Penn Engineering
University of Pennsylvania

CIS 5480 Recitation 00

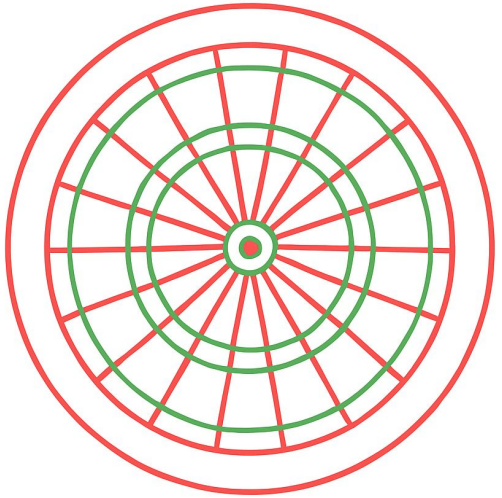
C Refresher,

2026-08-28

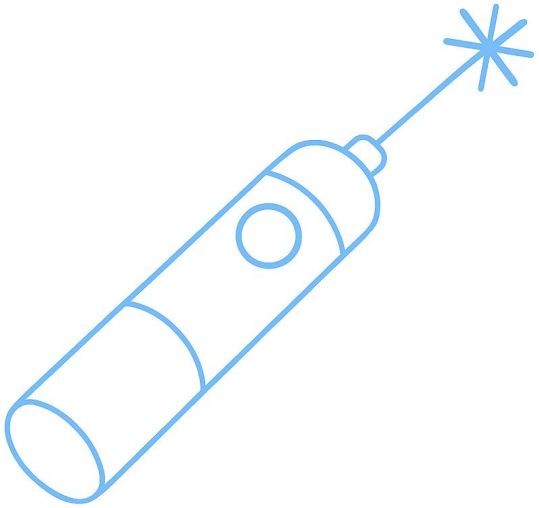
Site tour

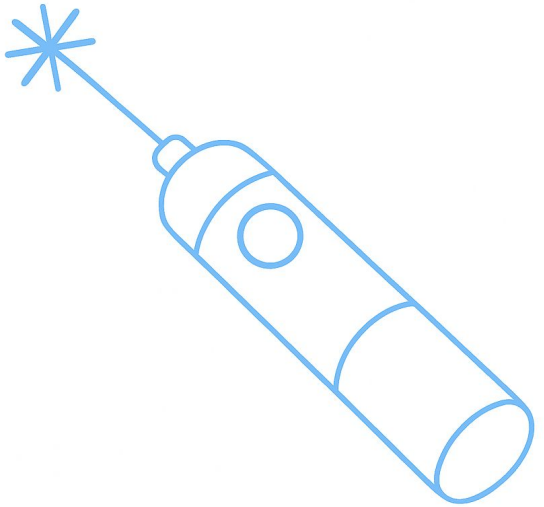
- https://www.engineering.upenn.edu/~cis5480/26fa/

Penn Engineering
University of Pennsylvania

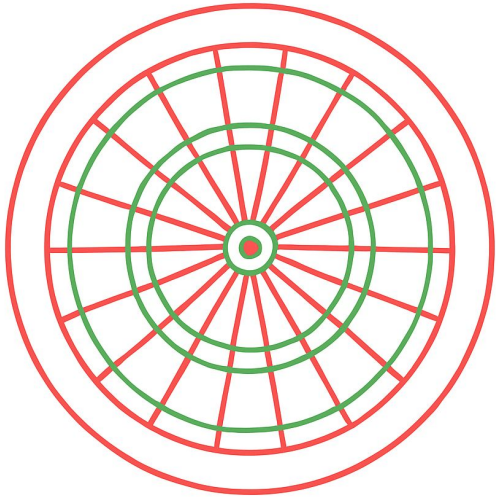


C Refresher



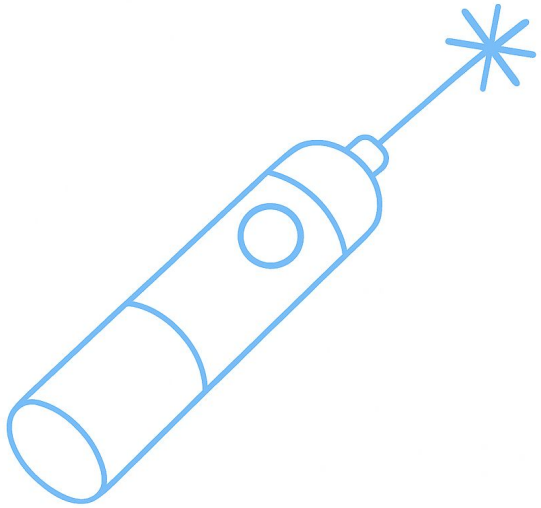


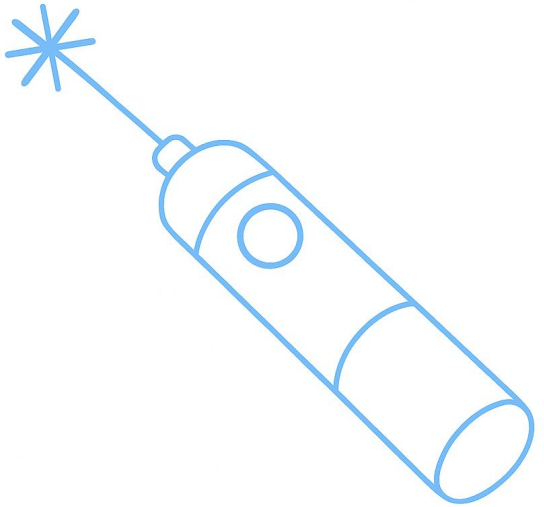
Penn Engineering
University of Pennsylvania



Pointer Review

👊👊👊

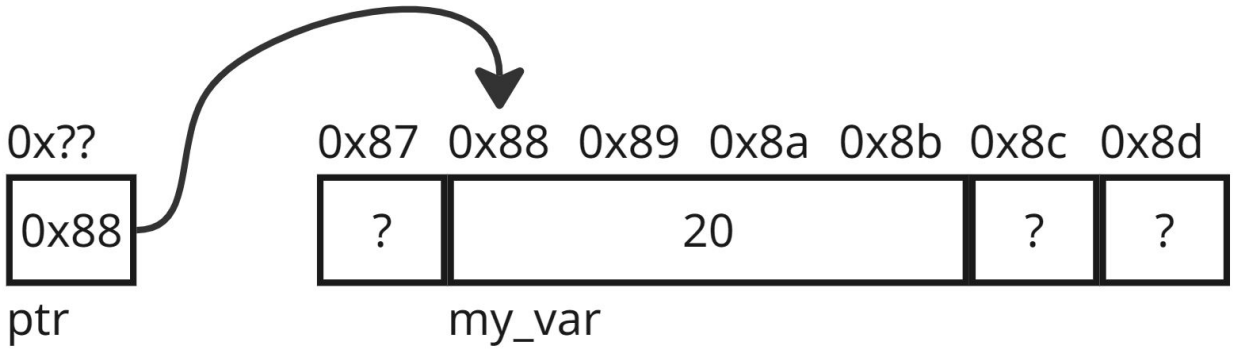




Pointers

- Primitive data type
- If you think of memory as a huge array of bytes, then a pointer is just an index
- They are also somewhere in memory

```
type* name;

int32_t my_var = 20;

int32_t* ptr = &my_var;
```



Pointer Syntax

Penn Engineering
University of Pennsylvania

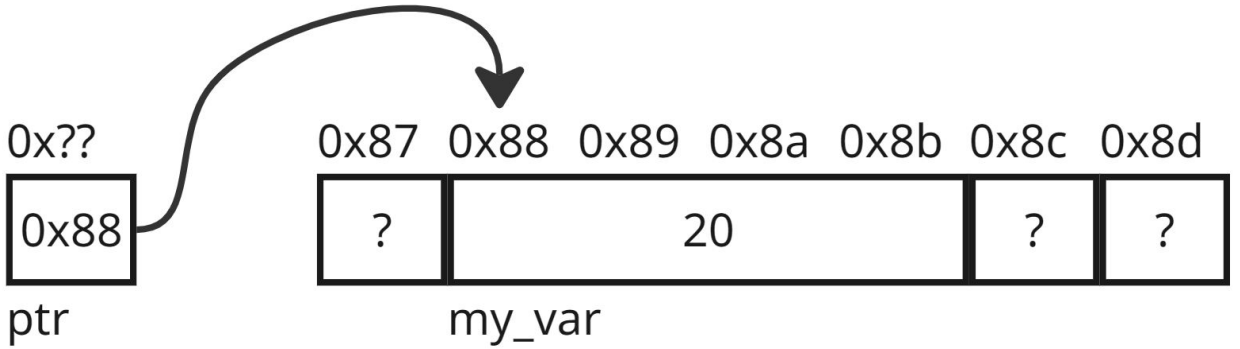
* in declaration → making it be a pointer

& in usage → getting the address of something

* in usage → accessing data at the address stored by the pointer

```c
int main(void) {
        int32_t my_var = 20;

        int32_t* ptr = &my_var;

        printf("%d\n", my_var);
        printf("%p\n", &my_var);
        printf("%p\n", ptr);
        printf("%d\n", *ptr);
}
```

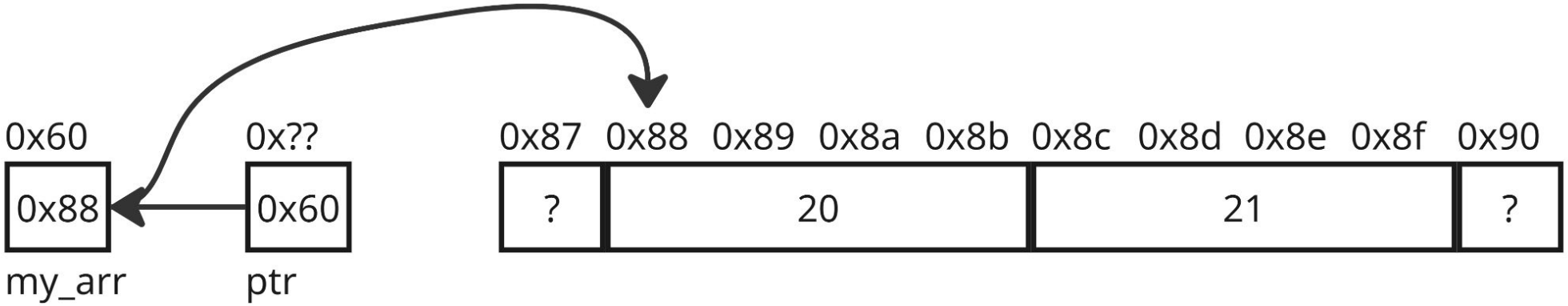
```
// 20
// 0x88
// 0x88
// 20
```



Pointer to a pointer?

