Recitation 0

Welcome, everybody!

Recitation in a Summer Course

- Summer courses are very fast paced!

- For learning material, active practice > passive listening
  - Passive learning may boost your confidence, but active learning makes you aware of what you don't know

- We'll do our best to spend the most time on the topics you feel least confident on (through code and worksheets)

Recitation Logistics

- Turn in your worksheets! This is how we grade recitation participation
  - Due every Thursday at 11:59 pm.

- No need to submit the original file, - can just upload a copy of your answers on blank paper, especially since we make you draw out answers sometimes

- Recitations are not recorded! We hope that this makes participation easier

Today's Topics

- Grab bag of C Review:
  - Intro to C
  - Pointers & Arrays
  - Strings
  - Structs
  - Stack vs Heap
  - Output Parameters
  - Header files
- Intro to GDB
- Environment Setup

Pointers

Pointers

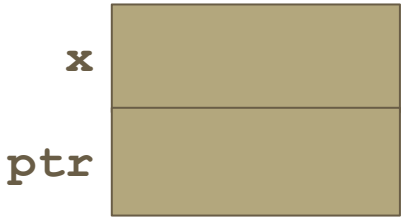
- Store addresses (an address is like an index in the array of memory)

- C is pass-by-value - to mimic pass-by-reference, you pass the pointer to the value

- **`*`** = “value of”
  - (UNLESS **`*`** is used in variable declaration, then it’s part of that variable’s type)
- **`&`** = “address of”

Pointers Example

```
int *ptr;

int x;
```

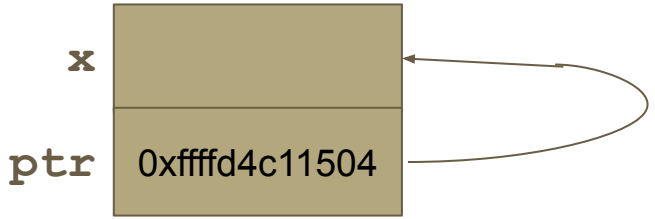


* = “value of”

& = “address of”

Pointers Example

```
int *ptr;
int x;
ptr = &x;
```

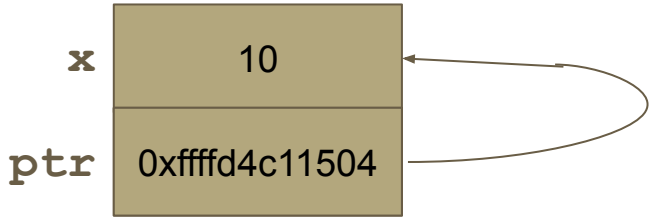


* = “value of”

& = “address of”

Pointers Example

```
int *ptr;
int x;
ptr = &x;
x = 10;
```

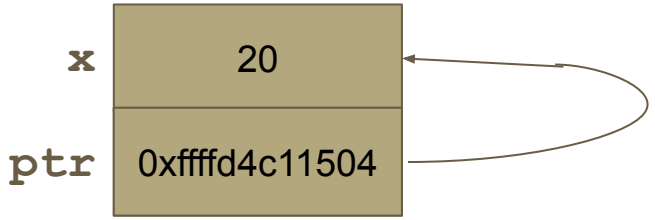


* = “value of”

& = “address of”

Pointers Example

```
int *ptr;
int x;
ptr = &x;
x = 10;
*ptr =  20;
```

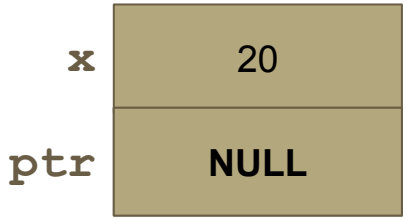


*** = “value of”**

& = “address of”

Pointers Example: Removing a Pointer

```
int *ptr;
int x;
ptr = &x;
x = 10;
*ptr =  20;
ptr = NULL;
```



*To ensure you never accidentally access freed memory, set the pointer to NULL when you know it's no longer needed! (think about malloc/free)

Arrays

Arrays

- Line of contiguous memory

- Lots of connections between pointers and arrays
  - For example, when you pass an array into a function, you're actually passing a pointer to the start of the array (a[0] is equivalent to *a)

- Must keep track of the length explicitly!

Exercise 1: Hello World

I wanted to print “Hello world!” in a fancier way using my new knowledge of pointers in C.

For some reason, things aren’t working. Help!

