/unc/comp211

Systems Fundamentals

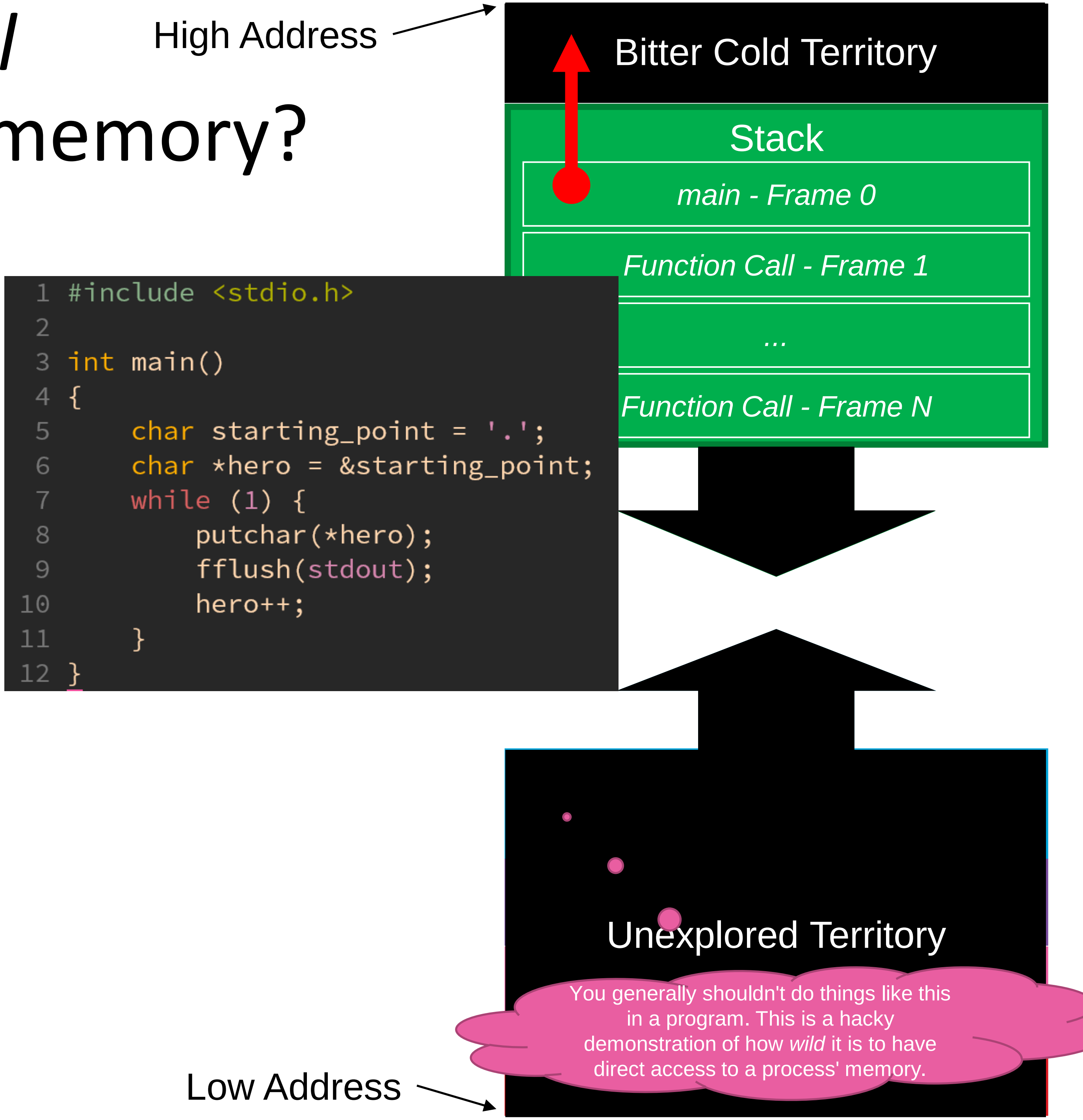
A Process'
Arguments **and** *Environment Variables* !

