Command Line Arguments & Multi-dimensional Arrays

Rob Hackman

Winter 2025

University of Alberta

Table of Contents

Command line arguments

Recall from learning how to use bash that when asking your operating system to execute a program you can also pass it command line arguments.

How can we access the command line arguments in our C programs?

argc & argv

Command line arguments are passed by the operating system into our program through the parameters of `main`.

Recall that `main` should actually have two parameters, an `int` and a `char *argv[]`.

```
1  int main(int argc, char *argv[]) {
2    printf("%d args\n", argc);
3    for (size_t i = 0; i < argc; ++i) {
4      printf("argv[%lu]: %s\n", i, argv[i]);
5    }
6  }
```

argc & argv

Command line arguments are passed by the operating system into our program through the parameters of `main`.

Recall that `main` should actually have two parameters, an `int` and a `char *argv[]`.

```
1  int main(int argc, char *argv[]) {
2    printf("%d args\n", argc);
3    for (size_t i = 0; i < argc; ++i) {
4      printf("argv[%lu]: %s\n", i, argv[i]);
5    }
6  }
```

Note: Running this program we are reminded: the first command line argument is always the name of the program itself!

`char **?`

What is the type of `argv` which was declared `char *argv[]`?

Let's apply the *spiral rule* to read this type.

Spiral Rule

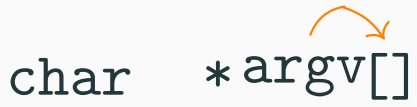
The spiral rule says to read variable types starting from the variable name and going in a clockwise circle.

```
char  *argv[]
```

argv is

Spiral Rule

The spiral rule says to read variable types starting from the variable name and going in a clockwise circle.



argv is an array

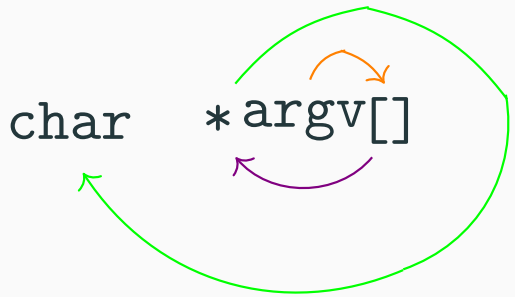
Spiral Rule

The spiral rule says to read variable types starting from the variable name and going in a clockwise circle.

argv is an array of pointers

Spiral Rule

The spiral rule says to read variable types starting from the variable name and going in a clockwise circle.



argv is an array of pointers characters

`argv` really

We know that declaring a parameter as an array type really means a pointer though due to array decay.

So really, our `argv` parameter could also be declared as such:

```
int main(int argc, char **argv)
```

Where here we would say `argv` is a pointer to a pointer to a character.

What we really have is `argv` is a pointer to an array of pointers, and each of the pointers in that array is a pointer to a null-terminated array of characters (a string!).

When does a pointer point at an array or an individual item?

You can't know just by looking at a pointer if it points at an array of items or an individual item.

We know `argv` is an array by convention, and because we're passed the length of it in `argc`. We also know strings are null-terminated arrays of characters.

It is only by convention that we know if an array points at a pointer or an individual piece of data. Documenting functions you write to clearly detail parameters and return values can help know when you mean one or the other!

Practice Question

1. Write a program that takes in two command line arguments that represent integers. Your program should multiply those two numbers together and print out the result. Remember your command line arguments are provided as strings!
2. Update the word count program we wrote earlier to mimic support of the `-l`, `-w`, and `-c` flags that the built-in `wc` command provides.

Table of Contents

2D arrays

We glossed over the fact that `argv` was our first example of a *2-Dimensional Array*.

In C it is common to implement a 2D array as an array of pointers (or, quite similarly, a pointer to an array of pointers).

A pointer to an array of pointers to integers would be the type `int **`. We also call a "pointer to a pointer" a *double pointer*. Similarly, an N-dimensional array can be an array of pointers to pointers to pointers ... to pointers

One can allocate 2D arrays on the stack, but we won't worry about that feature of C.

