COMP1511 Tutorial 7

pointers | struct pointers | command line args

pointers

Pointer Operations

`int *ptr` - Declare an integer pointer called `ptr`

`&num` - Give me the address of variable `num`

`*ptr` - Give me the variable at the address stored in `ptr` (dereferencing)

Pointers: declaring vs dereferencing

The asterisk (*) is used for 2 key pointer operations:

Declaring, e.g. `int *ptr;`

Dereferencing, e.g. `*ptr = 5;`

Rule of thumb: if the asterisk has a variable type before it (e.g. `int *`, `char *`) it's a pointer declaration, otherwise it's dereferencing a pointer

struct pointers

Do I use `.` or `->`?

With a regular struct variable (e.g. a `struct person` named `student1`), we would the `.` (dot) operator to access a field (e.g. `student1.name`)

If we had a pointer to a struct, we would need to dereference the pointer to get to the struct and then access its field: `(*student1_pointer).name`

We use the `->` operator to make this neater: `student1_pointer->name`

command line arguments

Command Line Arguments

`int argc` - The number of command line arguments

`char *argv[]` - The array of command line args (array of strings)

```
./program these are some args
```

```
argc = 5
```

```
argv = {“./program”, “these”, “are”, “some”, “args”}
```

Command Line Arg Exercises

Sum of Command Line Arguments: Write a C program that takes multiple integers as command-line arguments and prints their sum.

Count Characters in Command Line Arguments: Write a C program that counts the total number of characters in all the command-line arguments passed to it.

Reverse Command Line Arguments: Write a C program that prints all the command-line arguments passed to it in reverse order.

Check for Command Line Arguments: Write a C program that checks if any command-line arguments were provided except for the program name. If none were provided, print a message indicating so; otherwise, print the number of arguments.