What lies *North* of *the wall* (*read: stack*) in a process' memory?

- Let's go on a *perilous* adventure!
  - We'll establish a "*hero"* (*read: pointer*), walk it "north" one byte at a time, and print each byte as we reach it.
- Create a file named **adventure.c** in vim
  - Its contents are shown right.
  - Compile & run it:

```
$ gcc -o adventure adventure.c
$ ./adventure wildlings giants
```

- Do you see anything interesting in the output?
  - Hint: look for "./adventure" "wildlings" and "giants"!
  - Once you're getting a "Segmentation fault" you've reached the edge of the world



```c
#include <stdio.h>

int main()
{
    char starting_point = '.';
    char *hero = &starting_point;
    while (1) {
        putchar(*hero);
        fflush(stdout);
        hero++;
    }
}
```

You generally shouldn't do things like this in a program. This is a hacky demonstration of how *wild* it is to have direct access to a process' memory.

The *Arguments* and *Environment Variables* a Program is Executed With

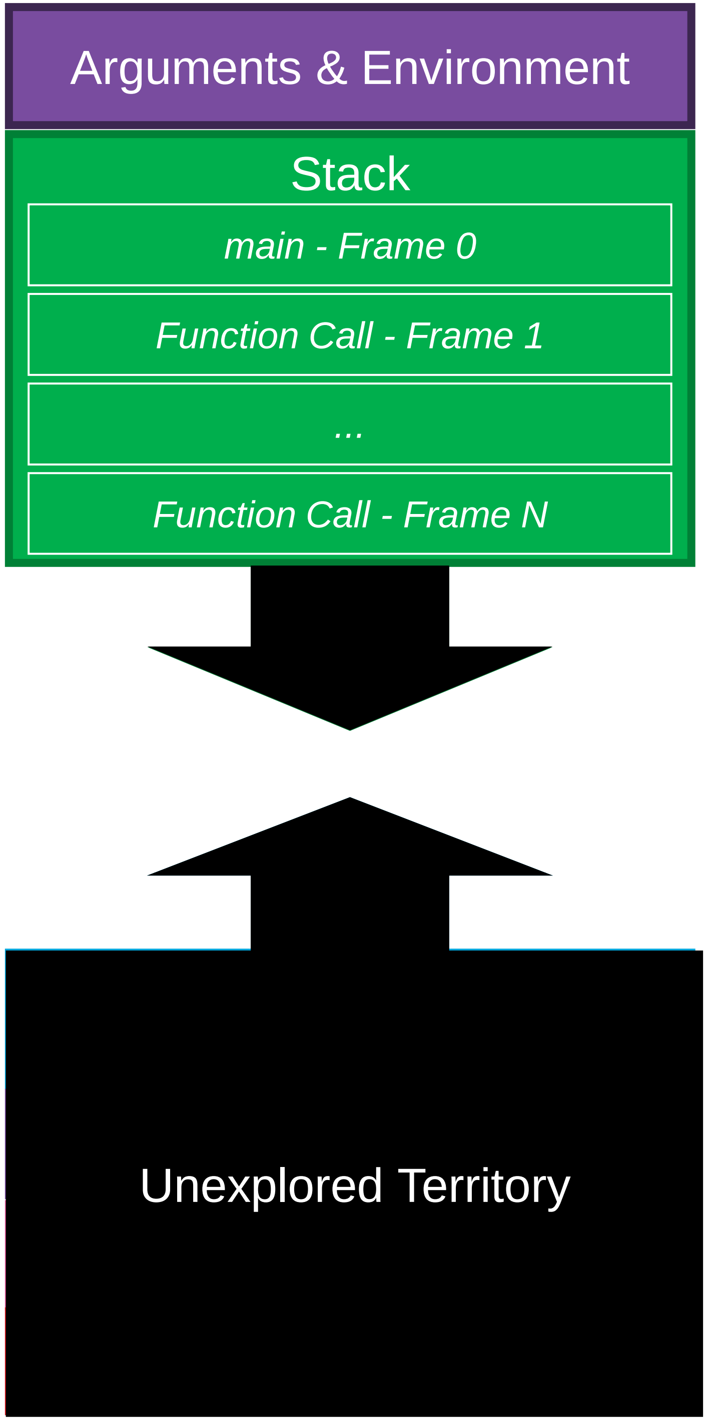
- When you run a program, data provided by the user from *outside* the program is loaded into the process' memory.
  - These values are used as *inputs* and *context* to the program.