Creating a 2D array

To create a 2D array one must:

1. Allocate the outer array of pointers.
2. Allocate each inner array of values and populate them as desired.
3. Store pointers to the inner arrays in the outer array.
4. When finished with the 2D array, first free the inner arrays and *then* free the outer array.

Let's write a function to create an *nxm* matrix with ones down the top-left to bottom-right diagonal. If $n = m$ then this is an identity matrix.

Creating a 2D array — code

```
1  int **identity_matrix(int n, int m) {
2    int **matrix = malloc(n*sizeof(int*));
3    for (size_t i = 0; i < n; ++i) {
4      int *row = malloc(m*sizeof(int));
5      for (size_t j = 0; j < m; ++j) {
6        row[j] = i == j ? 1 : 0;
7      }
8      matrix[i] = row;
9    }
10   return matrix;
11 }
```

Using a 2D array — code

```
1   int main() {
2     int **matrix = identity_matrix(3,3);
3     for (size_t i = 0; i < 3; ++i) {
4       printf("|");
5       for (size_t j = 0; j < 3; ++j) {
6         printf(" %d", matrix[i][j]);
7       }
8       printf(" |\n");
9     }
10    free(matrix);
11  }
```

Using a 2D array — code

```
1   int main() {
2     int **matrix = identity_matrix(3,3);
3     for (size_t i = 0; i < 3; ++i) {
4       printf("|");
5       for (size_t j = 0; j < 3; ++j) {
6         printf(" %d", matrix[i][j]);
7       }
8       printf(" |\n");
9     }
10    free(matrix); // LEAK!
11  }
```

This code leaks because we have freed `matrix`, but `matrix` was a pointer to our array of pointers. Freeing `matrix` loses us all pointers we had to our inner arrays, so we can no longer free them.

Using a 2D array — Fixed

```
1   int main() {
2     int **matrix = identity_matrix(3,3);
3     for (size_t i = 0; i < 3; ++i) {
4       printf("|");
5       for (size_t j = 0; j < 3; ++j) {
6         printf(" %d", matrix[i][j]);
7       }
8       printf(" |\n");
9     }
10    for (size_t i = 0; i < 3; ++i){
11      free(matrix[i]);
12    }
13    free(matrix);
14  }
```

Downsides of double indirection 2D arrays

There are performance losses from the *double indirection* of implementing a 2D array with pointers to pointers.

Since there is no guarantee about where the memory `malloc` gives you resides on the heap, each row of your 2D array could be spread out far across the heap, you also have the array of pointers to the rows to worry about.

Fetching memory from several different places can impact the performance of your CPU's cache. Additionally we are using more space than we really need due to having to store an entire array of only pointers to the rows.

Simulating a 2D array with a 1D array

If we want to avoid some of these costs we can by simulating a 2D array with a 1D array.

Observation: What is the difference between a 3×3 2D array of integers and 1D array of 9 integers?

Simulating a 2D array with a 1D array

If we want to avoid some of these costs we can by simulating a 2D array with a 1D array.

Observation: What is the difference between a 3×3 2D array of integers and 1D array of 9 integers?

Nothing more than perspective.

identity_matrix with 1D array

```
1       int *identity_matrix(int n, int m) {
2         int *matrix = malloc(n*m*sizeof(int));
3         for (size_t i = 0; i < n; ++i) {
4           for (size_t j = 0; j < n; ++j) {
5             matrix[i*m+j] = i == j ? 1 : 0;
6           }
7         }
8         return matrix;
9       }
```

Using our simulated 2D array

```
1   int main() {
2     int *id3 = identity_matrix(3, 3);
3     for (size_t i = 0; i < 3; ++i) {
4       printf("|");
5       for (size_t j = 0; j < 3; ++j) {
6         printf(" %d", id3[i*3+j]);
7       }
8       printf(" |\n");
9     }
10    free(id3);
11  }
```