- What if we wanted to change what my_arr points to?
- Let's first set it up (using **malloc**)

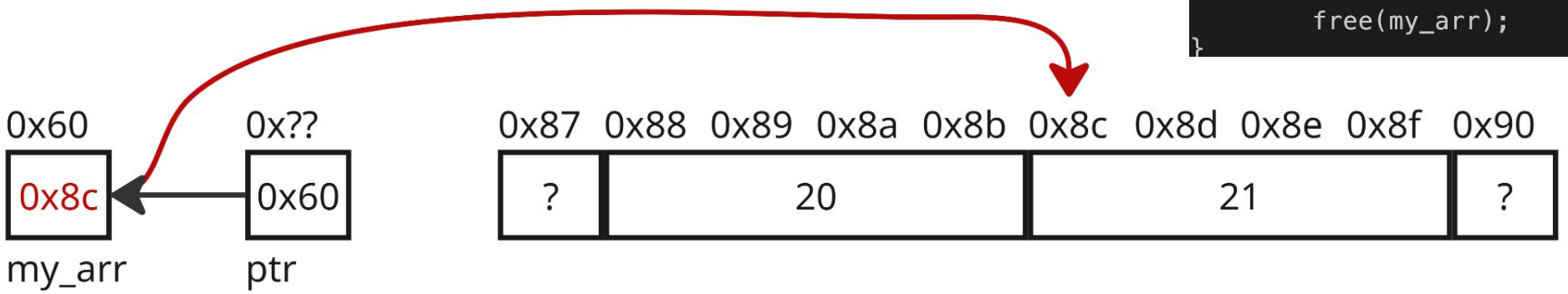
```
int main(void) {
        int32_t* my_arr = malloc(2 * sizeof(int32_t));
        my_arr[0] = 20;
        my_arr[1] = 21;

        int32_t** ptr = &my_arr;

        printf("%p\n", my_arr);   // 0x88
        printf("%d\n", *my_arr);  // 20
        printf("%p\n", ptr);      // 0x60
        printf("%p\n", *ptr);     // 0x88
        printf("%d\n", **ptr);    // 20

        free(my_arr);
}
```



Pointer to a pointer!

- We can modify the value in **my_arr** via ***ptr**
- We can then access the data as if it was the first element of the array, or by double-dereferencing **ptr**

```c
int main(void) {
        int32_t* my_arr = malloc(2 * sizeof(int32_t));
        my_arr[0] = 20;
        my_arr[1] = 21;

        int32_t** ptr = &my_arr;

        printf("%p\n", my_arr);   // 0x88
        printf("%d\n", *my_arr);  // 20
        printf("%p\n", ptr);      // 0x60
        printf("%p\n", *ptr);     // 0x88
        printf("%d\n", **ptr);    // 20

        *ptr += 1;

        printf("%p\n", my_arr);     // 0x8c
        printf("%d\n", *my_arr);    // 21
        printf("%d\n", my_arr[0]);  // 21
        printf("%p\n", ptr);        // 0x60
        printf("%p\n", *ptr);       // 0x8c
        printf("%d\n", **ptr);      // 21

        free(my_arr);
}
```

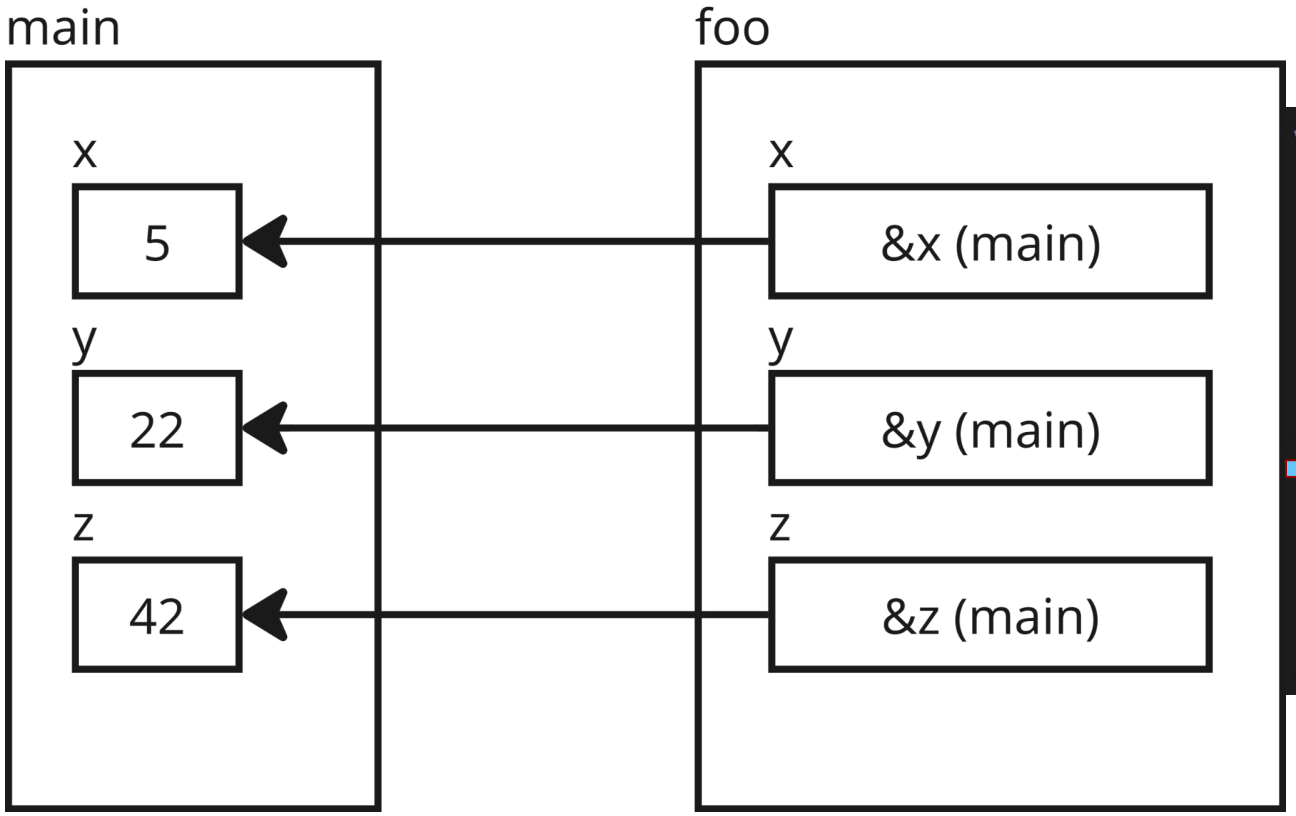


Your turn! Worksheet Q1

- Take some time to think about what this program does
- Draw a diagram and work it out

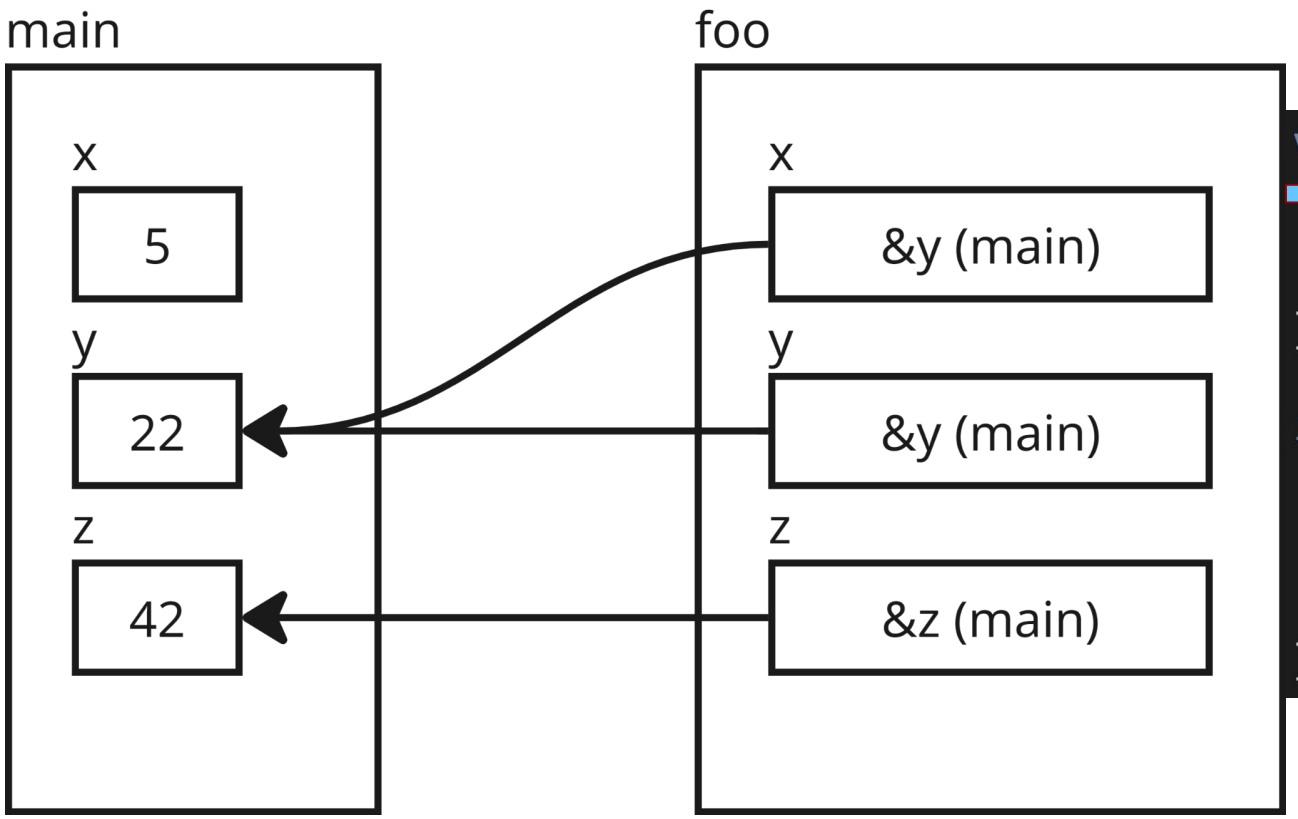
```
void foo(int32_t* x, int32_t* y, int32_t* z) {
        x = y;
        *x = *z;
        *z = 37;
}

int main(int argc, char** argv) {
        int32_t x = 5, y = 22, z = 42;
        foo(&x, &y, &z);
        printf("%d, %d, %d\n", x, y, z);
        return 0;
}
```