1. **Argument Values**

   ```
   $ ./adventure wildlings giants
   ```

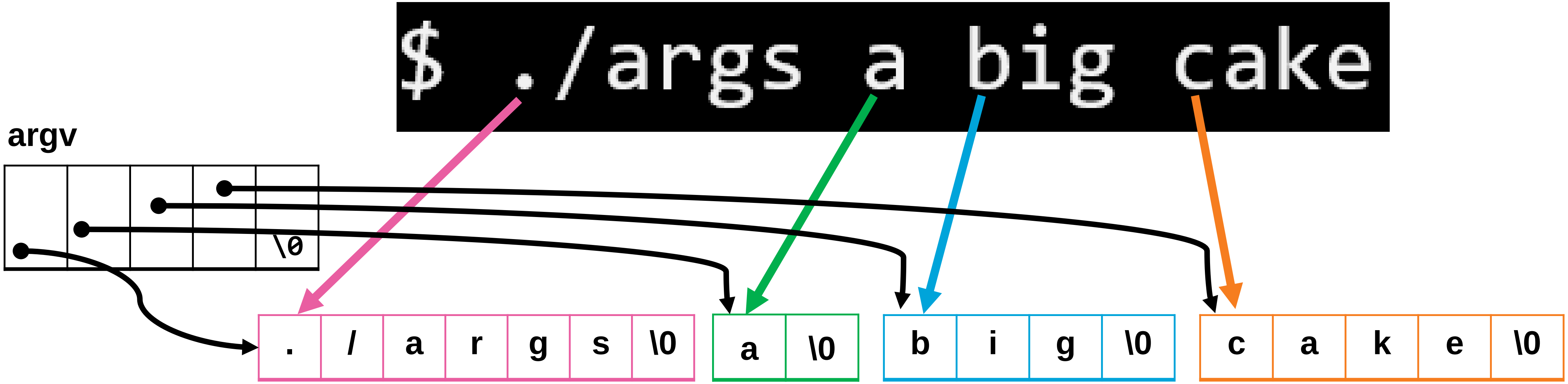
   - The **name/path of the program** and any **arguments**.

2. **Environment Variables**
   - These variables are often used for *configuration* purposes and managed by your command-line interface shell
   - You setup environment variables without knowing: GIT_AUTHOR_NAME
   - The printenv program will dump your environment variables



Program Arguments (1 / 4)

- When you execute a program, the shell reads your command as character data and breaks it up into *argument* ***tokens***:
  - 0 - The 0th token is conventionally the name/path of the program
  - 1...N - The 1st through Nth tokens

