Griffin Hurt
Undergraduate Teaching Fellow
griffhurt@pitt.edu
https://griffinhurt.com

Spring 2024, Term 2244
Friday 2 PM Recitation
Mar 22nd, 2024

Slides adapted from
Shinwoo Kim, Martha Dixon, and Vinicius Petrucci

Department of Computer Science
School of Computing & Information
University of Pittsburgh

Recitation 8: Fork/Dynamic Loading/Signals

- Course News
- Function Pointers
- Quiz
- Signals
- Fork/Exec

Course News

Loading and Forking Lab is due Thursday, March 28th at 5:59 PM

Bomb Project is due Monday, March 25th at 5:59PM

Function Pointers

- Don’t be scared!
- We’ve covered pointers before
  - Only this time, we’ll be able to point to the memory address of a function…
- We should be familiar with this idea from the assembly project… where we used a function’s address to change execution of a program
- So… much like variables… functions have addresses too!
  - That we can point to!

How Can We Declare Function Pointers?

```
return_type (*pointer_name)(list,of,argument,types);
```

- Let’s dissect this
    - First, we’ll start by declaring the functions return type
        - void, int, long, double char*, etc.
    - Next, the actual pointer variable to the function
        - *pointer_name is your choice of variable name, not the actual function name
    - Lastly, a list of argument types
        - So we know what kind of arguments we need to pass when dereferencing our function pointer
        - (int, int), (double, int), (int*, char*), etc.

How Can We Declare Function Pointers?

- How can we call the functions we point to?
- Well, in one of two ways
    - If we are explicitly declaring a function pointer to one of our written functions:

```
type (*ptr1)(int, int) = &fun1;
type (*ptr1)(int, int) = fun1;
```

Function Pointer Declarations - Option 1

```
type (*ptr1)(int, int) = &fun1;
```

- Here, we’re declaring our function pointer which will point to an address of a function
  - Address means we need to dereference
  - Very similar to an int pointer
    - To access it, we need to use the dereference operator (*)
- So, when actually calling the function, we can say:

```
(*ptr1)(num1, num2);
```

Function Pointer Declarations - Option 2

```
type (*ptr1)(int, int) = fun1;
```

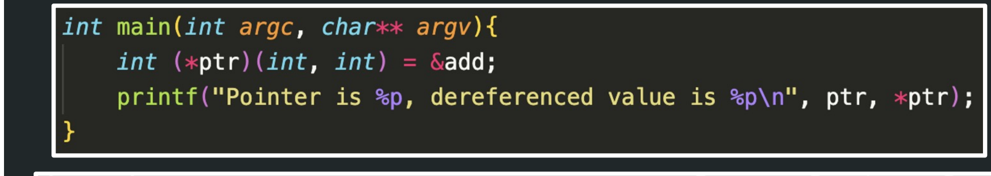
- Here, we’re declaring our function pointer which will point to an address of a function
- However, since we’re not using the (&) operator, we don’t need to dereference first
- Thus, our function pointer call is:

```
ptr1(num1, num2);
```

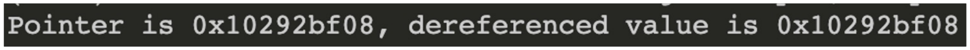
- It looks identical to just calling fun1(num1, num2)

What’s the Difference?

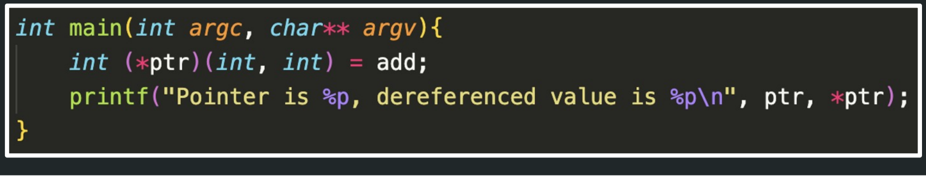
- Well… we might say the difference is in the address… but that turns out to not be quite the truth
- Consider these two snippets



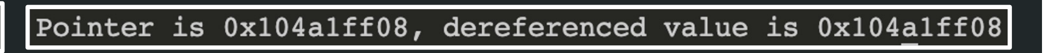
```
int main(int argc, char** argv){
    int (*ptr)(int, int) = &add;
    printf("Pointer is %p, dereferenced value is %p\n", ptr, *ptr);
}
```



```
Pointer is 0x10292bf08, dereferenced value is 0x10292bf08
```



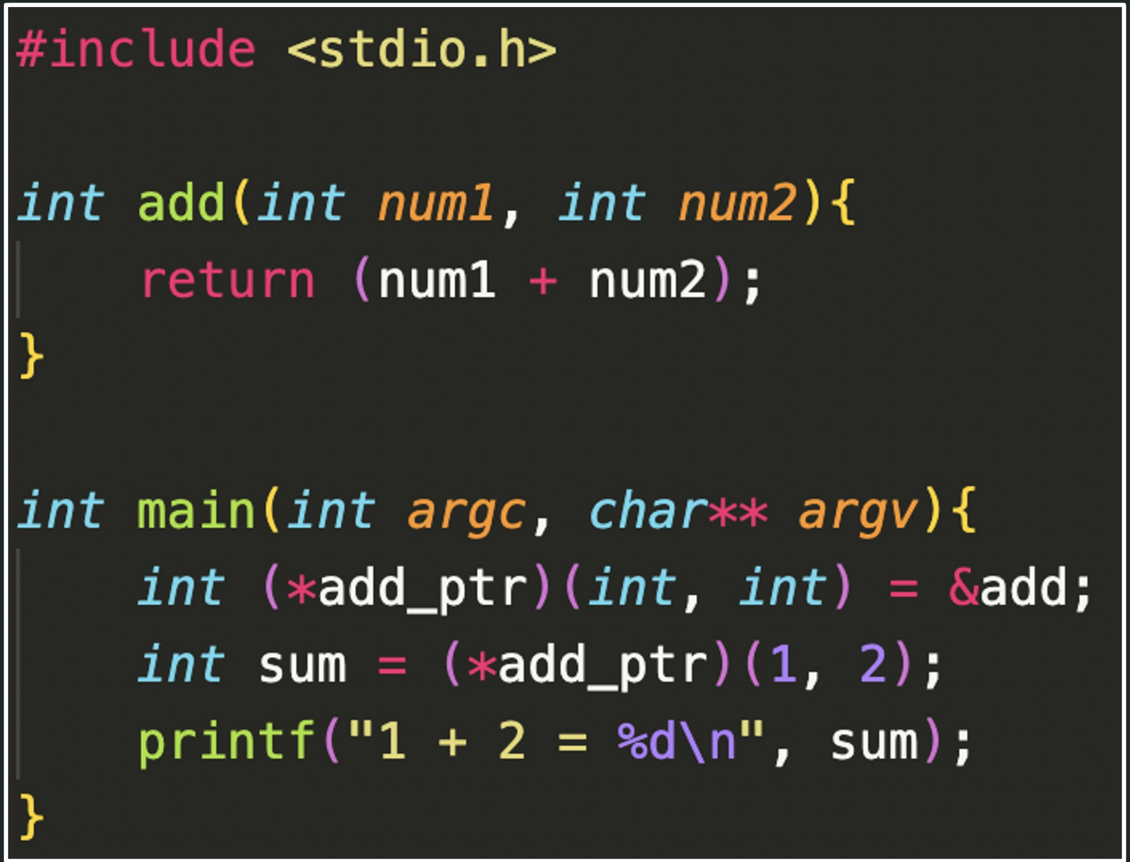
```
int main(int argc, char** argv){
    int (*ptr)(int, int) = add;
    printf("Pointer is %p, dereferenced value is %p\n", ptr, *ptr);
}
```



```
Pointer is 0x104a1ff08, dereferenced value is 0x104a1ff08
```

