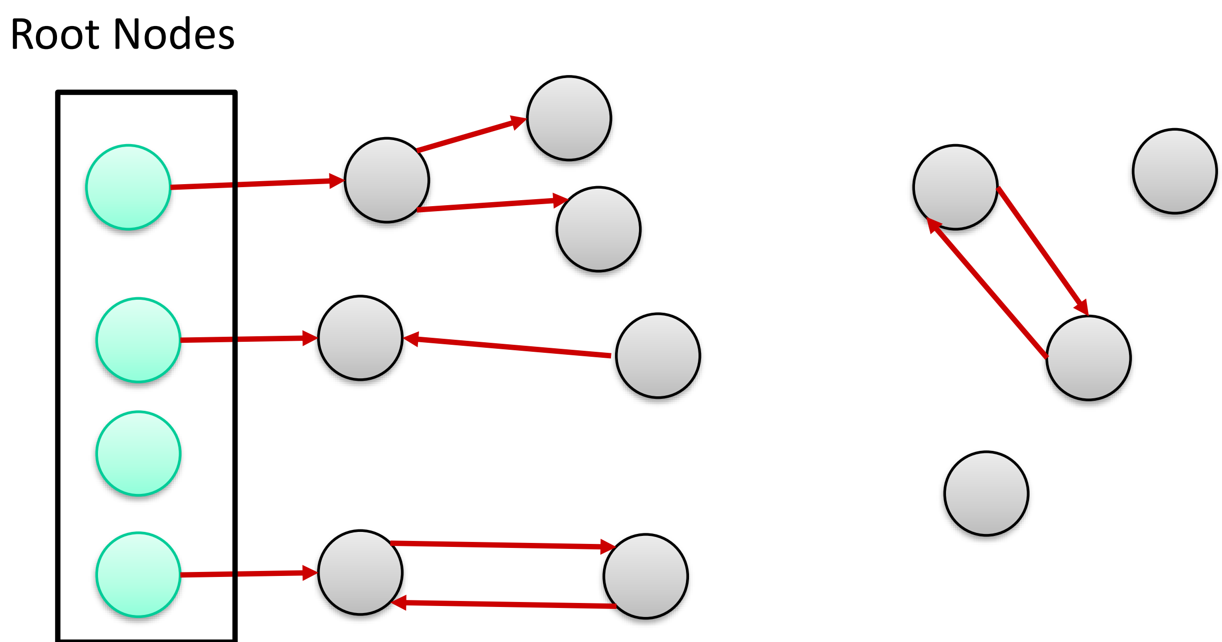
- To decide which memory can be deallocated, garbage collectors often trace memory to see which memory is still "reachable" by the user program.

- The garbage collector keeps track of all allocations and can draw memory references & allocations like a directed graph