Diagram Justifications



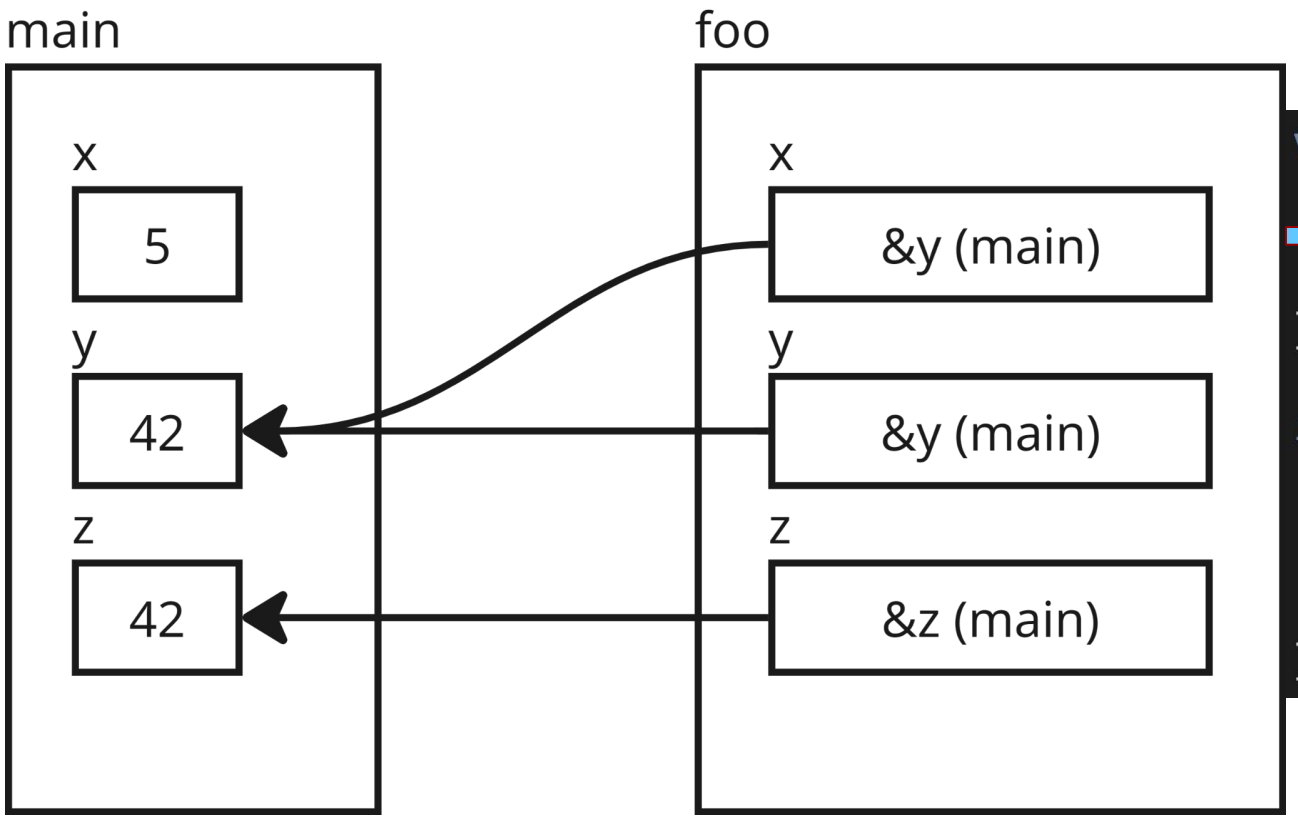

```c
void foo(int32_t* x, int32_t* y, int32_t* z) {
        x = y;
        *x = *z;
        *z = 37;
}

int main(int argc, char** argv) {
        int32_t x = 5, y = 22, z = 42;
        foo(&x, &y, &z);
        printf("%d, %d, %d\n", x, y, z);
        return 0;
}
```

Diagram Justifications



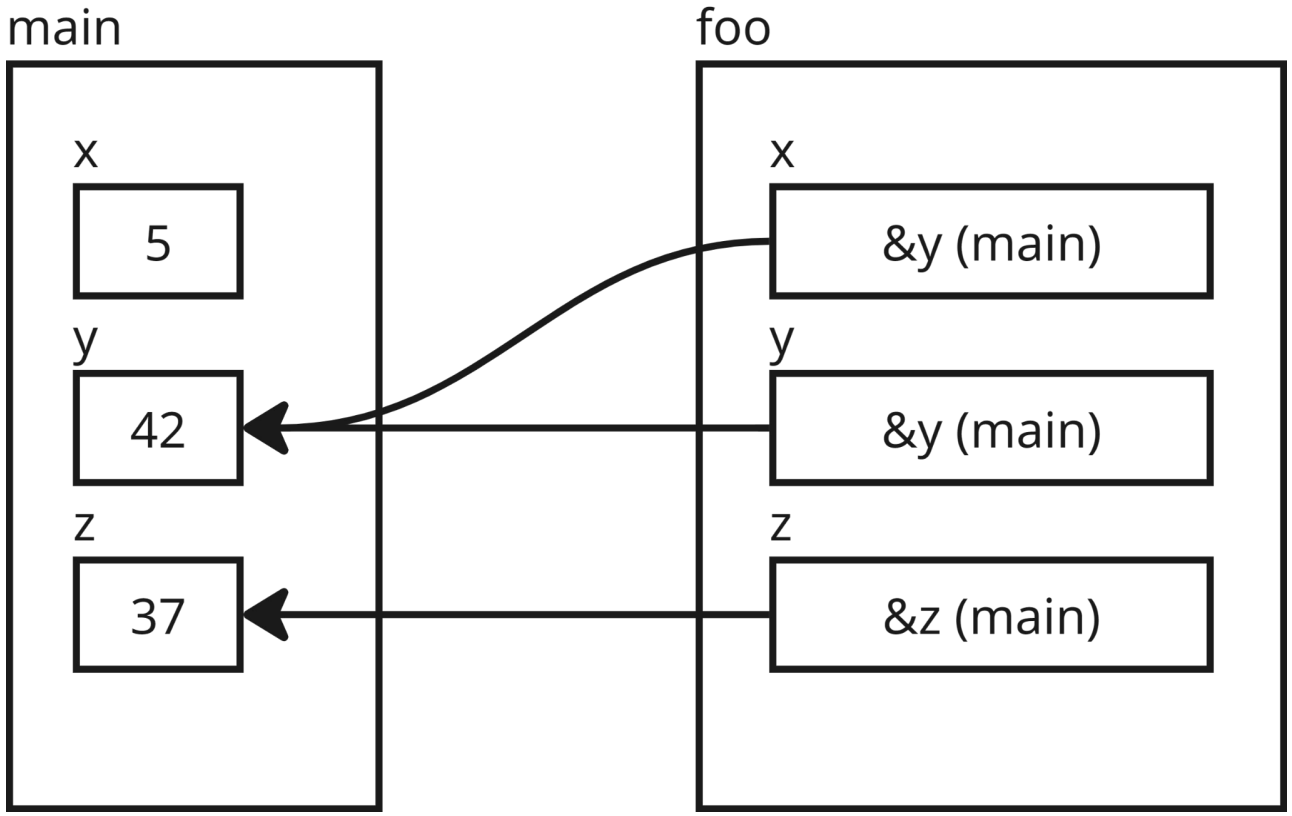

```
void foo(int32_t* x, int32_t* y, int32_t* z) {
        x = y;
        *x = *z;
        *z = 37;
}

int main(int argc, char** argv) {
        int32_t x = 5, y = 22, z = 42;
        foo(&x, &y, &z);
        printf("%d, %d, %d\n", x, y, z);
        return 0;
}
```

Diagram Justifications




```
void foo(int32_t* x, int32_t* y, int32_t* z) {
        x = y;
        *x = *z;
        *z = 37;
}

int main(int argc, char** argv) {
        int32_t x = 5, y = 22, z = 42;
        foo(&x, &y, &z);
        printf("%d, %d, %d\n", x, y, z);
        return 0;
}
```

Diagram Justifications



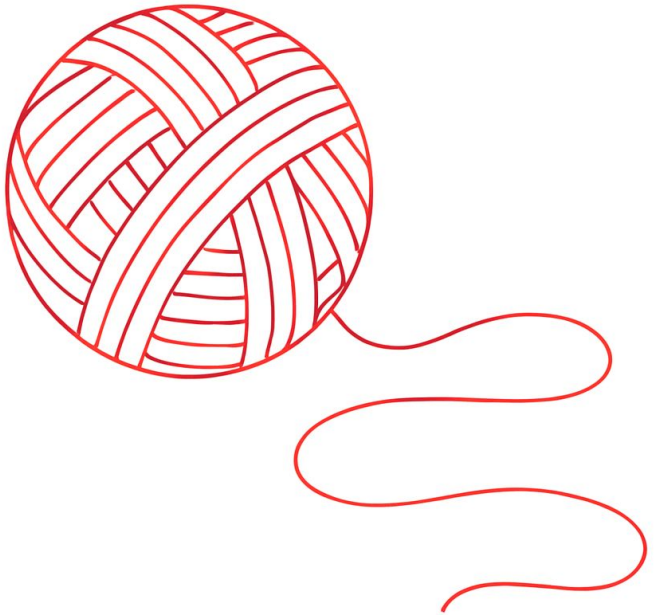

```
void foo(int32_t* x, int32_t* y, int32_t* z) {
        x = y;
        *x = *z;
        *z = 37;
}

int main(int argc, char** argv) {
        int32_t x = 5, y = 22, z = 42;
        foo(&x, &y, &z);
        printf("%d, %d, %d\n", x, y, z);
        return 0;
}
```

Your turn!