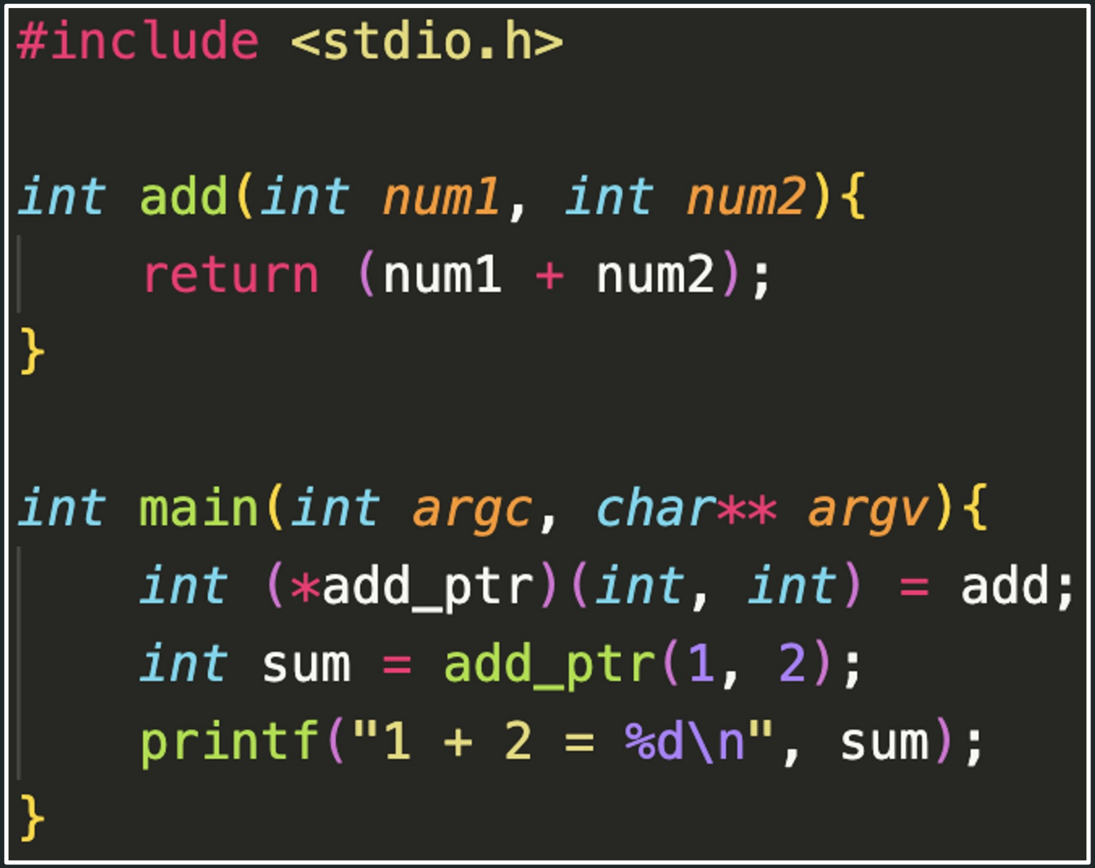
- The pointer and it’s value are the same…

Option 1 - Example



```c
#include <stdio.h>

int add(int num1, int num2){
    return (num1 + num2);
}

int main(int argc, char** argv){
    int (*add_ptr)(int, int) = &add;
    int sum = (*add_ptr)(1, 2);
    printf("1 + 2 = %d\n", sum);
}
```

Option 2 - Example

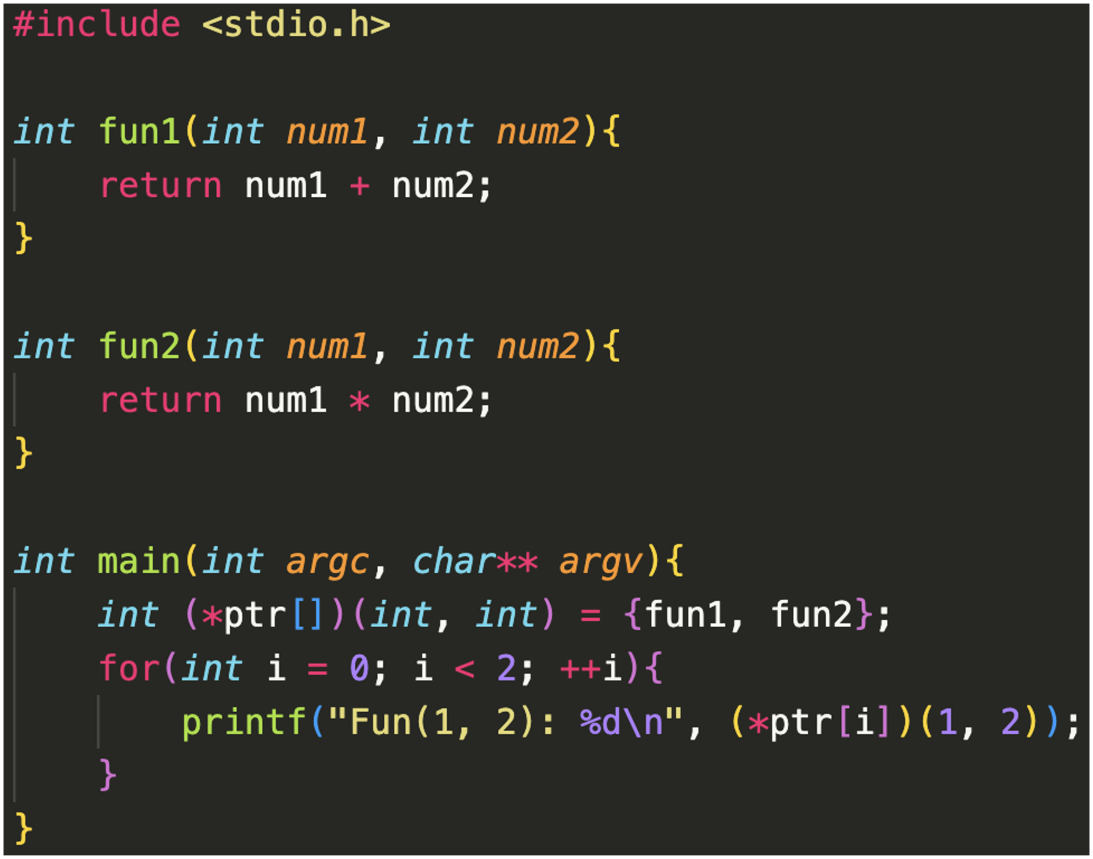


```c
#include <stdio.h>

int add(int num1, int num2){
    return (num1 + num2);
}

int main(int argc, char** argv){
    int (*add_ptr)(int, int) = add;
    int sum = add_ptr(1, 2);
    printf("1 + 2 = %d\n", sum);
}
```