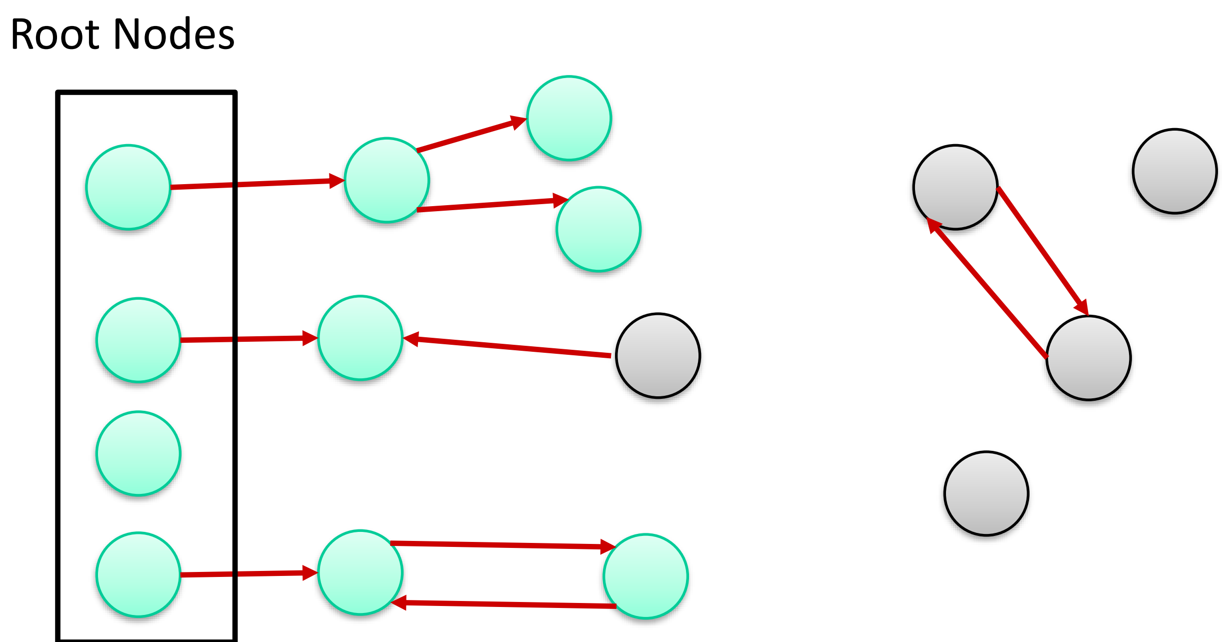
Trace Garbage Collection

- We start with a set of allocations we know are reachable and call these Root Nodes (usually these are held as references in local variables still on the stack)



Trace Garbage Collection

- We start with a set of allocations we know are reachable and call these Root Nodes (usually these are held as references in local variables still on the stack)
- We then trace through all references. Anything referenced ***from*** a reachable node is reachable.



Lecture Outline

None of this is on the final exam or HW10/HW11

- Java vs C
  - Java Datatypes
  - Java Compilation
  - Java Garbage Collector
- **C tips & Practice**

C: Common Mistakes

- The most common mistakes I notice in office hours teaching usually deal with handling memory:
  - How parameters are passed
  - Using Output parameters
  - Exceeding the bounds of an array
  - Issues with deallocating memory

C is Call-By-Value

- C (and Java) pass arguments by *value*
  - Callee receives a **local copy** of the argument
    - Register or Stack
  - If the callee modifies a parameter, the caller's copy *isn't* modified

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```

Broken Swap

Note: Arrow points to *next* instruction.

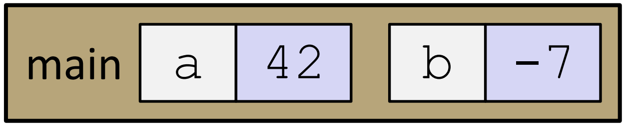
brokenswap.c

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```

Broken Swap

brokenswap.c

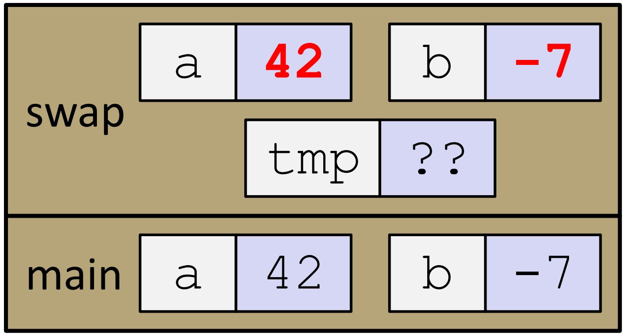
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



Broken Swap

brokenswap.c

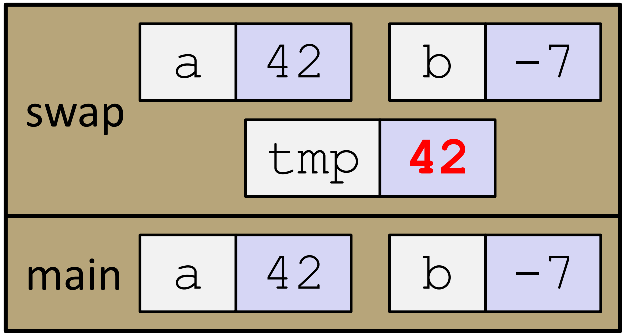
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