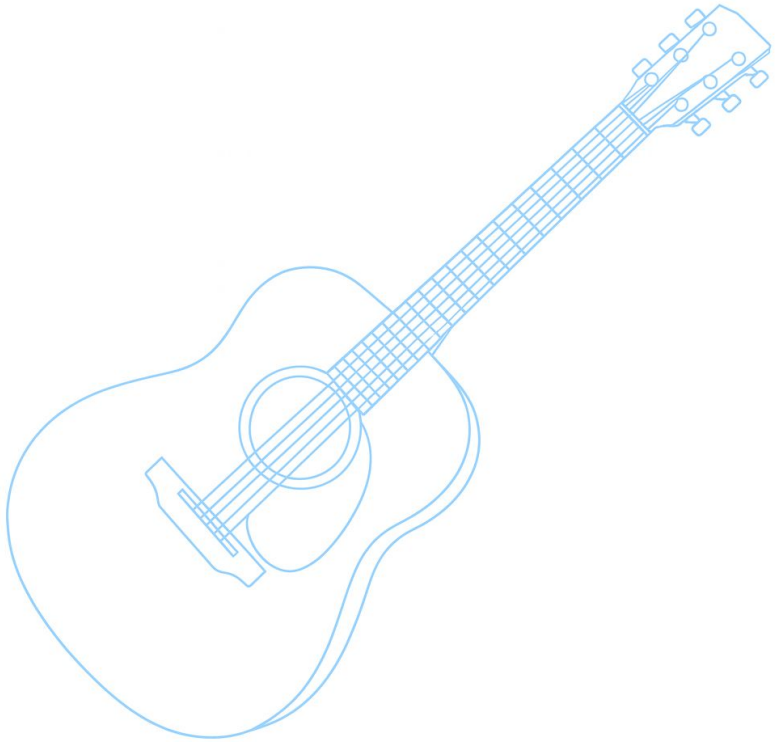
- Take some time to think about what this program does
- Draw a diagram and work it out
- **Ans: “5, 42, 37”**

```
void foo(int32_t* x, int32_t* y, int32_t* z) {
        x = y;
        *x = *z;
        *z = 37;
}

int main(int argc, char** argv) {
        int32_t x = 5, y = 22, z = 42;
        foo(&x, &y, &z);
        printf("%d, %d, %d\n", x, y, z);
        return 0;
}
```

Penn Engineering
University of Pennsylvania



Structs



Structs Initialization

- A `struct` is a C datatype that contains a set of fields
  - Acts similarly to primitive variables
- Generic declaration:

```
typedef struct point_st {
  float x;
  float y;
} point;

point pt;
point origin = {0.0f, 0.0f};
point other = (point) {
  .x = 3.14f,
  .y = 3.800f,
};

pt = origin; // pt now contains 0.0f, 0.0f
```

Default values are still garbage!

<- Initializer List

<- with designators

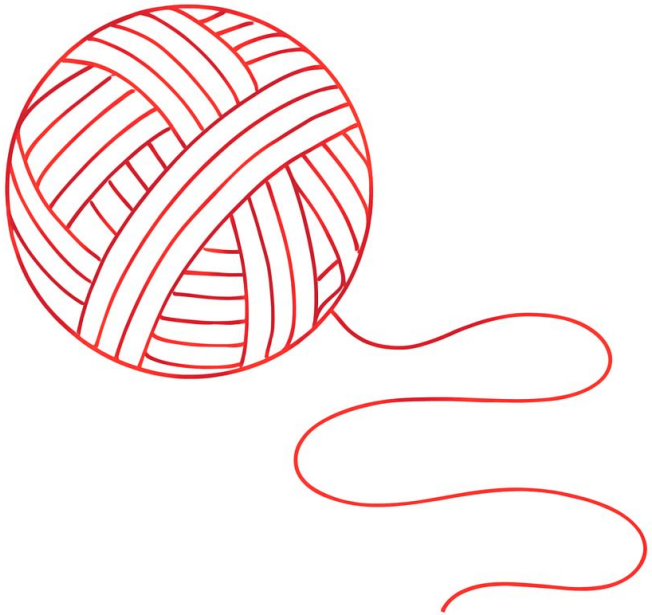
^ same as `pt.x = origin.x;` `pt.y = origin.y;`

Copying vs Referencing

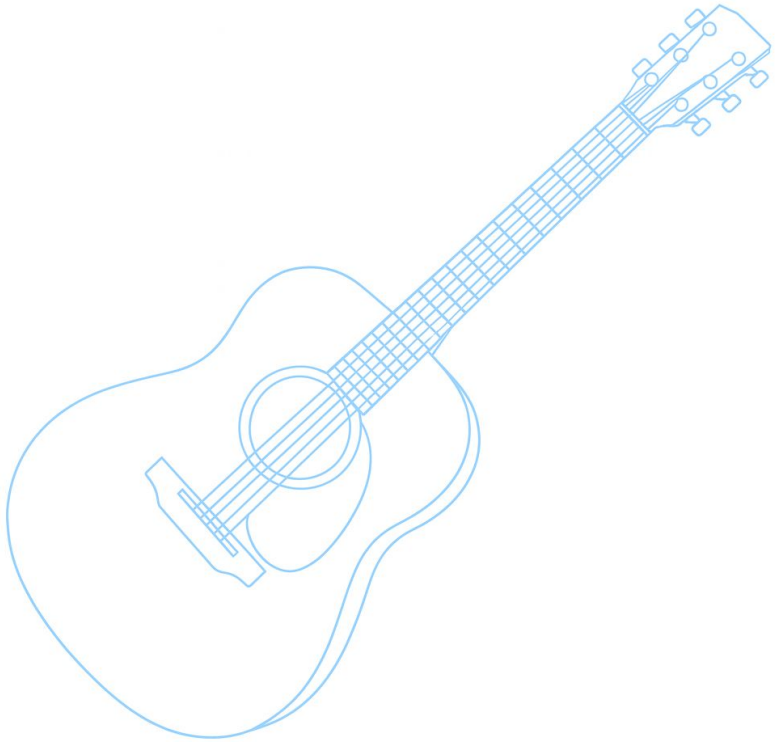
```c
typedef struct point_st {
  float x;
  float y;
} Point;

int main() {
  Point pt;
  Point origin = {0.0f, 0.0f};
  pt = origin; // pt now contains 0.0f, 0.0f

  pt.x = 3.0f;
  pt.y = 2.0f;
}
```

- Use “.” to refer to a field in a struct
- Use “->” to refer to a field from a struct pointer
  - Dereferences pointer first, then accesses field

Penn Engineering
University of Pennsylvania



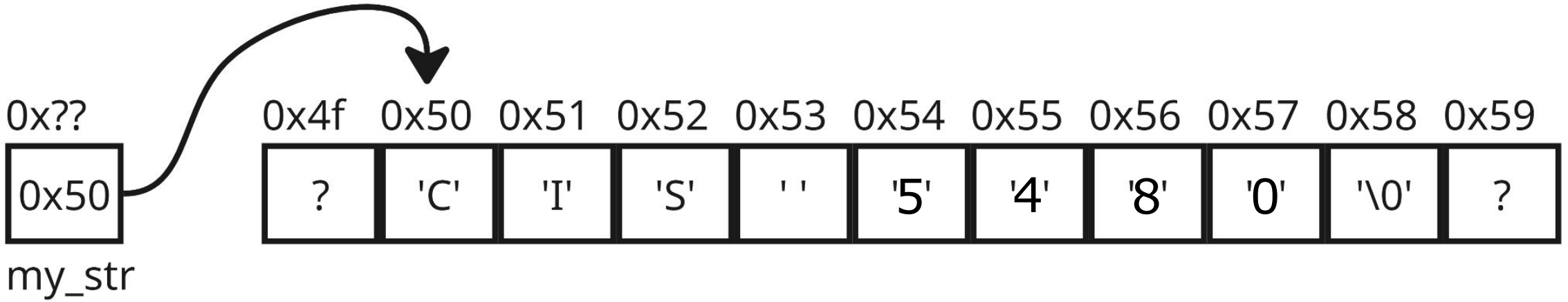
C Strings



C Strings

- They are null-terminated arrays of characters, meaning the '\0' character marks the end and must be present
- Remember to always allocate space for it!
- Segfault if you don’t (in the good case)

```
char my_str[9] = "CIS 5480";
```



Worksheet Q2 Re-assignment bug?

Penn Engineering
University of Pennsylvania

- Does this compile?
- Does this run? If yes, what outcome does it produce? If not, what error occurs?
- How should we change the code?

```c
void bar(char *str) {
  str = "ok bye!";
}

int main(int argc, char *argv[]) {
  char *str = "hello world!";
  bar(str);
  printf("%s\n", str);  // should print "ok bye!"
  return EXIT_SUCCESS;
}
```

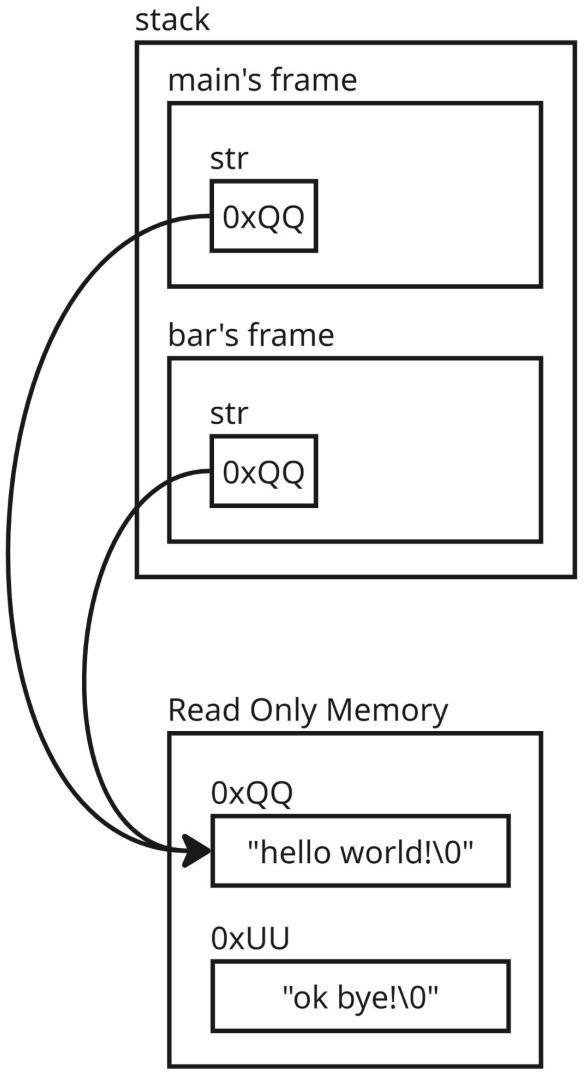
Worksheet Q2 Re-assignment bug?

