Multidimensional Arrays, Command-Line Arguments, and Functions as Arguments

CSE 130: Introduction to Programming in C

Stony Brook University

Multidimensional Arrays

- C lets us create arrays of any type
  - This includes arrays of arrays
- Just add a new bracket pair for each additional dimension: `int b[2][7]` creates a 14-element array
- Array elements are stored contiguously in memory starting at the base address of the array

Two-Dimensional Array Example

- `int a[3][5];`
- `a[i][j]` is equivalent to:
  - `*(a[i] + j)`
  - `(*(a + i))[j]`
  - `*((*(a + i)) + j)`
  - `*(&a[0][0] + 5*i + j)`

Formal Parameter Declarations

- When passing a multidimensional array to a function, all sizes except the first must be specified
  - e.g., `int sum(int a[][5])`
- The following three versions are equivalent:

```
int a[][5]    int a[3][5]    int (*a)[5]
```

Initializing Multidimensional Arrays

```
int a[2][3] = {1, 2, 3, 4, 5, 6};

int a[2][3] = {{1, 2, 3}, {4, 5, 6}}

int a[ ][3] = {{1, 2, 3}, {4, 5, 6}}
```

Arguments to `main()`

- To pass command-line arguments to `main()`, we need to use arrays of pointers
  - Ex. *gcc <filename>*
- `main()` can take two arguments:
  - `argc` (an int)
  - `argv[]` (an array of char *)

`argc` and `argv`

- `argc` provides the number of command-line arguments
- `argc` is always at least 1 (the program name)
- `argv` contains all of the command-line arguments
- `argv[0]` contains the name of the original command

An Example

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

Sample Output

Input: `my_echo`

Output:

```
argc = 1
argv[0] = my_echo
```

More Sample Output

Input: `my_echo try this`

Output:

```
argc = 3
argv[0] = my_echo
argv[1] = try
argv[2] = this
```

More on `argv`

- `argv` could also have been declared as:

```
char **argv;
```

- This is a pointer to pointer to `char`
- Alternately, `argv` is an array of pointers to `char` (an array of strings)
- The system allocates space for `argv`

Functions as Arguments

- Pointers to functions can be passed as function arguments, used in arrays, and returned from functions
- We just need to specify the function's arguments and return type
  - The compiler interprets it as a pointer

```
double my_function(double f(double x), int m, int n)
{
  ...
  int value = f(m);
  ...
}
```

- The passed function's parameter name (`x`) is not strictly necessary
- We can also make the pointer nature of the function explicit:

```
double my_function(double (*f)(double), int m, int n)
```

Pointers and `const`

- Remember that we can use `const` to *qualify* a given variable as a constant
- An unqualified pointer may ***NOT*** be assigned the address of a `const`-qualified variable
  - We might accidentally try to change the value of the variable that the pointer points to

More on Pointers and `const`

- To declare that a pointer points to a constant value, add `const` to its declaration:

```
const int a = 7;
const int *p = &a; /* p points to a constant
                      int */
```

- To make the pointer itself constant:

```
int a;
int * const p = &a; /* p can't be changed, but
                       *p can */
```

The Last Straw

- Finally, consider:

```
const int a = 7;
const int * const p = &a;
```

- `p` is a constant pointer to a constant `int`
- Neither `p` nor `*p` can be assigned to, incremented, or decremented