Broken Swap

brokenswap.c

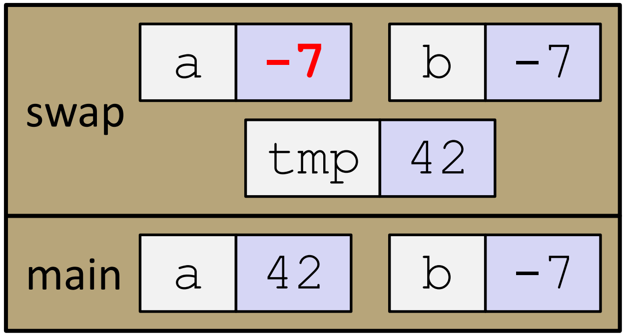
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



Broken Swap

brokenswap.c

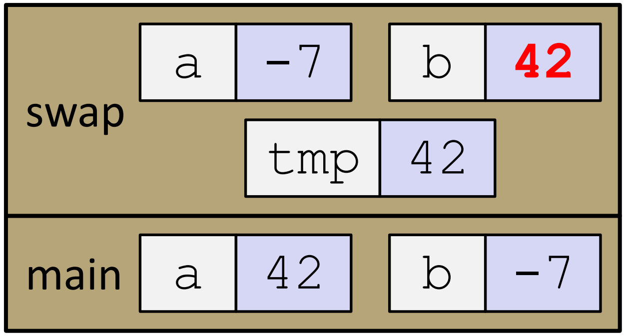
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



Broken Swap

brokenswap.c

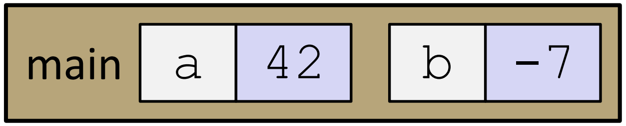
```c
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



Broken Swap

brokenswap.c

```c
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



Faking Call-By-Reference in C

- Can use pointers to *approximate* call-by-reference
  - Callee still receives a **copy** of the pointer (*i.e.* call-by-value), but it can modify something in the caller's scope by dereferencing the pointer parameter

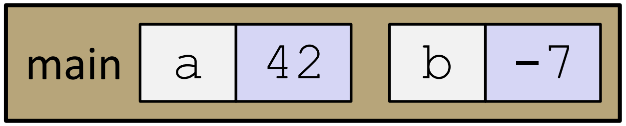
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(&a, &b);
  ...
```

Fixed Swap

Note: Arrow points to *next* instruction.

swap.c

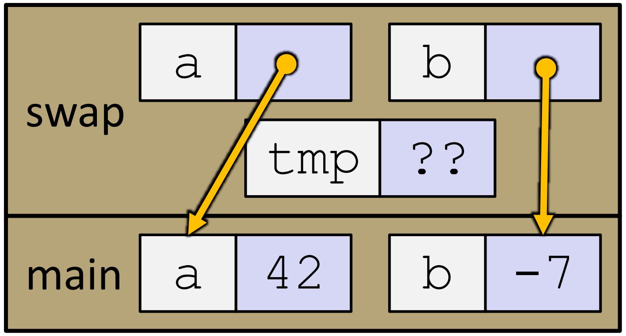
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(&a, &b);
  ...
```