- Does this compile?
  Yes
- Does this run? If yes, what outcome does it produce? If not, what error occurs?
  Yes, "hello world!"
- How should we change the code?
  Let's explain it first

```
void bar(char *str) {
  str = "ok bye!";
}

int main(int argc, char *argv[]) {
  char *str = "hello world!";
  bar(str);
  printf("%s\n", str);  // should print "ok bye!"
  return EXIT_SUCCESS;
}
```

Worksheet Q2 Re-assignment bug?

```c
void bar(char *str) {
  str = "ok bye!";
}

int main(int argc, char *argv[]) {
  char *str = "hello world!";
  bar(str);
  printf("%s\n", str);  // should print "ok bye!"
  return EXIT_SUCCESS;
}
```

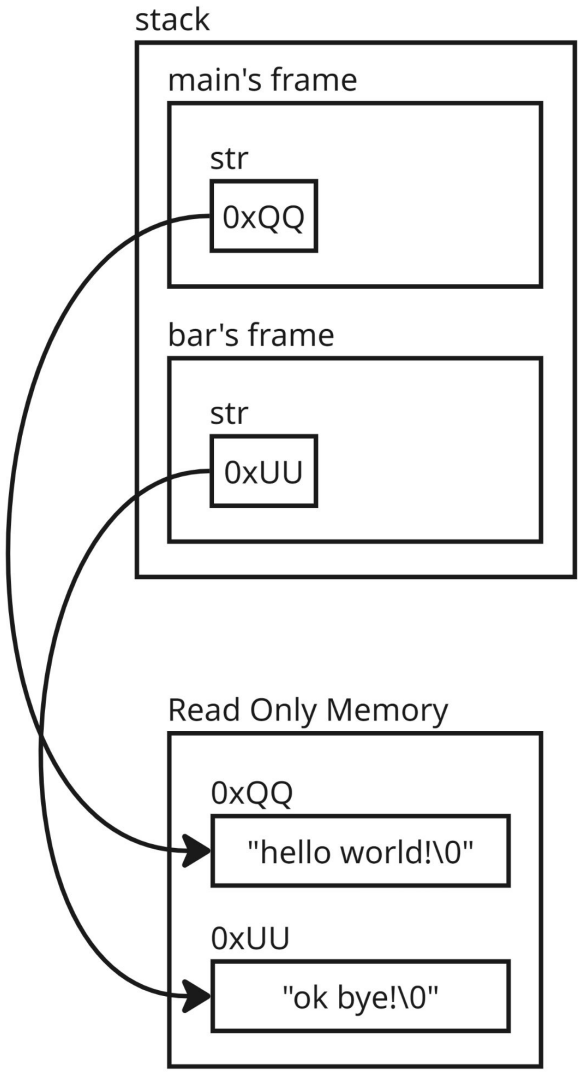


Note: 0xQQ and 0xUU are used here only to show that we're pointing to stuff in read only memory. This is not standard notation, don't think about it. What the arrows do is the important bit.

Worksheet Q2 Re-assignment bug?

```c
void bar(char *str) {
  str = "ok bye!";
}

int main(int argc, char *argv[]) {
  char *str = "hello world!";
  bar(str);
  printf("%s\n", str);  // should print "ok bye!"
  return EXIT_SUCCESS;
}
```

- How would we want the arrows to look at the end of bar?

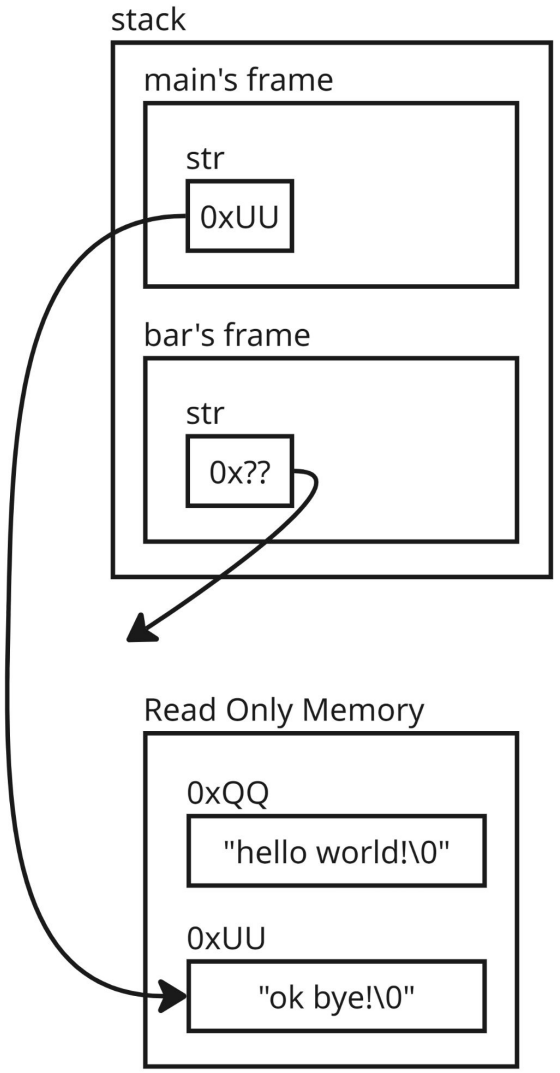


Worksheet Q2 Re-assignment bug?

Penn Engineering
University of Pennsylvania

```c
void bar(char *str) {
  str = "ok bye!";
}

int main(int argc, char *argv[]) {
  char *str = "hello world!";
  bar(str);
  printf("%s\n", str);  // should print "ok bye!"
  return EXIT_SUCCESS;
}
```

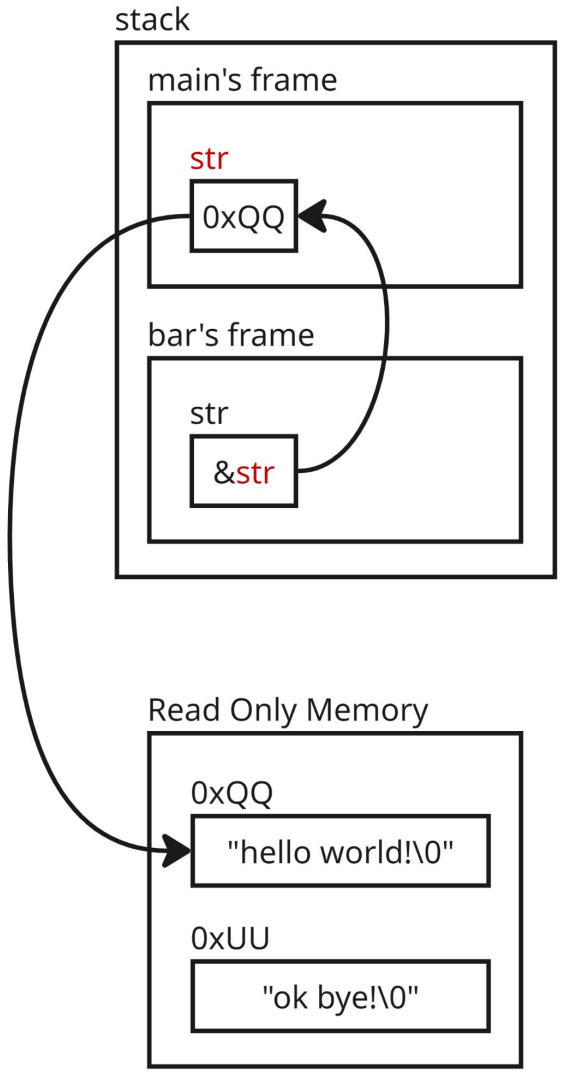
- In order to modify **main**'s **str** within **bar**, is the value of **str** enough?



Worksheet Q2 Re-assignment bug?

```c
void bar(char **str) {
  *str = "ok bye!";
}

int main(int argc, char *argv[]) {
  char *str = "hello world!";
  bar(&str);
  printf("%s\n", str);  // should print "ok bye!"
  return EXIT_SUCCESS;
}
```

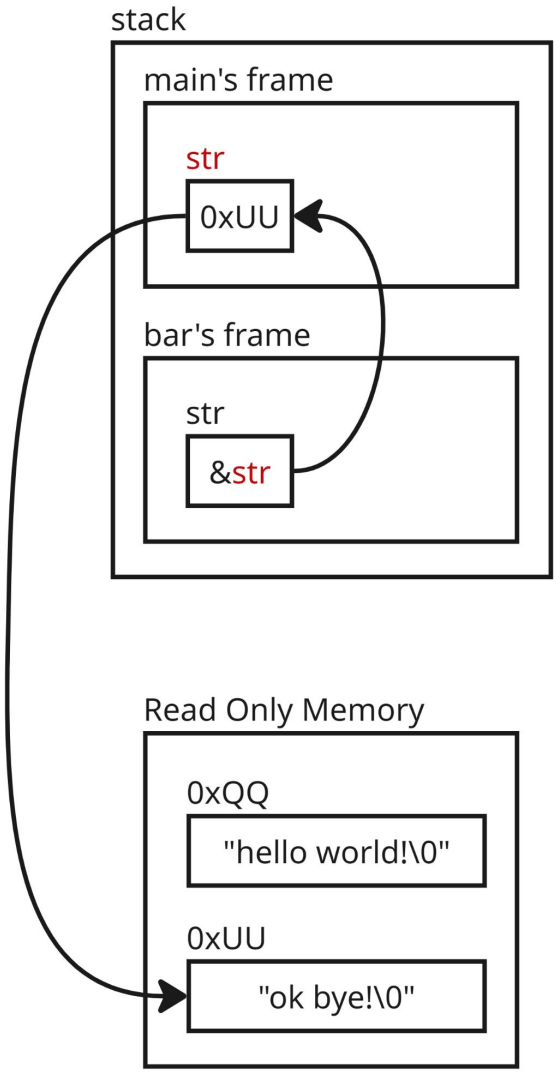
- We actually need the address of **main**'s **str**, so we can change the value stored by **main**'s **str**



Worksheet Q2 Re-assignment bug?

```c
void bar(char **str) {
  *str = "ok bye!";
}

int main(int argc, char *argv[]) {
  char *str = "hello world!";
  bar(&str);
  printf("%s\n", str);  // should print "ok bye!"
  return EXIT_SUCCESS;
}
```

- We actually need the address of **main**'s **str**, so we can change the value stored by **main**'s **str**



Output parameters

- A pointer parameter used to store output in a location specified by the caller
- Amazingly useful when a function needs to produce multiple results/ place results at a specific location

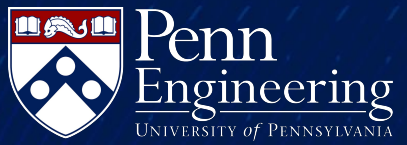
```
void bar(char **str) {
  *str = "ok bye!";
}

int main(int argc, char *argv[]) {
  char *str = "hello world!";
  bar(&str);
  printf("%s\n", str);  // should print "ok bye!"
  return EXIT_SUCCESS;
}
```