(Assume the printf syntax is fine)

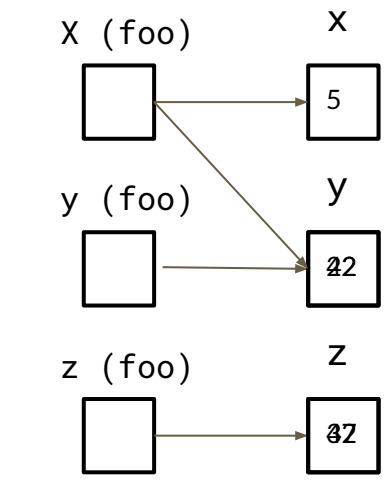
```c
#include <stdio.h>

int main() {
  char **a;
  fill_a(a);
  printf("%s %s", a[0], a[1]);
}

void fill_a(char **a) {
  a[0] = "Hello";
  a[1] = "world!\n";
}
```

Exercise 2: Memory Diagram

Draw a memory diagram for the following code and determine what the output will be.

Include a basic stack layout, see worksheet for details/example.

```
void foo(int32_t *x, int32_t *y, int32_t *z)
{
  x = y;
  *x = *z;
  *z = 37;
}
int main() {
  int32_t x = 5, y = 22, z = 42;
  foo(&x, &y, &z);
  printf("%d, %d, %d\n", x, y, z);
  return EXIT_SUCCESS;
}
```



So, the code will output 5, 42, 37.

Structs

Structs

- **Structs != Objects**

  Defining a struct:

```
struct Pokemon {
  char* name;
  int level;
};
```

Structs

- **Structs != Objects**

  Defining a struct:

```
struct Pokemon {
  char* name;
  int level;
};
```

```
This struct has two
fields: name and level
```

Structs

- **Structs != Objects**

Defining a struct:

```
struct Pokemon {
  char* name;
  int level;
};
```

Defining a struct variable:

```
struct Pokemon p;
p.name = "Pikachu";
p.level = 100;
p.name = "Raichu"
```

Structs

- **Structs != Objects**

Defining a struct:

```
struct Pokemon {
  char* name;
  int level;
};
```

Defining a struct variable:

```
struct Pokemon p;
p.name = "Pikachu";
p.level = 100;
p.name = "Raichu"
```

Must include "struct" in your variable declaration!

Structs

- **Structs != Objects**

Defining a struct:

```
struct Pokemon {
  char* name;
  int level;
};
```

Defining a struct variable:

```
struct Pokemon p;
p.name = "Pikachu";
p.level = 100;
p.name = "Raichu"
```

Can reassign fields of a struct variable

Structs versus Struct Pointers

Struct Variable:

```
struct Pokemon p;
p.name = "Pikachu";
p.level = 100;
p.name = "Raichu"
```

Struct Pointer Variable:

```
struct Pokemon *p = malloc(sizeof(struct Pokemon));
p->name = "Pikachu";
p->level = 100;
p->name = "Raichu"; //reassignment still possible
```

Structs versus Struct Pointers

Struct Variable:

```
struct Pokemon p;
p.name = "Pikachu";
p.level = 100;
p.name = "Raichu"
```

Struct Pointer Variable:

```
struct Pokemon *p = malloc(sizeof(struct Pokemon));
p->name = "Pikachu";
p->level = 100;
p->name = "Raichu"; //reassignment still possible
```

When using a struct pointer, -> (arrow) is used instead of . (period) to access a field. An equivalent assignment expression is:

```
*p.name = "Pikachu"; //using dereference operator
```

Structs on the Stack vs Heap

Bad Stack Allocation

```
struct Pokemon* bad_function() {
 struct Pokemon p; // stack
                   //allocated
 p.level = 100;
 return &p; // p is deallocated
            // at function return
}

void bad() {
  struct Pokemon* p =
                  bad_function();
  // undefined behavior!
  printf("Level: %d\n", p->level);
}
```

Good Stack Allocation

```
struct Pokemon* good_function() {
    struct Pokemon* p = malloc(sizeof(struct Pokemon));
    if (p != NULL) {
        p->level = 100;
    }
    return p;
}

void good() {
    struct Pokemon* p = good_function();
    printf("Level: %d\n", p->level); //  Safe
    free(p);
}
```

Structs on the Stack vs Heap

Bad Stack Allocation:

```
struct Pokemon* bad_function() {
  struct Pokemon p; // stack
                    //allocated
                   pointer
  p.level = 100;
  return &p; // p is deallocated
             // at function return
}

void bad() {
  struct Pokemon* p =
bad function();
  // undefined behavior!
  printf("Level: %d\n", p->level);
}
```