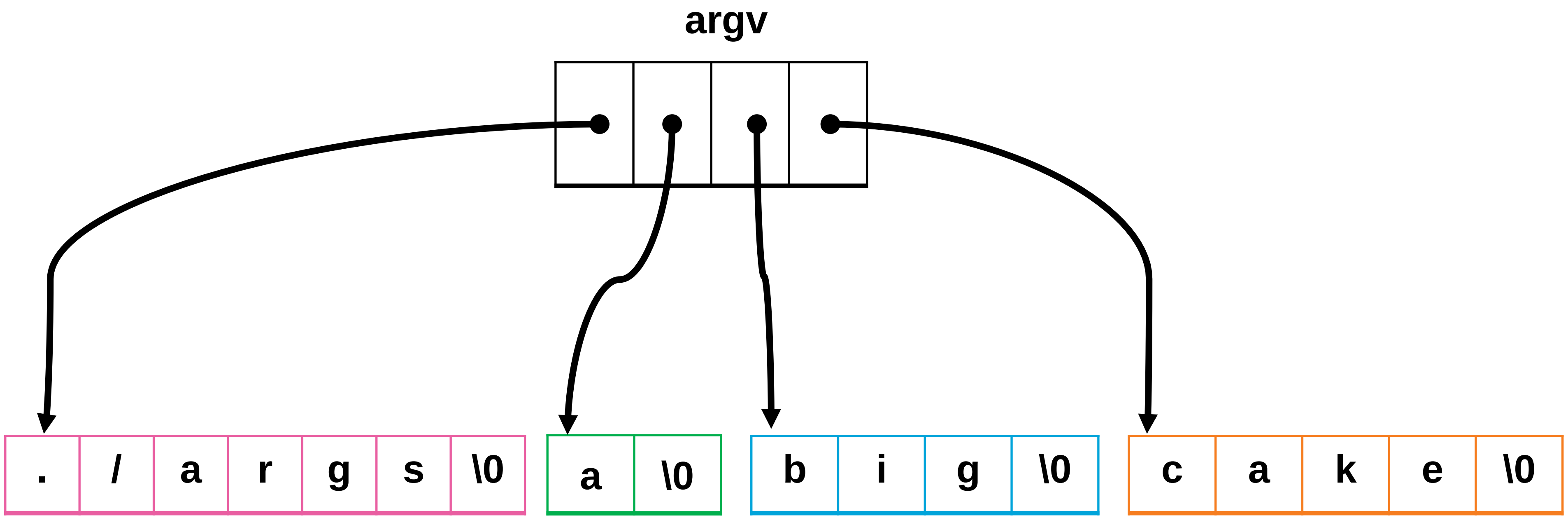


```
$ ./args a big cake
```

- Pointers to each of these values are added to an array of char pointers
  - **argv** is the conventional name of this array, short for "argument values"

Program Arguments (2 / 4)

- The **shell** tells the operating system to **exec**ute the program
  - This happens via a system function call
  - The operating system "function" call is given a pointer to **argv**
    - Technically this array must be null terminated, but we're not illustrating that here.