Output parameters

- **main**'s **str** is the pointer we need to change
- We give **bar** the address of **main**'s **str**
- **bar**'s **str** stores the address to **main**'s **str**, and is dereferenced in order to change **main**'s **str** from within **bar**

```c
void bar(char **str) {
  *str = "ok bye!";
}

int main(int argc, char *argv[]) {
  char *str = "hello world!";
  bar(&str);
  printf("%s\n", str);  // should print "ok bye!"
  return EXIT_SUCCESS;
}
```

Worksheet Q3 Output Parameters



- *strcpy* is a function from stdlib that copies a string *src* into an output parameter called *dest* and returns a pointer to the beginning of the destination string. Write the function below. You may assume that *dest* has sufficient space to store *src.*

```
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?

Worksheet Q3 Answers

```
// the strcpy() function copies the string src to dst (including the
// terminating '\0' character.)
char* strcpy(char* dst, const char* src) {
        if (dst == NULL || src == NULL)
                return NULL;

        size_t i = 0;
        while (src[i] != '\0') {
                dst[i] = src[i];
                ++i;
        }
        dst[i] = '\0';

        return dst;
}
```

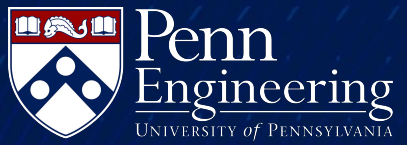
- How is the caller able to see the changes in *dest* if C is pass-by-value?

Although the dest pointer is passed-by-value it still points to the same memory as the caller's pointer → modifying dst[i] changes the underlying memory.

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

C cannot return arrays by value. Output parameters provide access to the caller's array by modifying the memory referenced by the copied pointer.

Worksheet Q4 Output Parameters



- Write a function to filter values out of a given input array.
- The function creates a new array allocated on the heap that is the same as the input array, except that all values that are equal to the input parameter “filter” are not in the array.

```
size_t filter(int* input, size_t length, int filter, int** out) {

}
```

Worksheet Q4 Answers

Penn Engineering
University of Pennsylvania

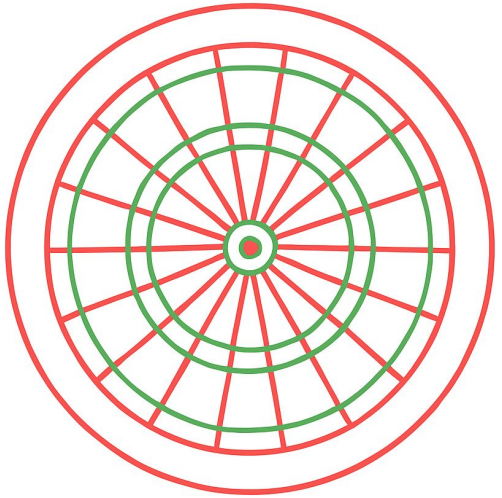
Two Passses

```c
// two pass approach (omitting malloc safety checks)
size_t filter(const int* input, size_t length, int filter, int** out) {
        if (input == NULL || out == NULL || length == 0)
                return 0;

        size_t filtered_len = 0;
        for (size_t i = 0; i < length; ++i) {
                if (input[i] != filter)
                        ++filtered_len;
        }

        if (filtered_len == 0)
                return 0;

        *out = malloc(sizeof(int) * filtered_len);

        size_t j = 0;
        for (size_t i = 0; i < length; ++i) {
                if (input[i] != filter)
                        (*out)[j++] = input[i];
        }

        return filtered_len;
}
```

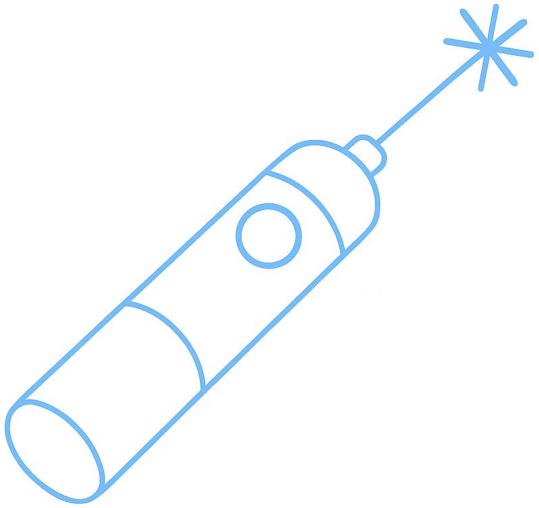
One Pass

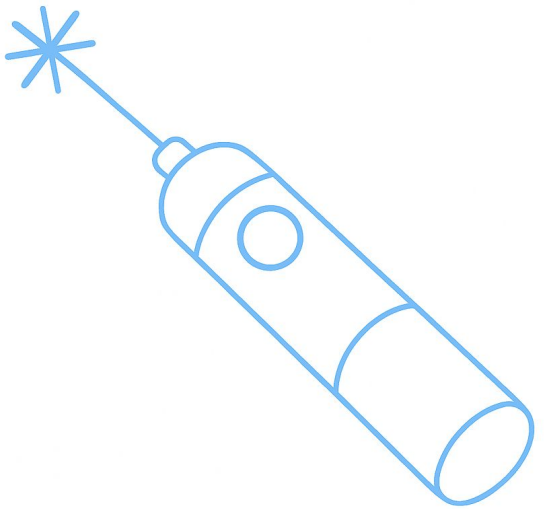
```c
// one pass approach (omitting malloc/realloc safety checks)
size_t filter(const int* input, size_t length, int filter, int** out) {
        if (input == NULL || out == NULL || length == 0)
                return 0;

        *out = malloc(sizeof(int) * length);

        size_t j = 0;
        for (size_t i = 0; i < length; ++i) {
                if (input[i] != filter)
                        (*out)[j++] = input[i];
        }

        if (j == 0) {
                free(*out);
                *out = NULL;
                return 0;
        }

        if (j < length) {
                *out = realloc(*out, sizeof(int) * j);
        }

        return j;
}
```

Penn Engineering
University of Pennsylvania



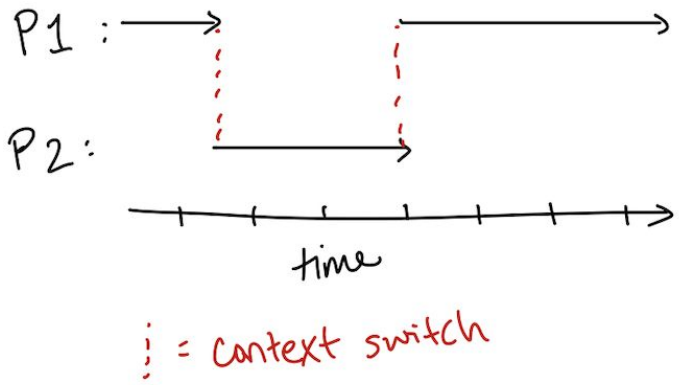
Processes





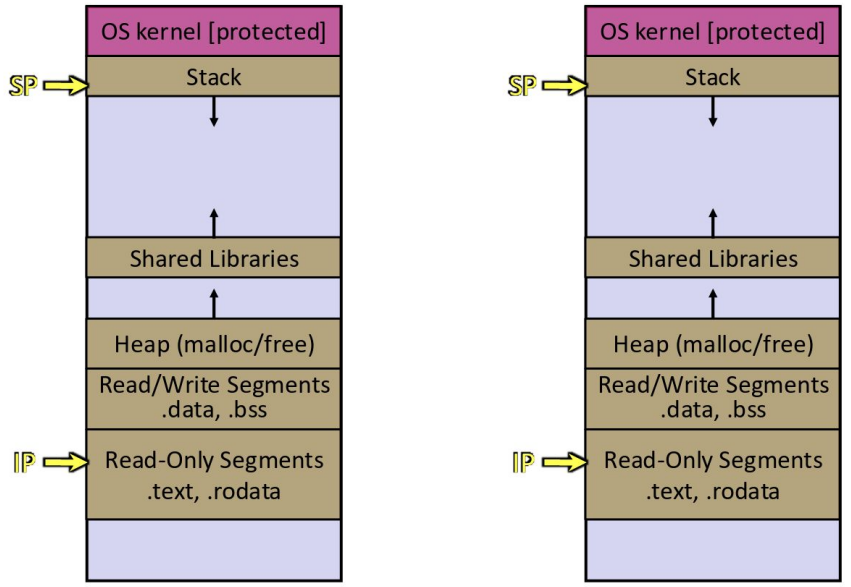
Processes

- Process: One instance of a running (or ready to run) program
- Two ways to visualize processes



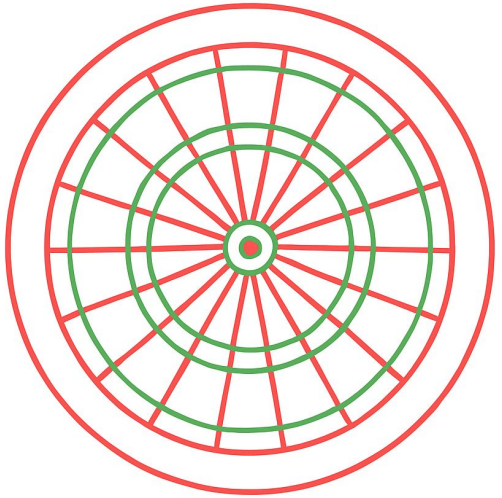
Processes as separate lines of execution

this visual will be more useful later in the semester

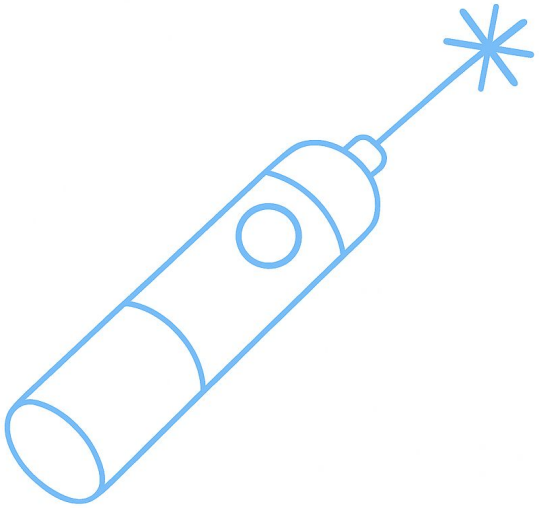


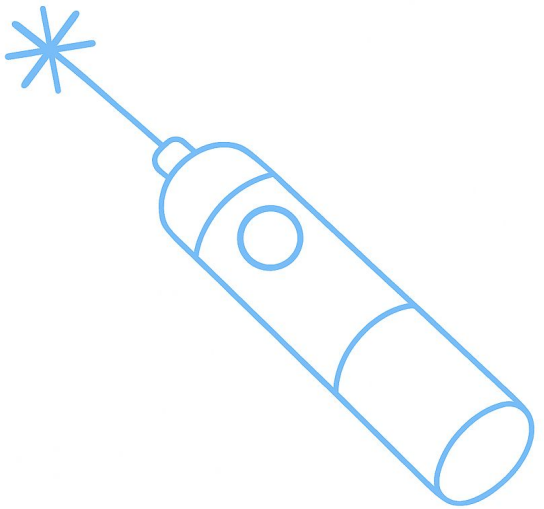
Processes as separate memory environments

Penn Engineering
University of Pennsylvania



Fork





Fork

- Clones the process that called fork()
  - Memory environment: stack, heap, read-only memory, registers, etc.
  - File descriptor table
  - Signal handlers & mask
- “The only function that returns twice”
- Generally invoked when we want to run a different program without terminating the current program
- Child starts running the line immediately following fork()

fork()

- **Clones the process that called fork()**
  - Memory environment: stack, heap, read-only memory, registers, etc.
  - other resources we haven't talked about yet :X

- The **only function that returns twice**: once in the parent and once in the child

- After fork(), there are **two separate processes** running the same code, both will resume running at the "return" of fork()

How fork() works:

Penn Engineering
University of Pennsylvania

1. Parent process calls fork(), creating a child process.
2. Return values of fork():
   a. In the child process, fork() returns 0.
   b. In the parent process, fork() returns the child's process ID (PID).
3. Both processes continue execution after fork() at the same point but run independently.

fork() Example

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    fork();
    fork();
    fork();
    printf("bingus\n");
    return 0;
}
```