Something Familiar

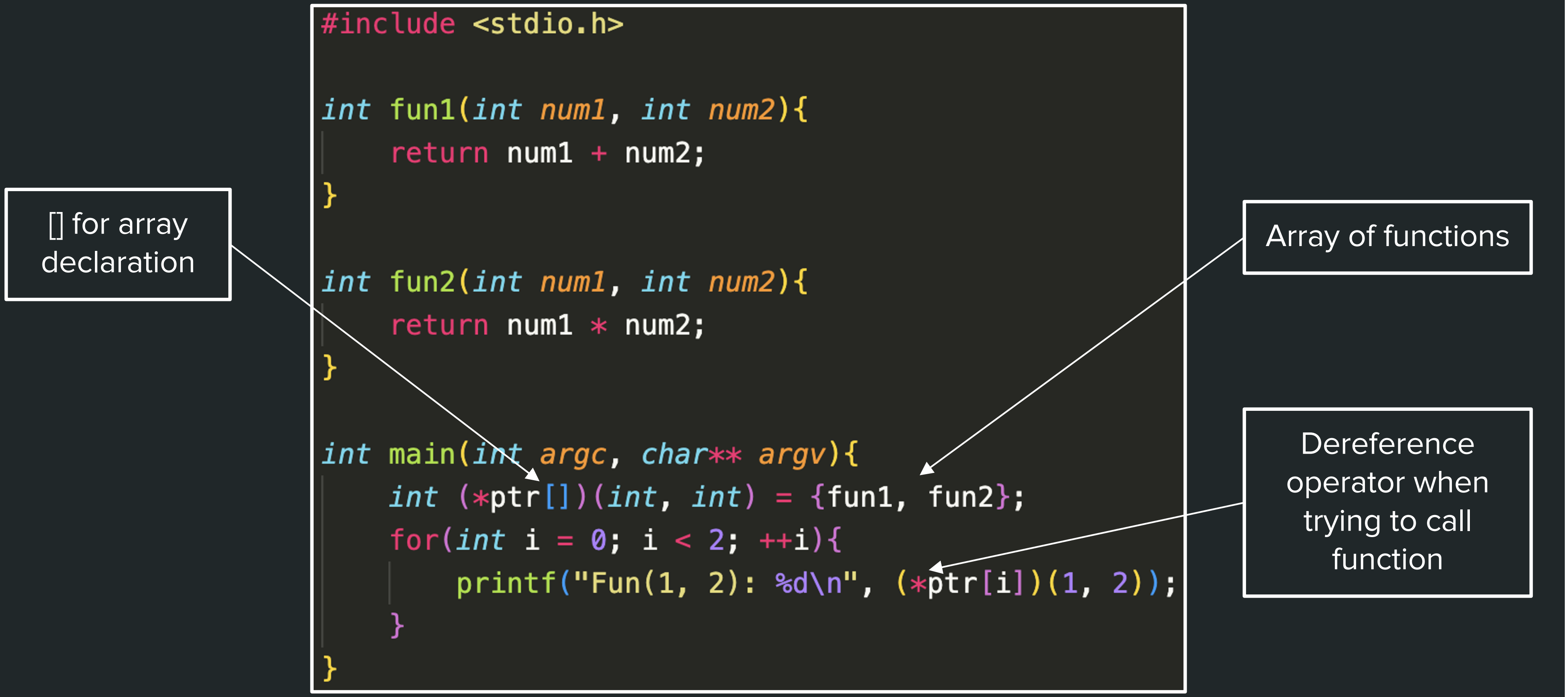
- When talking about pointers, something else comes to mind…
- Arrays are also pointers!
- Can we create an array of functions?
  - Absolutely we can
  - This can actually prove to be quite useful

Example - Declared Array of Function Pointers



```c
#include <stdio.h>

int fun1(int num1, int num2){
    return num1 + num2;
}

int fun2(int num1, int num2){
    return num1 * num2;
}

int main(int argc, char** argv){
    int (*ptr[])(int, int) = {fun1, fun2};
    for(int i = 0; i < 2; ++i){
        printf("Fun(1, 2): %d\n", (*ptr[i])(1, 2));
    }
}
```

Example - Declared Array of Function Pointers



```
#include <stdio.h>

int fun1(int num1, int num2){
    return num1 + num2;
}

int fun2(int num1, int num2){
    return num1 * num2;
}

int main(int argc, char** argv){
    int (*ptr[])(int, int) = {fun1, fun2};
    for(int i = 0; i < 2; ++i){
        printf("Fun(1, 2): %d\n", (*ptr[i])(1, 2));
    }
}
```

[] for array declaration

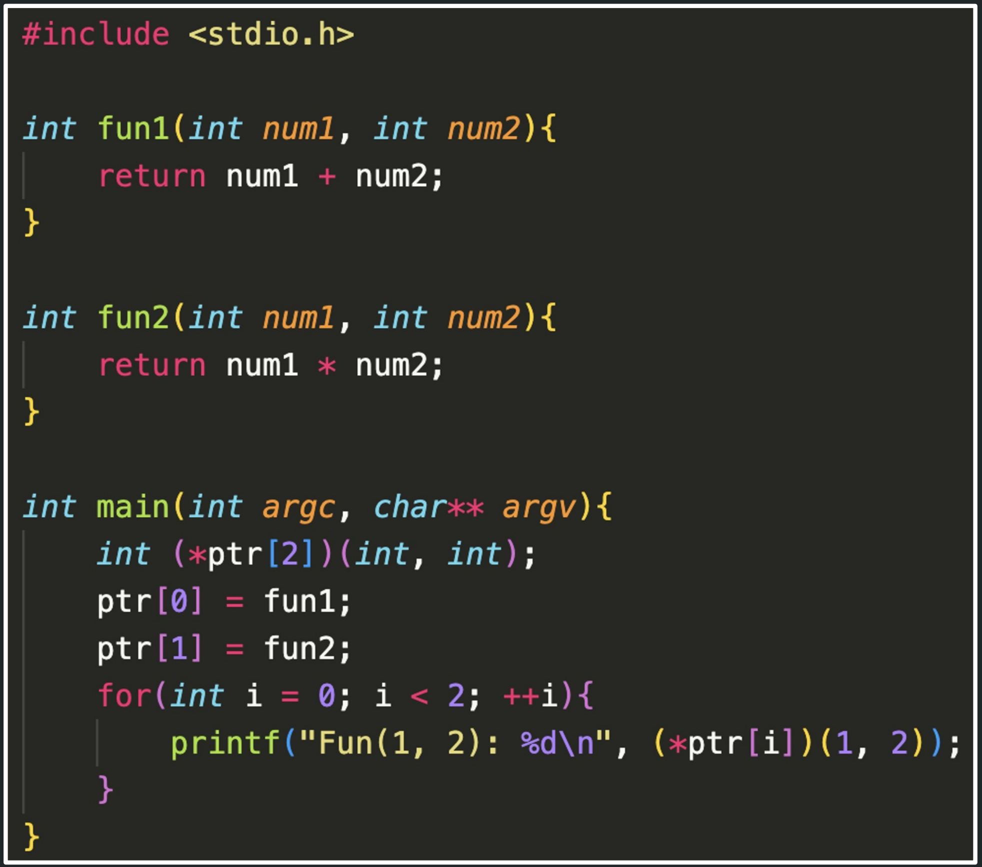
Array of functions

Dereference operator when trying to call function

Array of Function Pointers

- What if I don’t know what functions I want in my array at compile time?
- How can we add functions to the array?
- Let’s try to declare an array of function pointers on the stack without initializing it

Example - Declared Array of Function Pointers



```c
#include <stdio.h>

int fun1(int num1, int num2){
    return num1 + num2;
}

int fun2(int num1, int num2){
    return num1 * num2;
}

int main(int argc, char** argv){
    int (*ptr[2])(int, int);
    ptr[0] = fun1;
    ptr[1] = fun2;
    for(int i = 0; i < 2; ++i){
        printf("Fun(1, 2): %d\n", (*ptr[i])(1, 2));
    }
}
```

Quiz time!

Quiz is named *Week* ?

Password is: ________

Part B - Fork, Exec, and Signal Handling

Part B - Signals

- Processes can communicate to each other with signals
- For example, something we should be familiar with by now is the interrupt signal, or Ctrl+C
    - The interrupt signal will communicate to your process that the program should be terminated

Part B - Signals

- In C, we can actually catch these signals, and add some behavior to them
- You might have seen this in Project 3, where the program didn’t end immediately after Ctrl+C was pressed, but instead printed out some information before exiting
- How can we catch these signals?

