Faculty of Computer Science, Dalhousie University *5-Nov-2018*

CSCI 2132 — Software Development

Lecture 26: Writing Large Programs

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- A common mistake with VLA declarations
- More string reading examples
- Buffer overflow risks
- String library functions

20.5 Command-Line Arguments in C

Using what we learned about strings, we can now process command-line arguments of a program. In C, these are passed as an array of strings. To make use of command-line arguments, we should define the main function in the following form:

```
int main(int argc, char* argv[]) {
...
}
```

Here, `argc` means "argument count" and `argv` means "argument values". Thus, `argc` stores the number of arguments in the command line, while `argv` stores an array of `char` pointers, and each pointer points to a string that stores an argument. the first argument `argv[0]` points to the string that is the command name as executed, usually from the shell.

Let us explain this in greater detail using an example. Here, we write a program called `sortwords`, which sorts the command-line arguments lexicographically, and prints them in sorted order. Let us say that we invoke this program using the following command:

```
./sortwords orange apple banana
```

The program should produce the output

```
apple
banana
orange
```

The following figure shows the content of argv:

```
argv
argv[0] -------> ./sortwords\0
argv[1] -------> orange\0
argv[2] -------> apple\0
argv[3] -------> banana\0
argv[4] stores a NULL pointer
```

Here a NULL pointer is a special pointer that points to nothing. In this example, we have `argc = 4`, which is the number of command-line arguments, including the pathname of the command itself (`"./sortwords"`).

We can use array/string notation that we learned before to manipulate this array. For example, `argv[2][4]` stores the character `'e'`.

For the implementation of sortwords, see:

```
~prof2132/public/sortwords.c
```

Our implementation uses insertion sort. Pay attention to two things: First, the subscripts we use in our code. Here we sort `argv[1]`, `argv[2]`, ..., `argv[argc-1]`, not the entire `argv` array. Second, we use `strcmp` to compare strings.

The code is also included here:

```
/* Program: sortwords.c */
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
  int i, j;
  char *key;

  for (i = 2; i < argc; i++) {
    key = argv[i];

    j = i-1;
    while (j >= ______ && strcmp(argv[j], key) ______ 0) {
      argv[j+1] = argv[j];
      j--;
    }

    argv[j+1] = key;
  }


  for (i = _______; i < _________ ; i++) {
    puts(argv[i]);
  }

  return 0;
}
```

We will see more examples on strings when we learn how to allocate strings dynamically.

21 Writing Large Programs

Previously we learned that the primary goal of this course is to learn how to "program in the large". A large program consists of many modules, and programmers work on different modules of the same program. It is logical to use one or more files for each module, which facilitates collaboration and reusing.

Header Files

- Files that allow different source files (`*.c`) to share
  - Function prototypes
  - Type definitions
  - Macro definitions
  - etc.
- Naming convention: `*.h`

The `#include` Directive

- Tells the preprocessor to open a specified file and inserts its content into the current file
- Form 1: `#include <file_name>`
  - Search the directories in which system header files reside
  - On bluenose: `/usr/include`, ...
- Form 2: `#include "file_name"`
  - First search the current directory, if not found then
  - directories in which system header files reside
- Question: Which form for your own header files?