How many times does “bingus” print?

fork() Example

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    fork();
    fork();
    fork();
    printf("bingus\n");
    return 0;
}
```

How many times does “bingus” print?

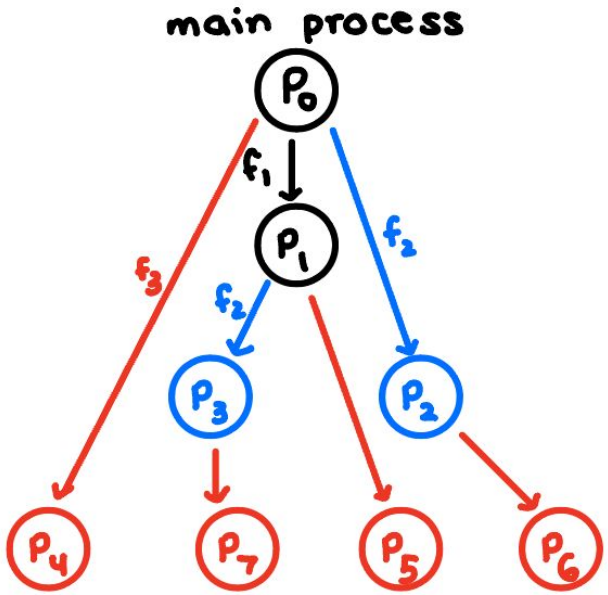
8 times

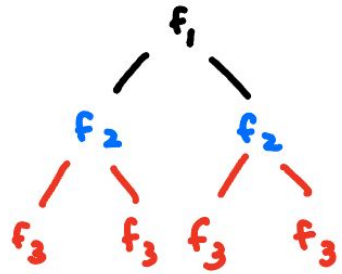
fork() Example

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    fork();
    fork();
    fork();
    printf("bingus\n");
    return 0;
}
```

How many times does “bingus” print?

8 times





Exercise 5

Consider the following program.

What are the possible outputs of this program?

```
int main(int argc, char** argv) {
        pid_t pid = fork();
        if (pid == 0) {
                pid = fork();
                if (pid == 0)
                        printf("my\n");
                else
                        printf("mind\n");
                exit(0);
        }

        pid = fork();
        if (pid == 0) {
                printf("skies\n");
                exit(0);
        }
        printf("of\n");
}
```

Exercise 5

Penn Engineering
University of Pennsylvania

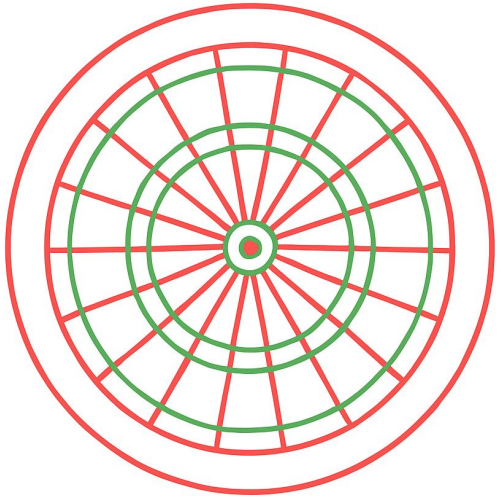
Each of the print statements can be printed at any order.

4! possible orderings.

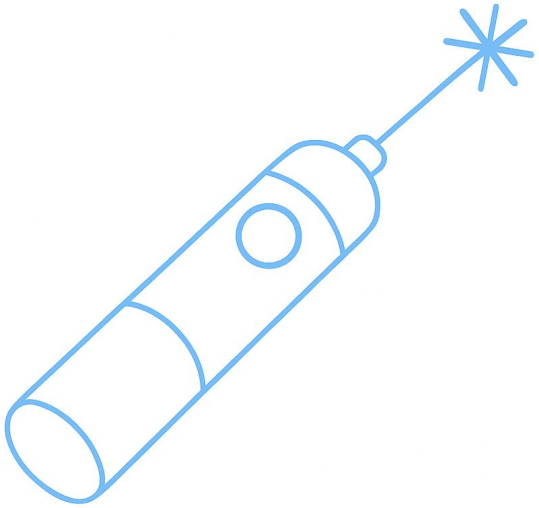
24 potential outputs.

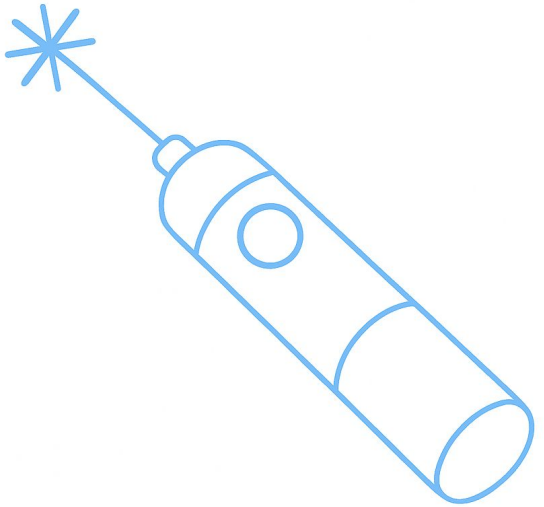
```
int main(int argc, char** argv) {
        pid_t pid = fork();
        if (pid == 0) {
                pid = fork();
                if (pid == 0)
                        printf("my\n");
                else
                        printf("mind\n");
                exit(0);
        }

        pid = fork();
        if (pid == 0) {
                printf("skies\n");
                exit(0);
        }
        printf("of\n");
}
```

Penn Engineering
University of Pennsylvania



Extra Time for Setup Help





Exec(ve)

- Replaces the current process with another

```
execve(char *pathname, char *argv[],
       char*envp[]);
```

  - `pathname` = string containing path to binary file to be executed
  - `argv` = array of strings containing arguments to run the next program
    - Argv[0] == pathname
  - `envp` = list of environment variables
    - Just set this parameter to `NULL`
- What's replaced?
- What's unchanged?

Comprehensive Question Pt. 1

Penn Engineering
University of Pennsylvania

Which process(es) have access to `file.txt`?

A. Parent
B. Child
C. Both
D. Neither

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

int main() {
  pid_t child = fork();
  int fd = open("file.txt", O_WRONLY);
  if (fd == -1) {
    exit(EXIT_FAILURE);
  }
  write(fd, "this is parent or child.", 25);
  close(fd);
  return 0;
}
```

Comprehensive Question Pt. 2

Penn Engineering
University of Pennsylvania

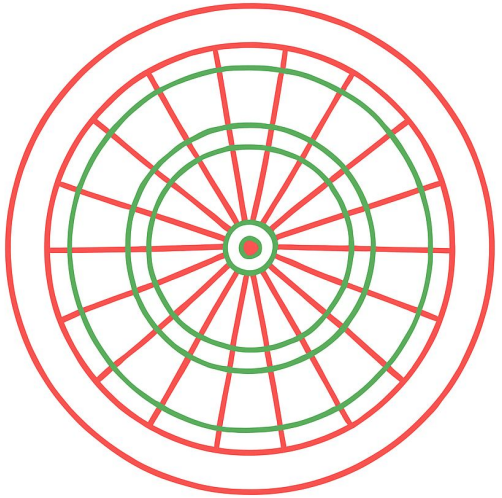
If the parent closes the file, can the child still write to `file.txt`?

Explain you answer.

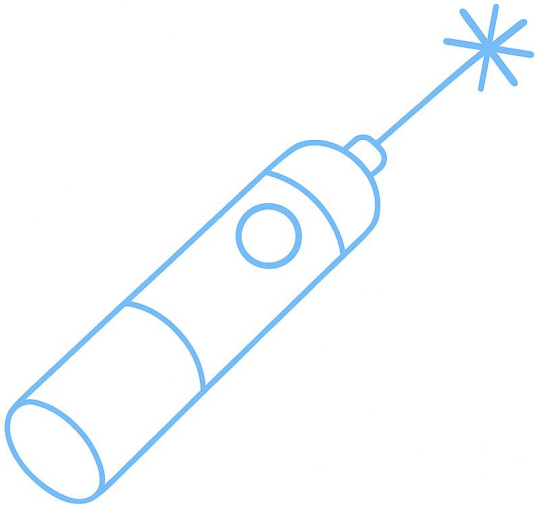
Yes, `open()` was called after fork. Processes have independent memory, but share the same computer (and file system)

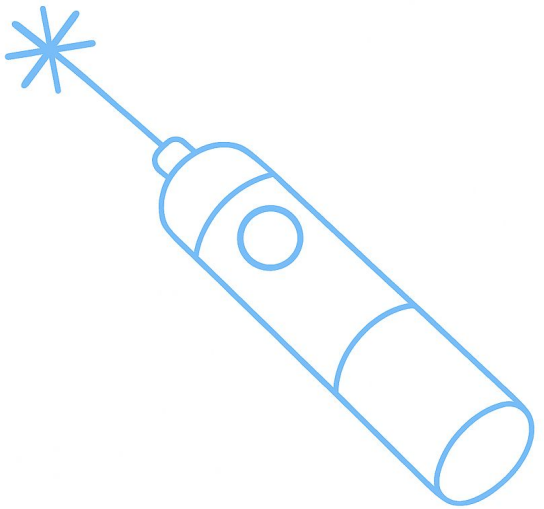
```c
#include <fcntl.h>
#include <stdlib.h>