Part B - Signals

- In the <stdlib.h> library, there’s a function we can use called **signal**
- **signal(int interrupt, void* function_ptr)**
  - Once signal detects the interrupt specified as a parameter (SIGKILL, SIGINT, etc), it will then execute the function pointed to by function_ptr, then perform the interrupt
- Once we call signal, the process will always be on the lookout for that signal, and if it occurs, will run the function

Part B

- For this part of the lab, we'll need to use our knowledge of fork(), exec(), and signals
- Our program needs to look for the interrupts SIGUSR1 and SIGUSR2
    - If SIGUSR1 is detected, fork a process have it run the command **ls**
    - If SIGUSR2 is detected, fork a process and have it run the command **ls -l -a**
- Your program should also run in the exact specified order and print the specified information
- **IMPORTANT - when running processes with multiple threads, testing it once is not enough**
- Due to the unpredictability of the scheduler, running your program once time may be fine, but the next time text may print out of order
- **BE THOROUGH**

Part A - Dynamic Libraries and Function Pointers

Part A

- For this part of the lab, we need to use **dlopen** and **dlsym** from the <dlfcn.h> library
- We’ll also need to create our plugin.c file and compile it to a shared object file (plugin.so)
  - To create the .so:

```
gcc plugin.c -o plugin.so -shared
```

Part A

- Once we have a shared object file, we can load it into the main program using **dlopen** and call the functions using **dlsym**
- **void* dlopen(char* plugin_path, int mode)**
  - Plugin path will be the so plugin we're trying to run
  - For example, if I want to load the plugin called my_plugin.so, I'd use the path ./my_plugin.so
  - It returns a handle to be used by dlsym and dlclose()
- **Much like malloc, dlopen calls should be matched with a dlclose(handle) call**

Part A

- **void* dlsym(void* handle, const char* fname)**
  - Takes in a handle (returned by dlopen) and a function name to be called
  - Returns a function pointer to that function
- For example, if I have a handle returned from dlopen and I want a function pointer to
  int fun1(int, long):

```
int (*fun_ptr)(int, long) = dlsym(handle, “fun1”);
```

Part A

- From there, we can call the function same as any other function pointer
  - fun_ptr(my_int, my_long)
  - This will call the functions in the shared library

Part A - Assignment

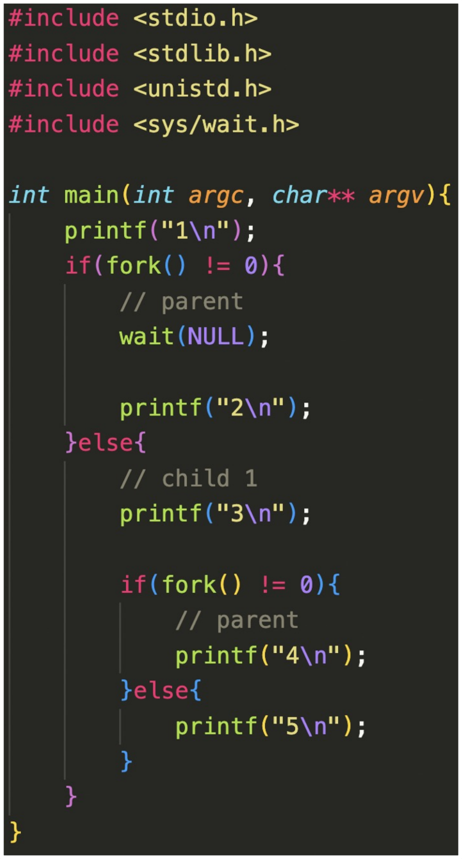
- You will need to write a plugin manager that takes a plugin name as a command line argument
- The plugin should have 3 functions:
  - int initialize()
  - int run()
  - int cleanup()
- Your plugin manager should be able to load and run all three of these functions
  - While also checking for errors

Fork()ing and Exec()uting

An Important Topic - Fork Tracing

- When discussing fork tracing, we need to determine which possible orders the processes can run in
- Since we can use instructions like wait(), we can limit these number of possible orders
- However, we still need to trace which outputs are possible with a given program

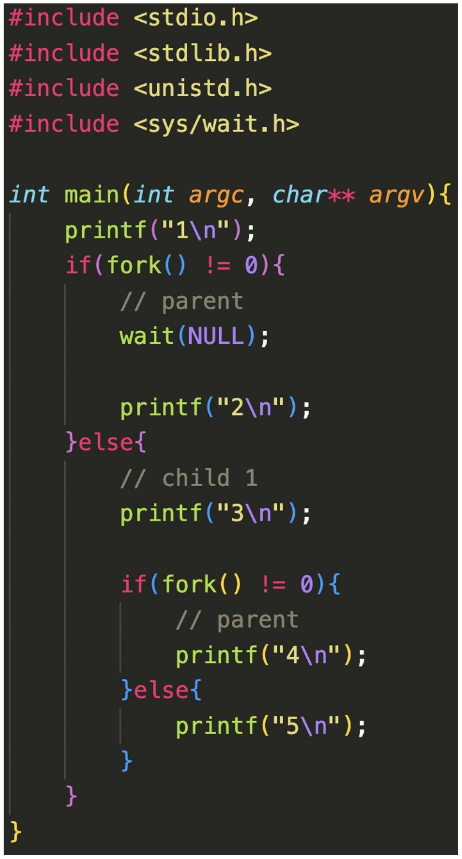
Fork Tracing Example



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char** argv){
    printf("1\n");
    if(fork() != 0){
        // parent
        wait(NULL);

        printf("2\n");
    }else{
        // child 1
        printf("3\n");

        if(fork() != 0){
            // parent
            printf("4\n");
        }else{
            printf("5\n");
        }
    }
}
```

Can Certain Print Orders Happen?

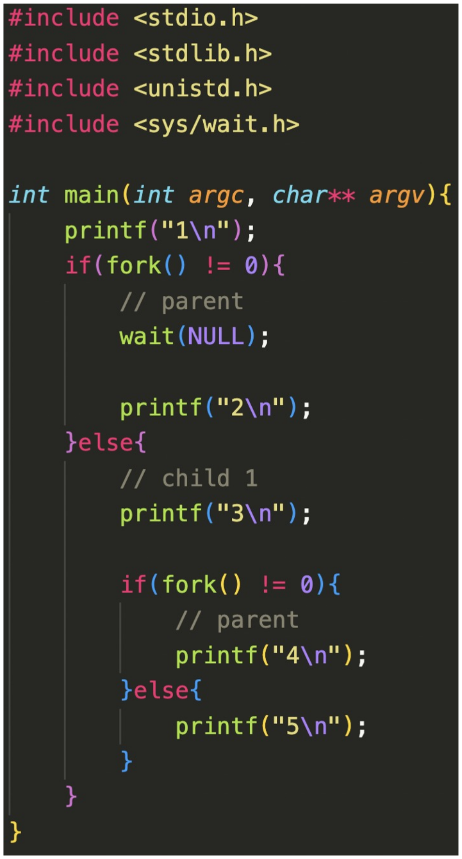
- Can we have an order of 2, 3, 4, 5, 1?



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char** argv){
    printf("1\n");
    if(fork() != 0){
        // parent
        wait(NULL);

        printf("2\n");
    }else{
        // child 1
        printf("3\n");

        if(fork() != 0){
            // parent
            printf("4\n");
        }else{
            printf("5\n");
        }
    }
}
```

Can Certain Print Orders Happen?

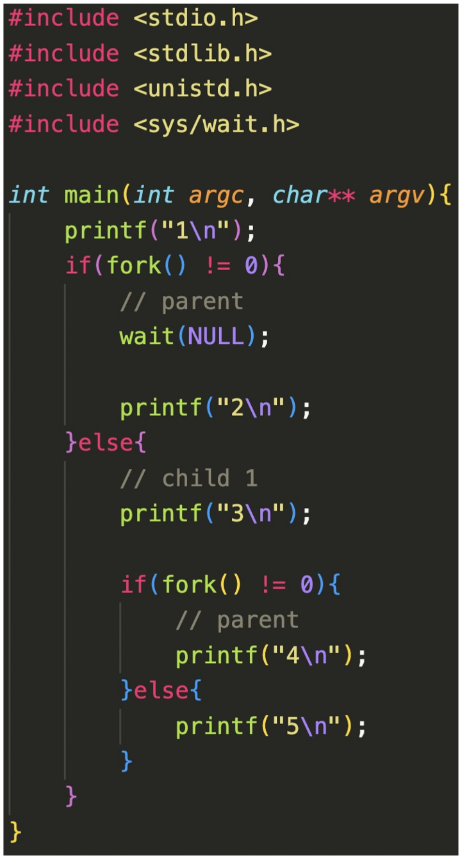
- Can we have an order of 2, 3, 4, 5, 1?
  - NO! But why?
  - At the beginning, there’s only one parent thread, so 1 must ALWAYS be printed first