Fixed Swap

swap.c

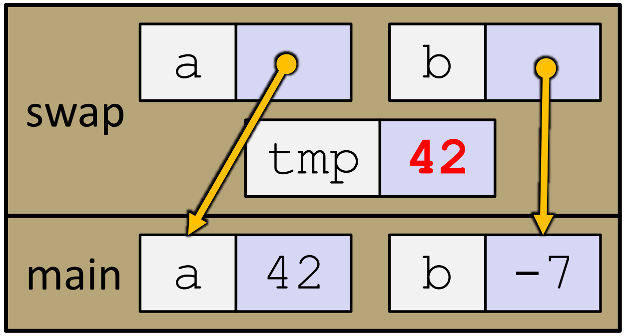
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(&a, &b);
  ...
```



Fixed Swap

swap.c

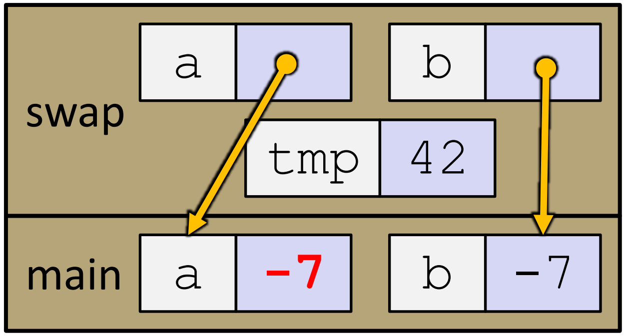
```c
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(&a, &b);
  ...
```



Fixed Swap

swap.c

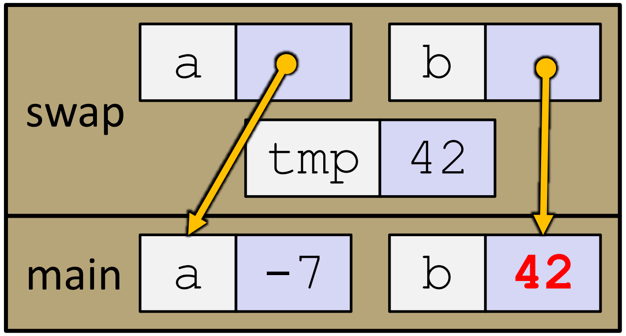
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(&a, &b);
  ...
```



Fixed Swap

swap.c

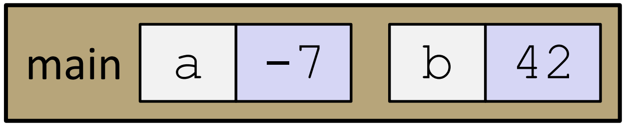
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(&a, &b);
  ...
```



Fixed Swap

swap.c

```c
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(&a, &b);
  ...
```



Practice Problem:

- What does this code print?

```
typedef struct point_st {
  int x, y;
} Point;

void increment_point(Point p) {
  p.x++;
  p.y++;
}

int main() {
  Point p = {1, 5};
  increment_point(p);
  printf("x: %d y: %d\n", p.x, p.y);
}
```

Practice Problem:

- What does this code print?

```
typedef struct point_st {
  int x, y;
} Point;

void increment_point(Point p) {
  p.x++;
  p.y++;
}

int main() {
  Point p = {1, 5};
  increment_point(p);
  printf("x: %d y: %d\n", p.x, p.y);
}
```

Structs are passed by value

This code prints “x: 1, y: 5”

Practice Problem: Fixed

- Fixed code that uses pointers to simulate pass-by-reference

```
typedef struct point_st {
  int x, y;
} Point;

void increment_point(Point* p) {
  p->x++;
  p->y++;
}

int main() {
  Point p = {1, 5};
  increment_point(&p);
  printf("x: %d y: %d\n", p.x, p.y);
}
```

Practice Problem:

- What is wrong with this code?

```
#define LINE_LEN 250

int main() {
  FILE* f = fopen("Hi.txt", "r");
  if (f == NULL)
    return EXIT_FAILURE;

  char buf[10];

  while(fread(buf, sizeof(char), LINE_LEN, f)) {
    printf("%s", buf);
  }
  fclose(f);
  return EXIT_SUCCESS;
}
```

Practice Problem:

- What is wrong with this code?