Structs on the Stack vs Heap

Bad Stack Allocation:

```
struct Pokemon* bad_function() {
  struct Pokemon p; // stack
                    //allocated
                   pointer
  p.level = 100;
  return &p; // p is deallocated
             // at function return
}

void bad() {
  struct Pokemon* p =
bad_function();
  // undefined behavior!
  printf("Level: %d\n", p->level);
}
```

Good Stack Allocation:

```
struct Pokemon good_function() {
  struct Pokemon p;
  p.level = 100;
  return p; // copy of p is returned
}

void good() {
  struct Pokemon p =
good_function();
  printf("Level: %d\n", p.level);
}
```

Structs on the Stack vs Heap

Bad Heap Allocation:

```
struct Pokemon* bad_function() {
 // bad style: unnecessary malloc
 struct Pokemon* p =
     malloc(sizeof(struct Pokemon));
 if (p != NULL) {
   p->level = 100;
   return p;
 }
 return NULL;
}

void bad() {
  struct Pokemon* p =
bad_function();
  printf("Level: %d\n", p->level);
}
```

Structs on the Stack vs Heap

Bad Heap Allocation:

```
struct Pokemon* bad_function() {
 // bad style: unnecessary malloc
 struct Pokemon* p =
     malloc(sizeof(struct Pokemon));
 if (p != NULL) {
   p->level = 100;
   return p;
 }
 return NULL;
}

void bad() {
  struct Pokemon* p =
bad_function();
  printf("Level: %d\n", p->level);
}
```

Good Heap Allocation:

```
void good_function(struct Pokemon *p) {
  if (p != NULL) {
    p->level = 100;
  }
}

void good() {
  // malloc in caller function: good!
  struct Pokemon *p =
      malloc(sizeof(struct Pokemon));
  good_function(p);
  printf("Level: %d\n", p->level);
}
```

Exercise 3: Struct Debugging

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

struct Pokemon {
  char* name;
  int level;
};

void rename_pokemon(struct Pokemon x, char*
new_name){
  x.name = new_name;
}

int main() {
  struct Pokemon pikachu;
  pikachu.name = "Pikachu";
  pikachu.level = 100;
  rename_pokemon(pikachu, "Sparky");
  printf("%s\n", pikachu.name);
  return 0;
}
```

1. What is this code trying to do? What should the output or final state of pikachu be if everything worked as intended?

2. Why doesn't pikachu end up with the name "Sparky"? What happens to the value modified by `rename_pokemon`?

3. What does C do when you pass a struct to a function like this? Is it passed by reference or by value?

4. How could you fix this?

Output Parameters

Output Parameters

Definition: a pointer parameter used to store output in a location specified by the caller.

Useful for returning multiple items :)

Output Parameter Example

Consider the following function:

```
void getFive(int ret) {
  ret = 5;
}
```

Will the user get the value '5'?

Output Parameter Example

Consider the following function:

```
void getFive(int ret) {
  ret = 5;
}
```

Will the user get the value '5'?

No! You need to use a pointer so that the caller can see the change

```
void getFive(int* ret) {
  *ret = 5;
}
```

Exercise 4: Output Params

```
char *strcpy (char* dest, char* src) {




}
```

How is the caller able to see the changes in `dest` if C is pass-by-value?

Why do we need an output parameter? Why can't we just return an array we create in strcpy?

Exercise 4: Output Params

```
char *strcpy (char* dest, char* src) {
  char *ret_value = dest;
  while (*src != '\0') {
    *dest = *src;
    src++;
    dest++;
  }
  *dest = '\0'; // don't forget null
                // terminator!
  return ret_value;
}
```

How is the caller able to see the changes in `dest` if C is pass-by-value?

The caller can see the copied over string in `dest` since we are dereferencing `dest`. Note that modifications to dest that do not dereference will not be seen by the caller(such as `dest++`). Also note that if you used array syntax, then `dest[i]` is equivalent to `*(dest+i)`.

Why do we need an output parameter? Why can't we just return an array we create in strcpy?

If we allocate an array inside strcpy, it will be allocated on the stack. When we return an array, we are actually returning a pointer to the array, and not a copy of the array itself.Thus, we have no control over this memory after strcpy returns, which means we can't safely use the array whose address we've returned.

Header Files

What is a Header File?

- A place for:
  - function declarations
  - struct definitions
  - Constants and macros
  - Includes used in other files

Why use one?

- Better organization
- Reusability
- Prevents bugs (no accidental double inclusions!)

Example: Header Files