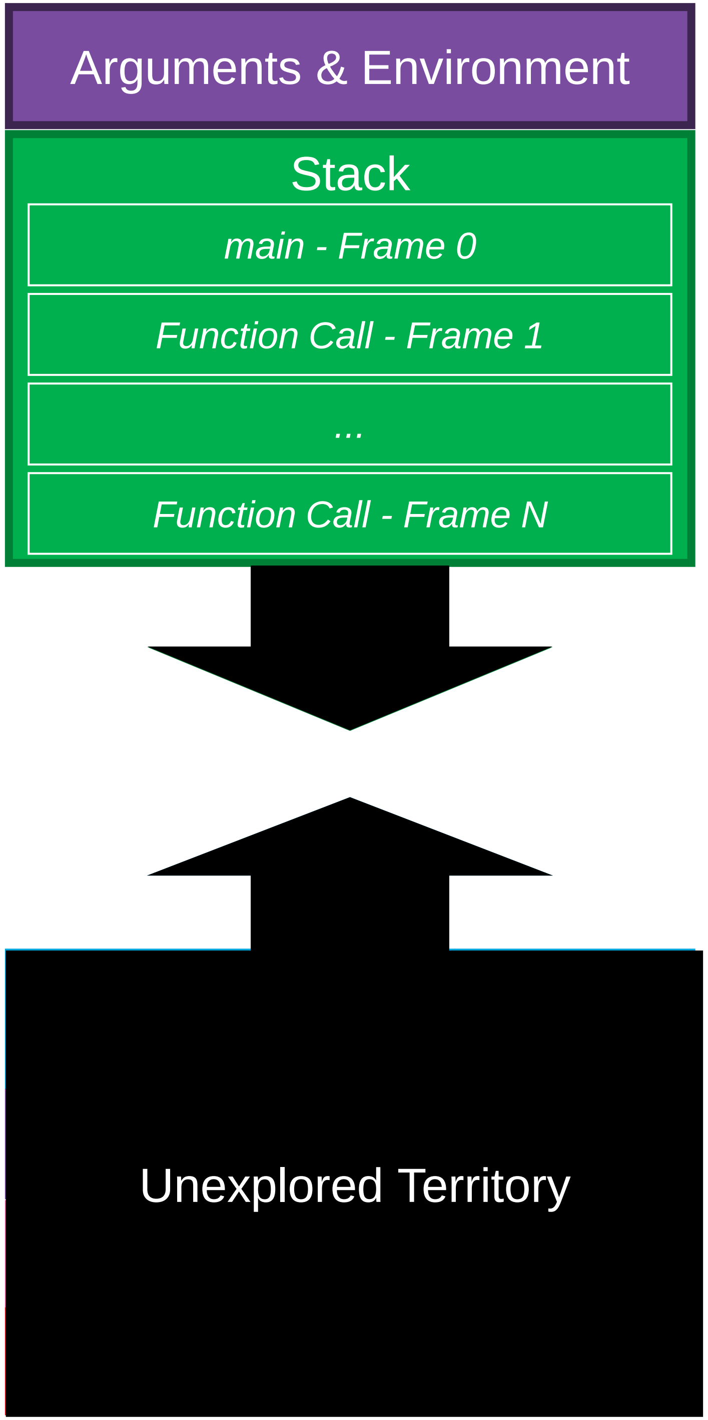


Program Arguments (3 / 4)

- Before your program enters the **main** function, as the operating system sets up memory for the process, it copies **argv** and the **char[]** data it points to from the shell's memory into the process' memory.
  - Where? Higher than the call stack.
    - You explored this area in the opening example!

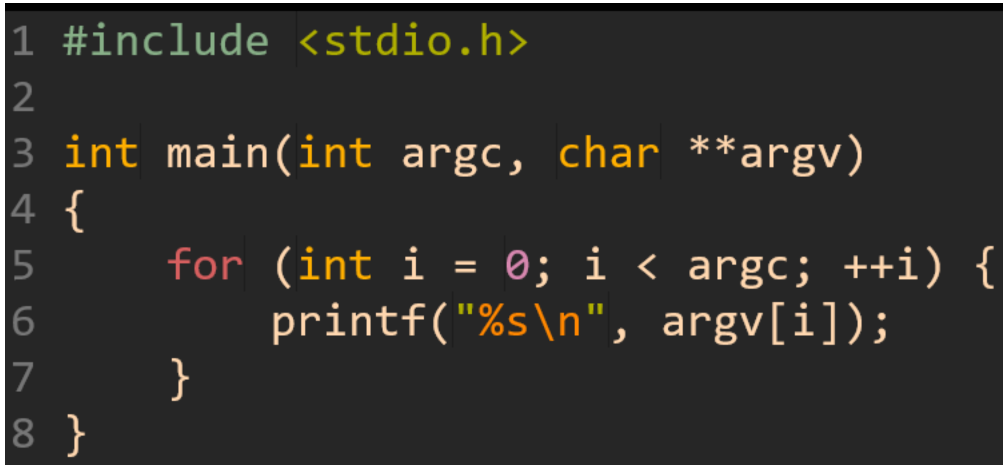
- In C, when you write a main function with the params: **int argc**, **char *argv[]**
  - The *count of argument pointers* is assigned to **argc**
  - A pointer to the array of pointers to char[] arguments is assigned to **argv**

- This is how you can access command-line arguments!