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char** argv){
    printf("1\n");
    if(fork() != 0){
        // parent
        wait(NULL);

        printf("2\n");
    }else{
        // child 1
        printf("3\n");

        if(fork() != 0){
            // parent
            printf("4\n");
        }else{
            printf("5\n");
        }
    }
}
```

Can Certain Print Orders Happen?

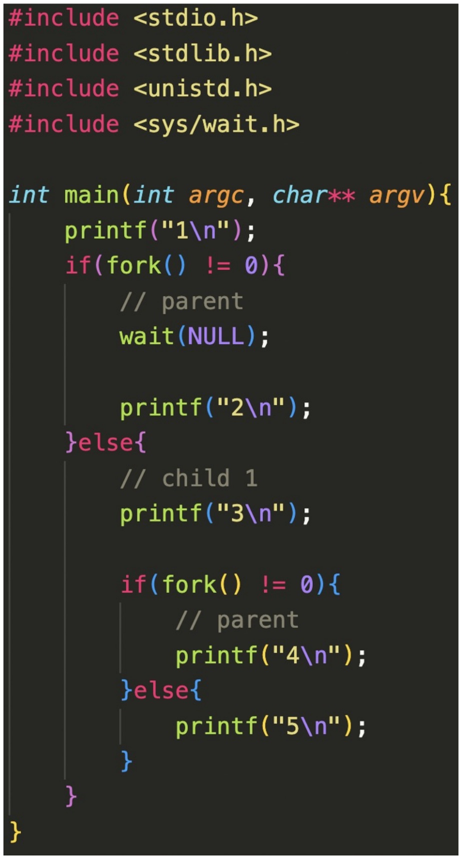
- Can we have an order of 2, 3, 4, 5, 1?
  - NO! But why?
  - At the beginning, there’s only one parent thread, so 1 must ALWAYS be printed first
- What about the order 1, 2, 3, 4, 5 happen?



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char** argv){
    printf("1\n");
    if(fork() != 0){
        // parent
        wait(NULL);

        printf("2\n");
    }else{
        // child 1
        printf("3\n");

        if(fork() != 0){
            // parent
            printf("4\n");
        }else{
            printf("5\n");
        }
    }
}
```

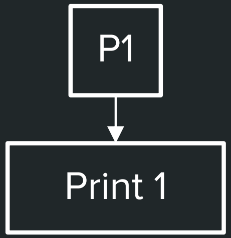
Can Certain Print Orders Happen?

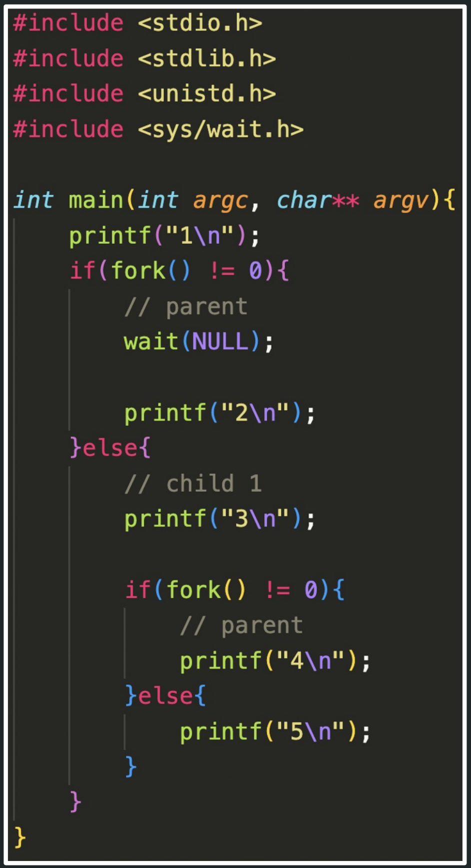
- Can we have an order of 2, 3, 4, 5, 1?
  - NO! But why?
  - At the beginning, there’s only one parent thread, so 1 must ALWAYS be printed first
- What about the order 1, 2, 3, 4, 5 happen?
  - NO! Because the first parent thread waits for the child to finish, so 2 can only be printed after the child finishes executing (after 3 and 4... but NOT 5)



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char** argv){
    printf("1\n");
    if(fork() != 0){
        // parent
        wait(NULL);

        printf("2\n");
    }else{
        // child 1
        printf("3\n");

        if(fork() != 0){
            // parent
            printf("4\n");
        }else{
            printf("5\n");
        }
    }
}
```

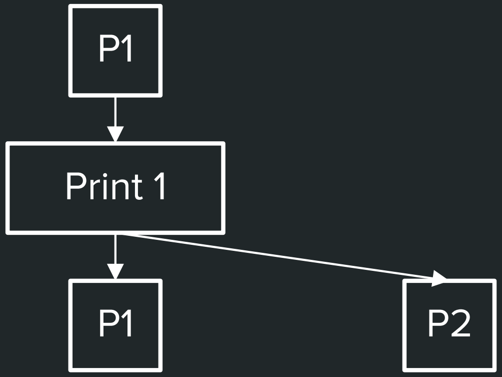
Building a Fork Tree

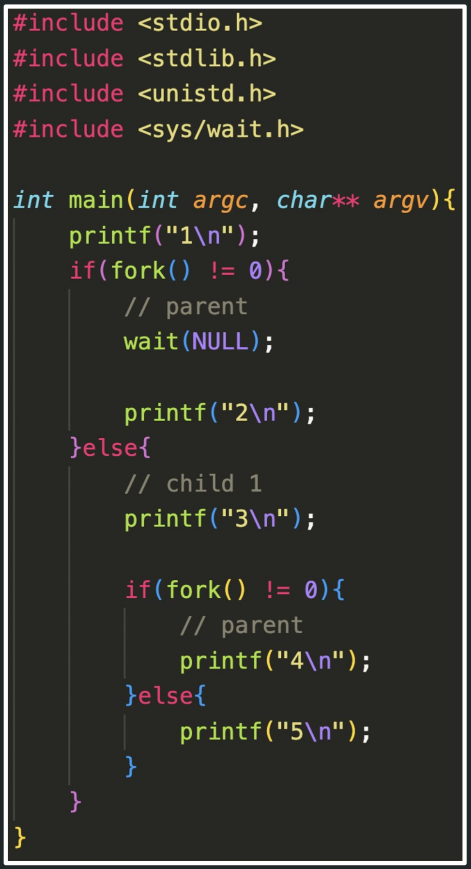




```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char** argv){
    printf("1\n");
    if(fork() != 0){
        // parent
        wait(NULL);

        printf("2\n");
    }else{
        // child 1
        printf("3\n");

        if(fork() != 0){
            // parent
            printf("4\n");
        }else{
            printf("5\n");
        }
    }
}
```

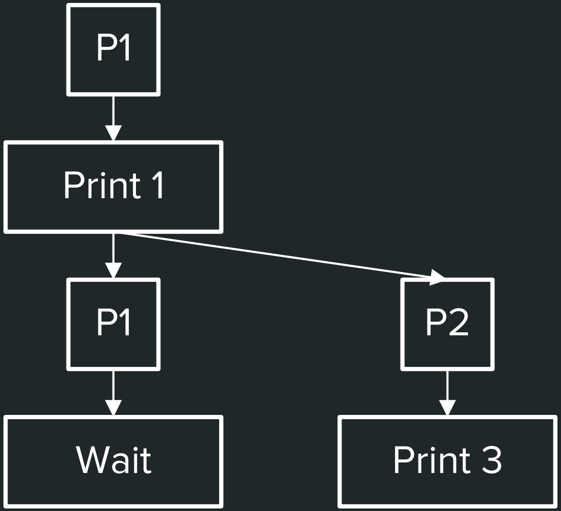
Building a Fork Tree

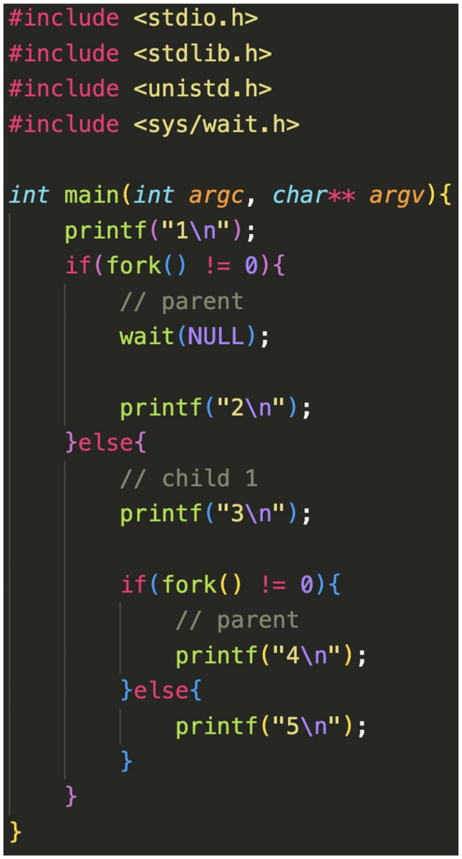