```c
#include <stdio.h>

void say_hello() {
  printf("Hello!\n");
}

int add(int a, int b) {
  return a + b;
}

int main() {
  say_hello();
  int result = add(3,4);
  printf("3 + 4 = %d\n",
                result);
  return 0;
}
```

How do we split this across multiple files?

Example: Header Files

```c
#include <stdio.h>

void say_hello() {
  printf("Hello!\n");
}

int add(int a, int b) {
  return a + b;
}

int main() {
  say_hello();
  int result = add(3,4);
  printf("3 + 4 = %d\n",
              result);
  return 0;
}
```

main.c

```c
#include <stdio.h>
#include "math_utils.h"

int main() {
  say_hello();
  int result = add(3,4);
  printf("3 + 4 = %d\n",
              result);
  return 0;
}
```

Example: Header Files

```c
#include <stdio.h>

void say_hello() {
  printf("Hello!\n");
}

int add(int a, int b) {
  return a + b;
}

int main() {
  say_hello();
  int result = add(3,4);
  printf("3 + 4 = %d\n",
              result);
  return 0;
}
```

main.c

```c
#include <stdio.h>
#include "math_utils.h"

int main() {
  say_hello();
  int result = add(3,4);
  printf("3 + 4 = %d\n",
              result);
  return 0;
}
```

Example: Header Files

```c
#include <stdio.h>

void say_hello() {
  printf("Hello!\n");
}

int add(int a, int b) {
  return a + b;
}

int main() {
  say_hello();
  int result = add(3,4);
  printf("3 + 4 = %d\n",
                result);
  return 0;
}
```

main.c

```c
#include <stdio.h>
#include "math_utils.h"

int main() {
  say_hello();
  int result = add(3,4);
  printf("3 + 4 = %d\n",
                result);
  return 0;
}
```

math_utils.h

```c
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);
void say_hello();

#endif // MATH_UTILS_H
```

math_utils.c

```c
#include <stdio.h>
#include "math_utils.h"

void say_hello() {
  printf("Hello!\n");
}

int add(int a, int b) {
  return a + b;
}
```

Example: Header Files

```c
#include <stdio.h>

void say_hello() {
  printf("Hello!\n");
}

int add(int a, int b) {
  return a + b;
}

int main() {
  say_hello();
  int result = add(3,4);
  printf("3 + 4 = %d\n",
                result);
  return 0;
}
```

Header guards →

Header guards →

math_utils.h

```c
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);
void say_hello();

#endif // MATH_UTILS_H
```

Header guards ensure that stuff within a header file is only defined once. Useful for preventing circular includes

math_utils.c

```c
#include <stdio.h>
#include "math_utils.h"

void say_hello() {
  printf("Hello!\n");
}

int add(int a, int b) {
  return a + b;
}
```

GDB (Debugger)

What is GDB?

- A scary program

GNU Project Debugger

- GNU = GNU's not Unix! (recursive acronym)
- You can see what's going on "inside" a program as it executes
- Supports Assembly, C, C++, D, Fortran, Go, Rust, etc
- Your best friend for this course!

Why not just use print statements?

- Print statements require knowing where the bug is
- Don't let you see state after a crash,
- It's hard to trace the call stack

What is GDB?

GNU (GNU's Not Unix) Project Debugger

- You can can examine the program as it is executing!
- Supports a variety of languages
  - (e.g. asm, C, C++, D, Fortran, Go, Rust, etc.)
- ***Your best friend for this course!***

"But can I use print statements?"

- Using print statement can be tedious.
- Requires the use of many to find where the bug is.
- Unable to examine the state of the program after it crashes.
- `printf("here\n");` can only take you so far.

Using GDB

Use `-g` or `-g3` flag in Makefile for compiling (the provided Makefile has this)

Type in your shell:

a) gdb pokemon_buggy

b) <enter>, then run

start	Start from beginning and stop there
run	Start and run program from beginning
continue	Run until program exits*
step	Run until next line*
bt	Shows call stack
b [fname:]function b [fname:]linenum	Sets breakpoint at beginning of function or at line
print var	Prints var

* = or until next breakpoint

Exercise 5: GDB

- Download files from website
- Run `make`, then debug via gdb

Environment Setup - Any Questions?

- Docker Setup
  - Installation
  - Container
- Git Repo Setup
  - SSH Key
  - Clone the repo
- VSCode
  - Installation
  - Extension
  - Entering the project