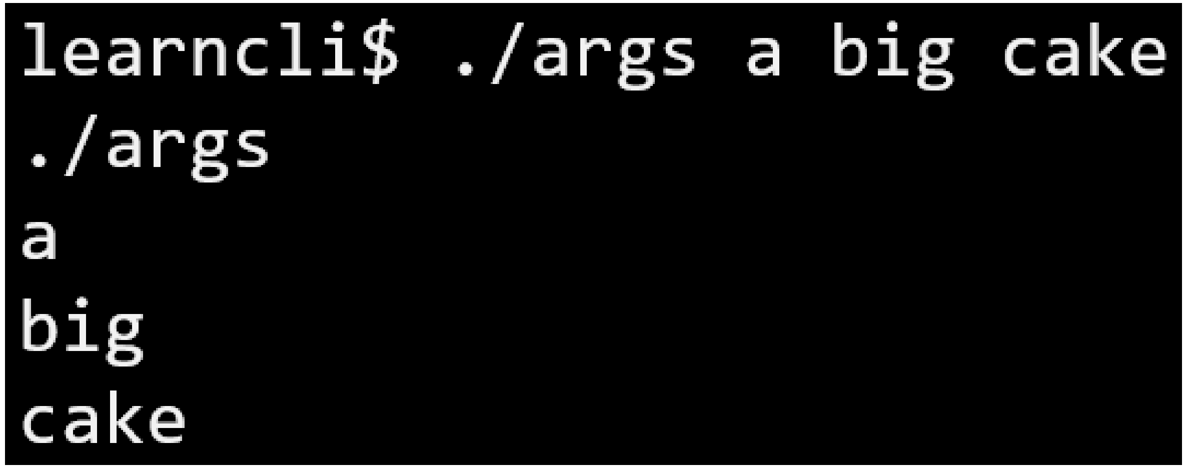
Program Arguments (4 / 4)

- Let's try writing a simple program to print command-line args together.
- Source code: **`args.c`**
- Compile: **`gcc -o args args.c`**
- Run: **`./args a big cake`**



```c
#include <stdio.h>

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



```
learncli$ ./args a big cake
./args
a
big
cake
```

- Rather than using indexing notation with the argv pointer, try using array arithmetic and dereferencing, instead!

Aside: About Java's `main` method…

- Remember writing the following method signature?...