int main() {
  pid_t child = fork();
  int fd = open("file.txt", O_WRONLY);
  if (fd == -1) {
    exit(EXIT_FAILURE);
  }
  write(fd, "this is parent or child.", 25);
  close(fd);
  return 0;
}
```

Penn Engineering
University of Pennsylvania



Man Pages & GCC





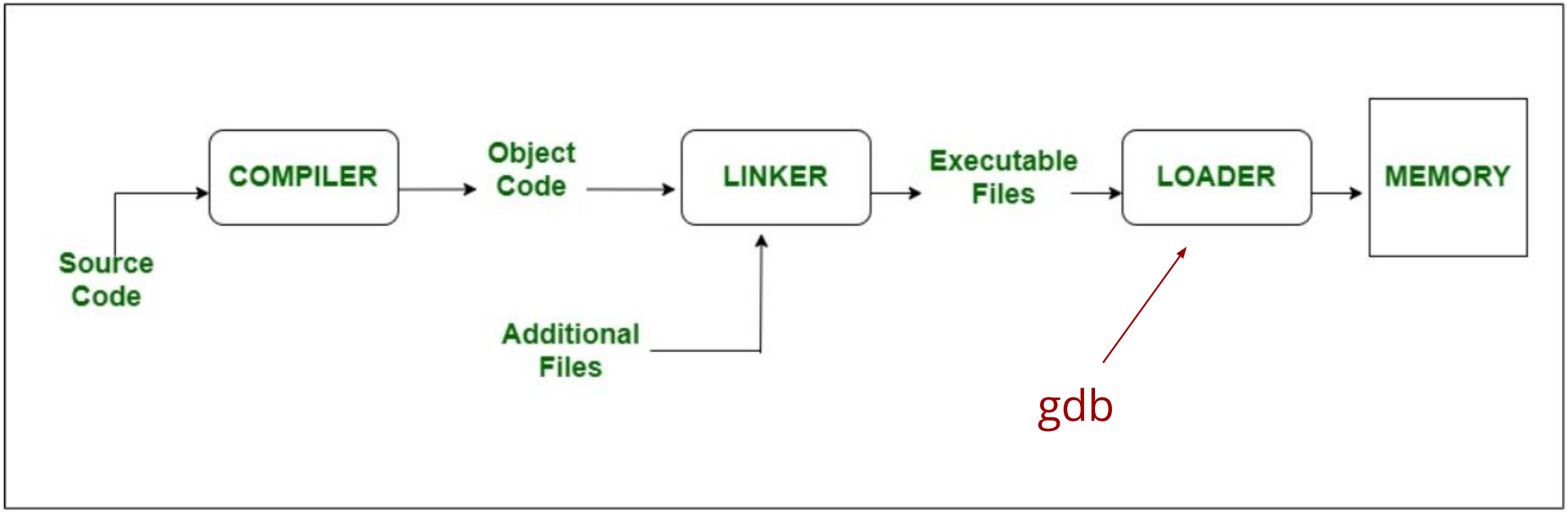
The Man Pages

Penn Engineering
University of Pennsylvania

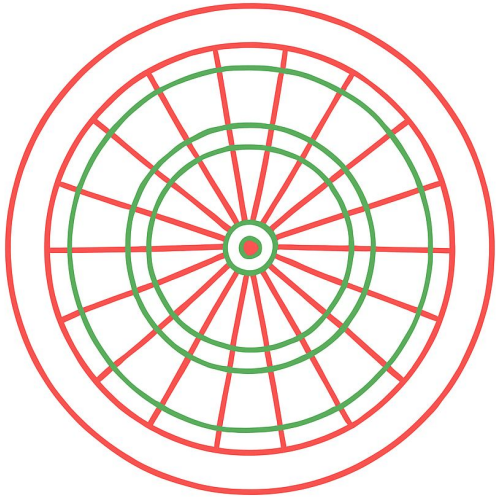
- Type **man read** in the terminal. Do you see what you expect?
- What do you need to type in order to see the *correct* man page for the C function for **read**?
- What about for **printf**? Even if its a function in C, does it have the same section as **read**? Why or why not?

How do Programs Run? (gcc)

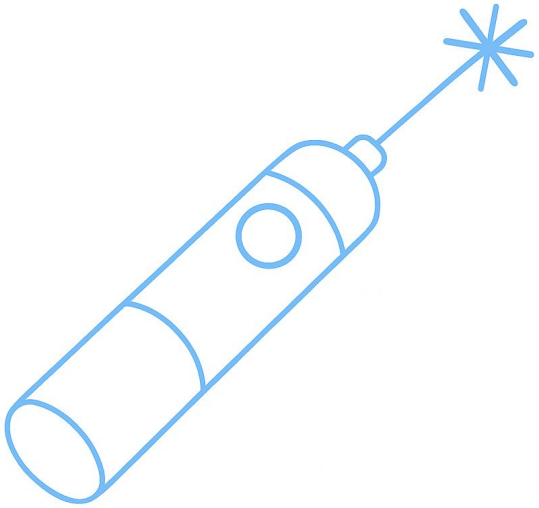
- gcc (GNU Compiler Collection)
  - We use makefiles, but these rely on gcc!
  - For more: https://web.stanford.edu/class/archive/cs/cs107/cs107.1202/resources/gcc

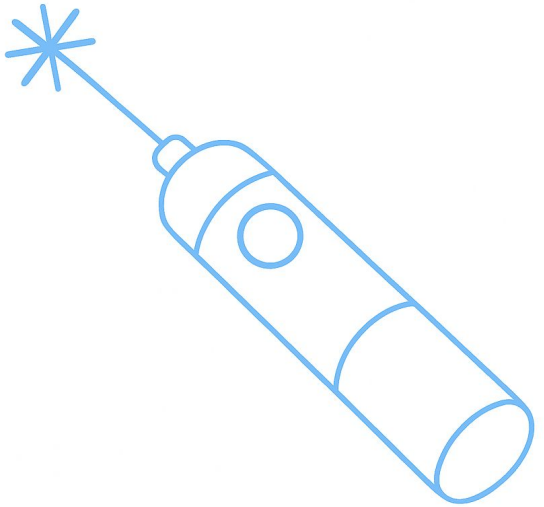


Penn Engineering
University of Pennsylvania



GDB (Debugger)





What is GDB?

Penn Engineering
University of Pennsylvania

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:

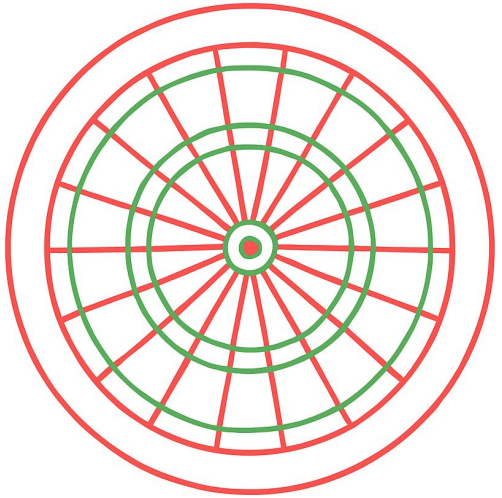
a) **`gdb pokemon_buggy`**

b) <enter>, then **`run`**

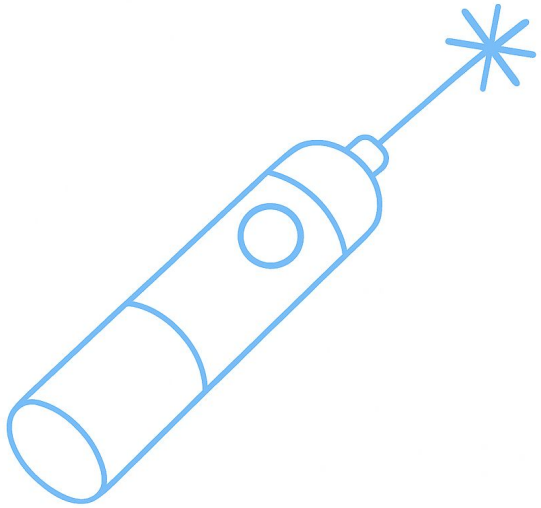
start	Start from beginning and stop there
run	Start and run program from beginning
continue	Run until program exits*
step	Run until next line*
bt	Shows call stack
b [fname:]function b [fname:]linenum	Sets breakpoint at beginning of function or at line
print var	Prints var

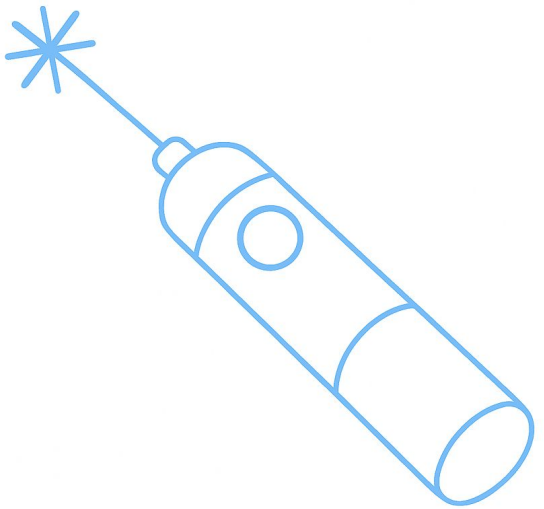
* = or until next breakpoint

Penn Engineering
University of Pennsylvania



Valgrind





Valgrind

Penn Engineering
University of Pennsylvania

- Your handy debugging tool for memory mismanagement
- It runs your program, and looks for any memory errors during execution
- It will only catch errors it encounters in runtime! Pay attention to **code coverage** - ensure all* lines of code are run in a valgrind session

*or at the very least, the most critical lines

Valgrind Errors

Penn Engineering
University of Pennsylvania

- Memory leaks: memory that hasn't been freed by the time the program exits
- Invalid read/write: accessing unallocated (or deallocated) memory
- Uninitialized bytes: Using memory that was allocated but never had any values put into them

How to Run Valgrind

Penn Engineering
University of Pennsylvania

- Can run in terminal: **`valgrind ./program <program arguments>`**
- Useful valgrind arguments (put between `valgrind` and `./program`)
  - **`--trace-children=<yes|no>`** (default: no)
  - **`--track-origins=<yes|no>`** (default: no)
  - **`--leak-check=<no|summary|yes|full>`** (default: summary)
    - `full` option gives you the most info