```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char** argv){
    printf("1\n");
    if(fork() != 0){
        // parent
        wait(NULL);

        printf("2\n");
    }else{
        // child 1
        printf("3\n");

        if(fork() != 0){
            // parent
            printf("4\n");
        }else{
            printf("5\n");
        }
    }
}
```

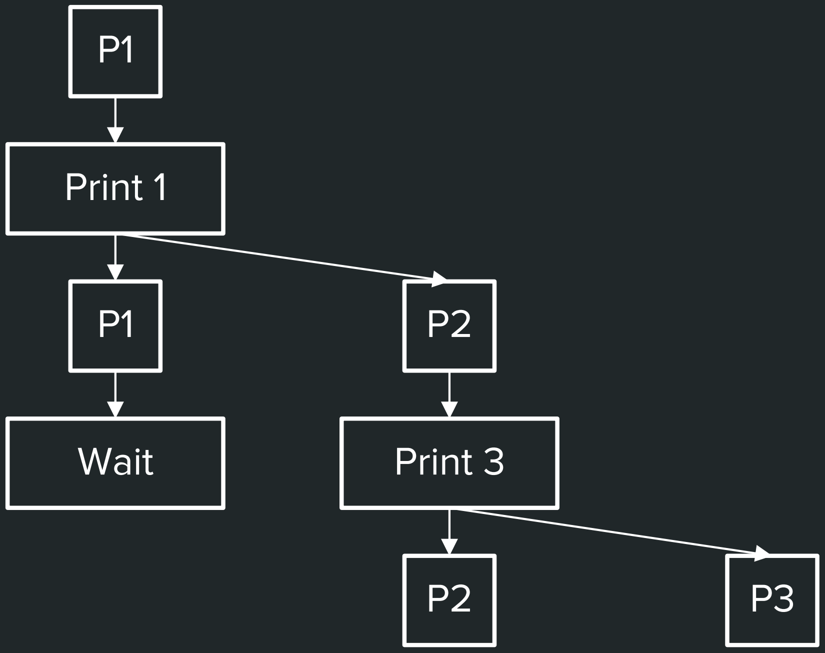
Building a Fork Tree

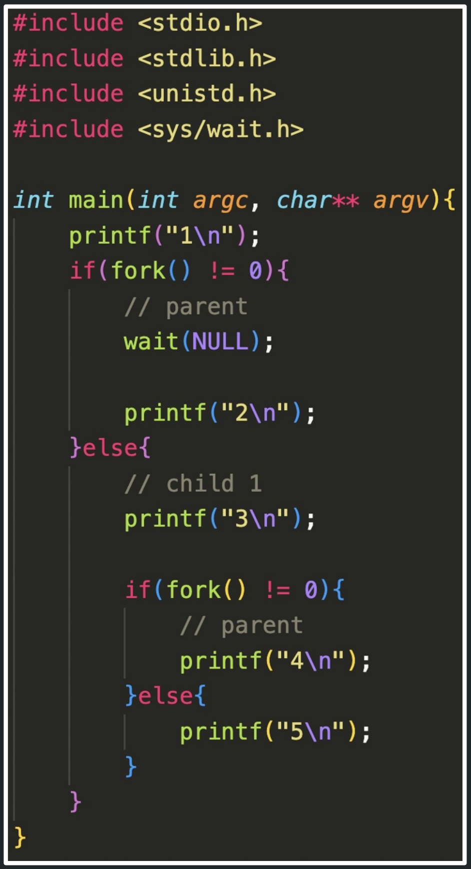




```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char** argv){
    printf("1\n");
    if(fork() != 0){
        // parent
        wait(NULL);

        printf("2\n");
    }else{
        // child 1
        printf("3\n");

        if(fork() != 0){
            // parent
            printf("4\n");
        }else{
            printf("5\n");
        }
    }
}
```

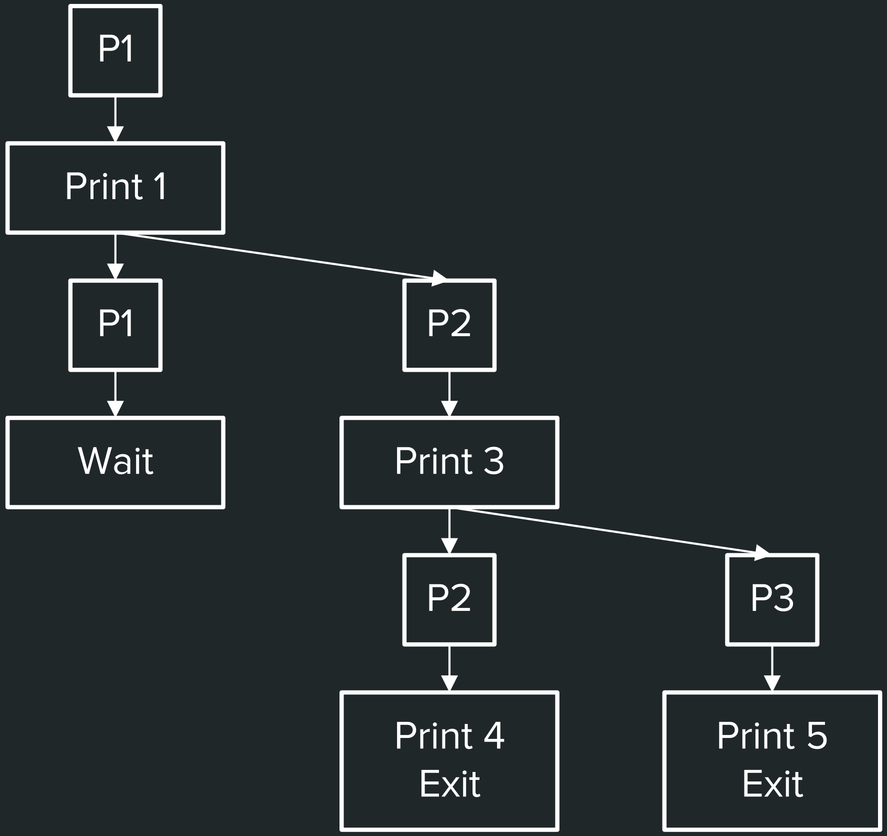
Building a Fork Tree

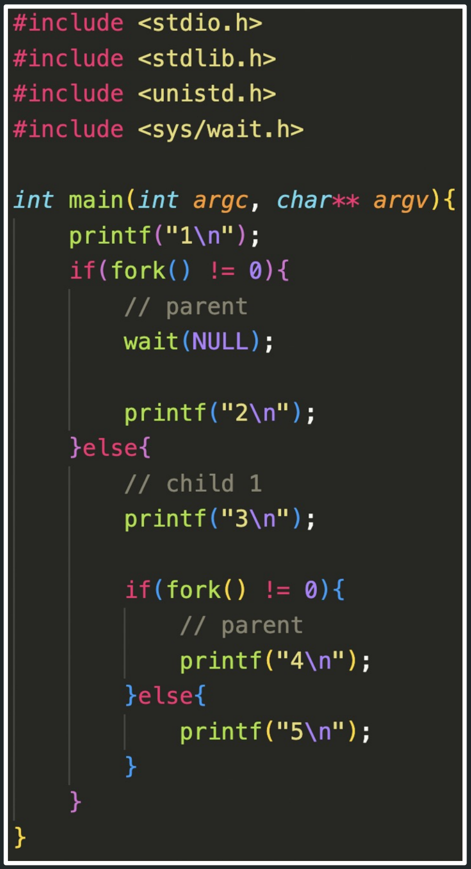