```
class Foo {
   public static void main(String[] args) { /* ... */ }
}
```

- What was *up* with **String[] args**? The same concept!

- When you run a Java program from the command-line, the char[] values you give as arguments to the shell ultimately are copied into the String[] args of your main function.

- Every general-purpose programming language has a straightforward way of reading command-line arguments along these lines!

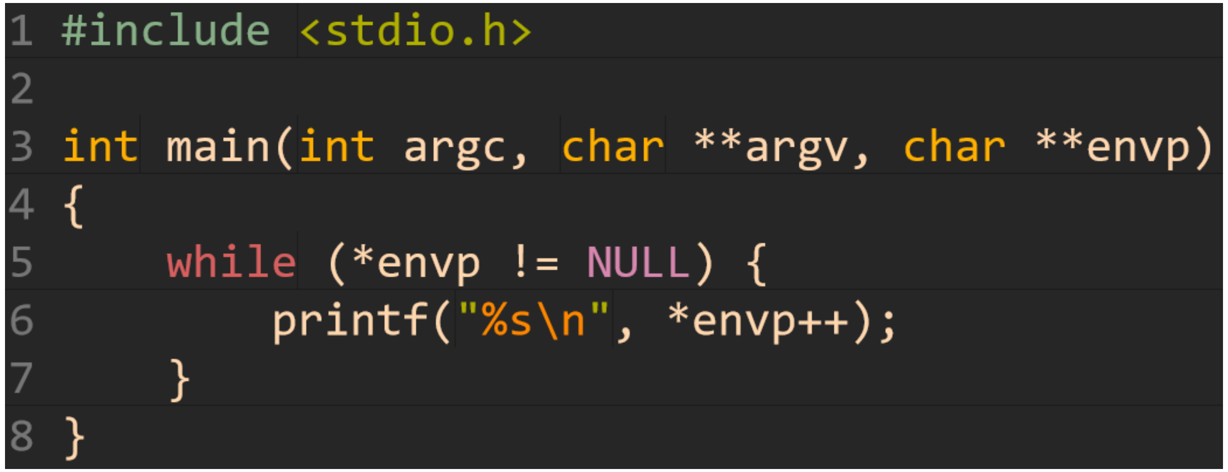
Environment Variables (1 / 3)

- Your shell session maintains a set of named Environment Variables
  - Example: the PWD variable is the path to your working directory
- You can use environment variables from the shell: **echo PWD is ${PWD}**
- The purpose of environment variables is to provide *context* to programs
  - You established your git author and email address via environment variables in an earlier lecture. You can try printing it out: **echo ${GIT_AUTHOR_NAME}**
- Environment variables are used commonly in industry
  - Development: to configure API keys to services you're using such as AWS
  - Production: to manage application configuration in server programs
- Later this semester we'll spend more time on shell variables, for now:
  - How does a program access environment variables?

Environment Variables (2 / 3)

- Just like *arguments*, environment variables can be accessed through a conventional parameter in the **main** function.

- Also just like arguments, "the environment" is given to you as a pointer to an array of char[] pointers, conventionally named **envp**.
    - **Like argv**, the array of environment variable pointers is **null terminated**.
    - Unlike argv, you are not given a count parameter like argc.

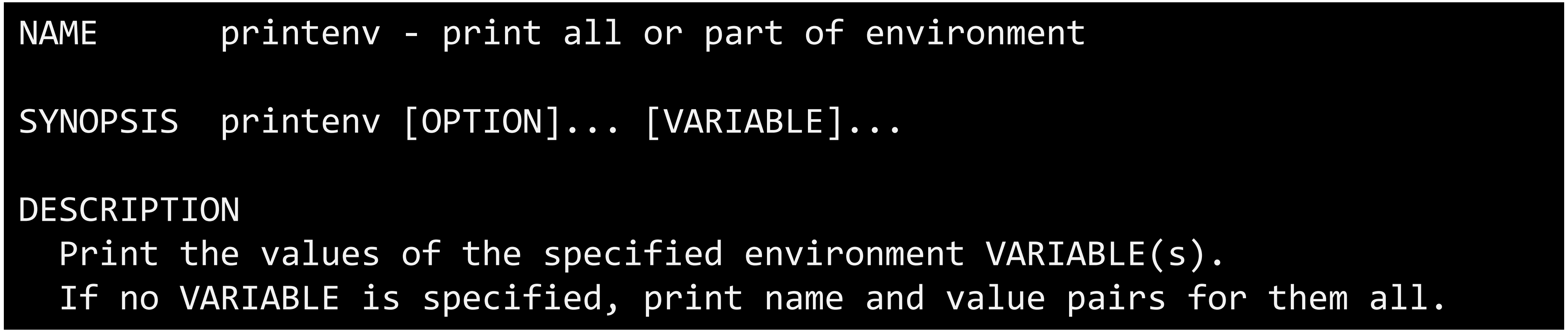


```
#include <stdio.h>

int main(int argc, char **argv, char **envp)
{
    while (*envp != NULL) {
        printf("%s\n", *envp++);
    }
}
```

Environment Variables (3 / 3)

- The program **printenv** is a standard system utility



```
NAME      printenv - print all or part of environment

SYNOPSIS  printenv [OPTION]... [VARIABLE]...

DESCRIPTION
  Print the values of the specified environment VARIABLE(s).
  If no VARIABLE is specified, print name and value pairs for them all.
```

- Usage:

`$ printenv` # prints all name/value pairs

`$ printenv PWD` # prints the value assigned to PWD variable

`$ printenv PWD GIT_AUTHOR_NAME` # prints both values on separate lines

- The next problem set will be a short, naive implementation of **printenv** using pointer arithmetic only.