```
#define LINE_LEN 250

int main() {
  FILE* f = fopen("Hi.txt", "r");
  if (f == NULL)
    return EXIT_FAILURE;

  char buf[10];

  while(fread(buf, sizeof(char), LINE_LEN, f)) {
    printf("%s", buf);
  }
  fclose(f);
  return EXIT_SUCCESS;
}
```

`buf` only has space for 10 characters, but `fread` tries to read 250!

This causes stack smashing, program probably crashes

Practice Problem:

- What is printed by this code?

```
int main() {
  uint16_t i = 0;
  for (i = 0; i < 65536; i++) {
    printf("%d ", i);
  }
  return EXIT_SUCCESS;
}
```

Practice Problem:

- What is printed by this code?

```
int main() {
  uint16_t i = 0;
  for (i = 0; i < 65536; i++) {
    printf("%d ", i);
  }
  return EXIT_SUCCESS;
}
```

Code goes infinite!
`i` is of type `uint16_t` which only has a max value of 65535!

Practice Problem:

- Similar Issue with unsigned types:

```
int main() {
  uint16_t i;
  for (i = 120; i >= 0; i--) {
    printf("%d ", i);
  }
  return EXIT_SUCCESS;
}
```

`i` never becomes negative, so the loop condition always evaluates to true

Buggy Program

- What is wrong with this program?
  - (ignoring style issues)

pair.h

```
#define FOO 240
typedef struct pair_st {
  int x, y;
} pair;
```

util.h

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

void Pair_Allocate(pair *out) {
  out = (pair *) malloc(sizeof(pair))
  out->x = 0;
  out->y = 0;
}

void Pair_Print(pair *p) {
  printf("(x:%d, y:%d)", p.x, p.y);
}
```

main.c

```
#include "pair.h"
#include "util.h"

int main(){
  pair * p;
  Pair_Allocate(p);
  p->x = FOO;
  p->y = 595;
  Pair_Print(*p);
}
```

Buggy Program

- What is wrong with this program?
  - (ignoring style issues)

No header guards!

pair.h

```
#define FOO 240
typedef struct pair_st {
  int x, y;
} pair;
```

util.h

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

void Pair_Allocate(pair *out) {
  out = (pair *) malloc(sizeof(pair))
  out->x = 0;
  out->y = 0;
}

void Pair_Print(pair *p) {
  printf("(x:%d, y:%d)", p.x, p.y);
}
```

Output parameter misuse

Needs to use -> syntax

main.c

```
#include "pair.h"
#include "util.h"

int main(){
  pair * p;
  Pair_Allocate(p);
  p->x = FOO;
  p->y = 595;
  Pair_Print(*p);
}
```

Memory leak

Shouldn't Dereference

Low Level Programming Review

- Complete the function "rand_string", which generates a random string of random length. Assume we have the following functions available to you:

  - ```
    int rand_len();
    // returns a random int in the range of 1 - 256
    ```

  - ```
    char rand_char();
    // returns a random printable character
    // (no '\0' or other special characters)
    ```

- If you finish, write a small main function that calls `rand_string` and prints out the string

Low Level Programming Review

```
// returns a random string and its length. Returns -1 on error
int rand_string(char **output) {
  // generate random length
  int len = rand_len();

  // allocate space for the string (+1 for null terminator)
  char* result = (char *) malloc((len+1)*sizeof(char));
  // error checking
  if (result == NULL)
      return -1;

  // assign random characters
  for (int i = 0; i < len; i++)
    result[i] = rand_char();

  // add null terminator
  result[len] = '\0';

  // return results
  *output = result;
  return len;
}
```

Low Level Programming Review

```
int main() {
  char* str;
  // generate random string
  int len = rand_string(&str);

  if (len == -1)
    return EXIT_FAILURE;

  printf("%s\n", str);

  free(str);

  return EXIT_SUCCESS;
}
```

C Tips: Ownership

- In C and C++, there is no garbage collector and instead the programmed has to manage memory allocation/deallocation themselves.

- To help reason about code, try to think of who has the "Ownership" or "Responsibility" to free code and who is just "Borrowing" memory.

- In the previous example, the comment for `rand_string` should say something like "The caller is responsible for freeing the resulting string"