```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char** argv){
    printf("1\n");
    if(fork() != 0){
        // parent
        wait(NULL);

        printf("2\n");
    }else{
        // child 1
        printf("3\n");

        if(fork() != 0){
            // parent
            printf("4\n");
        }else{
            printf("5\n");
        }
    }
}
```

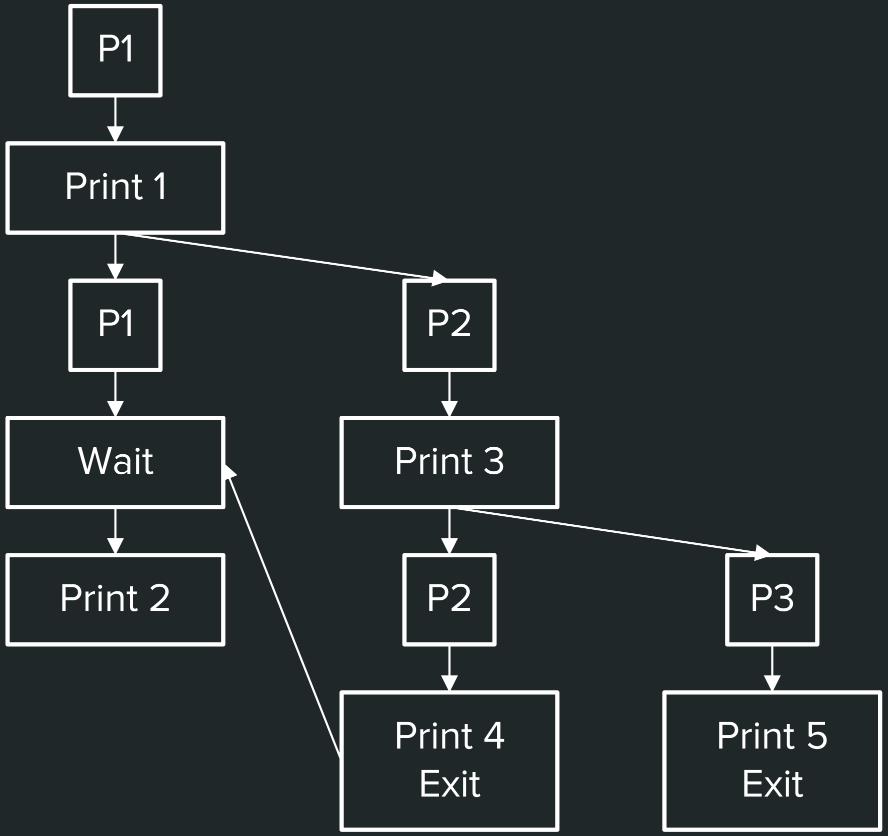
Building a Fork Tree





```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char** argv){
    printf("1\n");
    if(fork() != 0){
        // parent
        wait(NULL);

        printf("2\n");
    }else{
        // child 1
        printf("3\n");

        if(fork() != 0){
            // parent
            printf("4\n");
        }else{
            printf("5\n");
        }
    }
}
```

Building a Fork Tree



P1

Print 1

P1

P2

Wait

Print 3

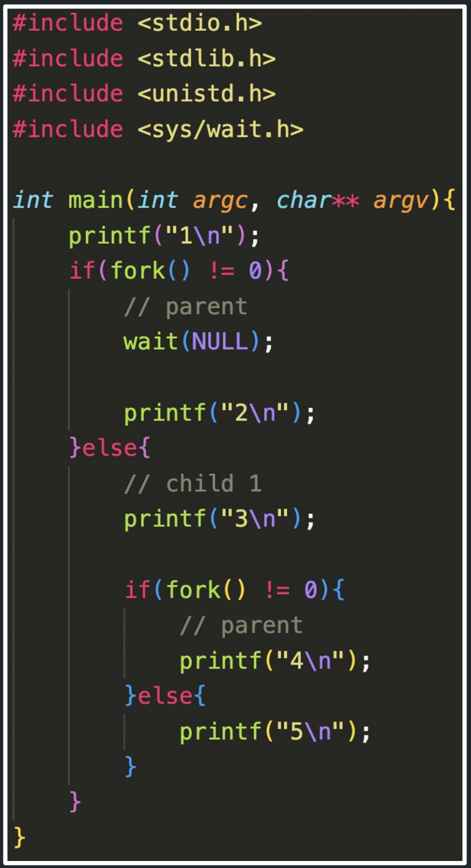
Print 2

P2

P3

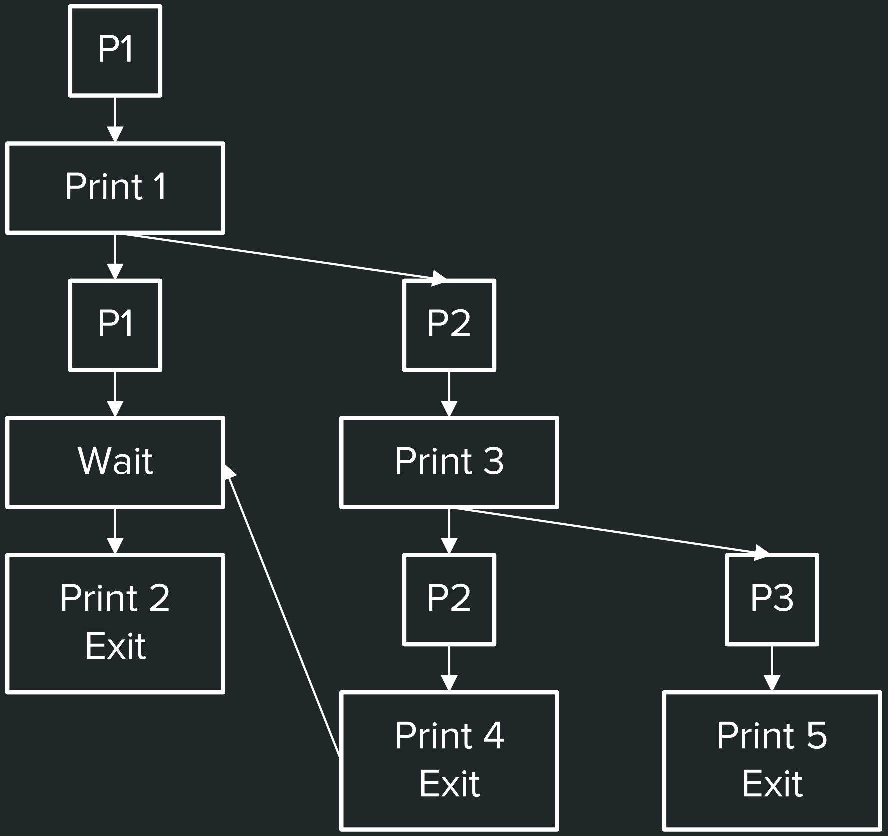
Print 4
Exit

Print 5
Exit



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char** argv){
    printf("1\n");
    if(fork() != 0){
        // parent
        wait(NULL);

        printf("2\n");
    }else{
        // child 1
        printf("3\n");

        if(fork() != 0){
            // parent
            printf("4\n");
        }else{
            printf("5\n");
        }
    }
}
```

Building a Fork Tree



P1

Print 1

P1

P2

Wait

Print 3

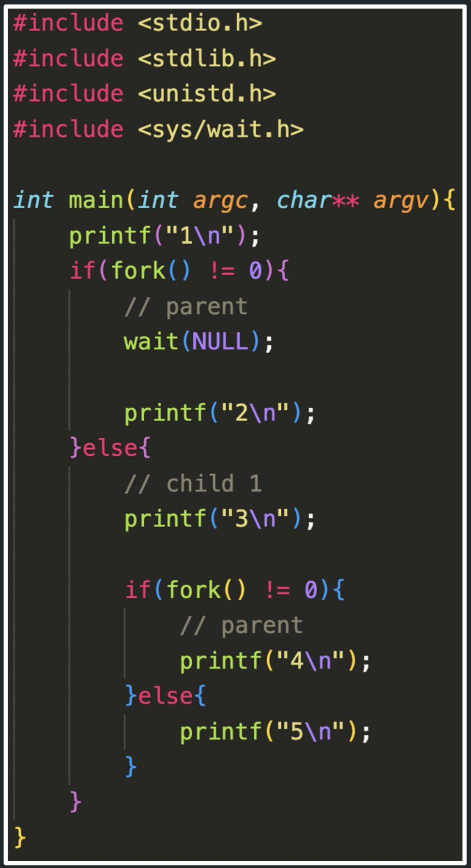
Print 2
Exit

P2

P3

Print 4
Exit

Print 5
Exit



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char** argv){
    printf("1\n");
    if(fork() != 0){
        // parent
        wait(NULL);

        printf("2\n");
    }else{
        // child 1
        printf("3\n");

        if(fork() != 0){
            // parent
            printf("4\n");
        }else{
            printf("5\n");
        }
    }
}
```

man strtok abridged

The strtok() function can help tokenize strings

```
#include <string.h>
char *strtok(char *str, const char *delim);
```

- Breaks string str into a series of tokens using the delimiter delim.
- Returns a pointer to the next token, or NULL if there are no more tokens.

Called in one of two ways:

1. `strtok(str, d) // starts processing a new string`
2. `strtok(NULL, d) // continue processing a string`

A strtok() example

```c
#include <stdio.h>
#include <string.h>
int main(){
    char str[] = "I:love-programming";
    char delim[] = "-:";
    char *token;
    token = strtok(str, delim);
    printf("%s\n", token);
    return 0;
}
```

```
$ ./strtok_example
I
```

What will be printed?

A strtok() example

```c
#include <stdio.h>
#include <string.h>
int main(){
    char str[] = "I:love-programming";
    char delim[] = "-:";
    char *token;
    token = strtok(str, delim);
    printf("%s\n", token);
    token = strtok(str, delim);
    printf("%s\n", token);
    return 0;
}
```

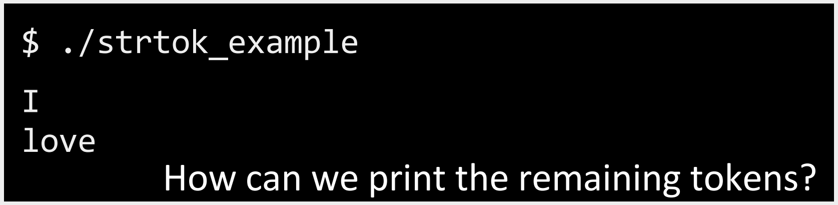
```
$ ./strtok_example
I
I
```

🤔 But the second token should be "love"

What will be printed?

A `strtok()` example

```c
#include <stdio.h>
#include <string.h>
int main(){
   char str[] = "I:love-programming";
   char delim[] = "-:";
   char *token;
   token = strtok(str, delim);
   printf("%s\n", token);
   token = strtok(NULL, delim);
   printf("%s\n", token);
   return 0;
}
```

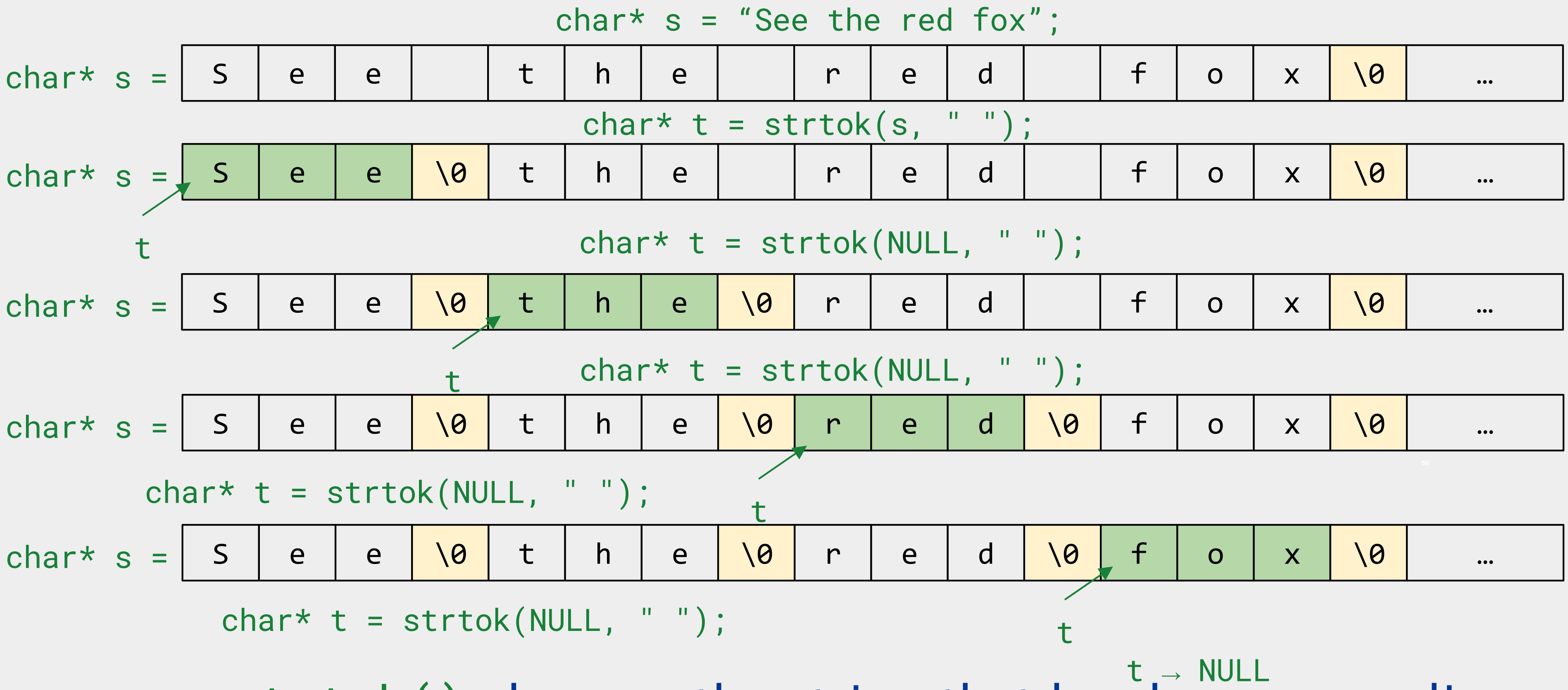


```
$ ./strtok_example
I
love
```

How can we print the remaining tokens?

What will be printed?

A strtok() example



`char* s = “See the red fox”;`

char* s =	S	e	e		t	h	e		r	e	d		f	o	x	\0	…

`char* t = strtok(s, " ");`

char* s =	S	e	e	\0	t	h	e		r	e	d		f	o	x	\0	…

t

`char* t = strtok(NULL, " ");`

char* s =	S	e	e	\0	t	h	e	\0	r	e	d		f	o	x	\0	…

t

`char* t = strtok(NULL, " ");`

char* s =	S	e	e	\0	t	h	e	\0	r	e	d	\0	f	o	x	\0	…

`char* t = strtok(NULL, " ");`

t

char* s =	S	e	e	\0	t	h	e	\0	r	e	d	\0	f	o	x	\0	…

`char* t = strtok(NULL, " ");`

t

t → NULL

strtok() changes the string that has been parsed!