Week 3

This is CS50. Harvard University. Fall 2014.

Cheng Gong

Table of Contents

Command-Line Arguments

- We've been writing programs that look like this, whereby `main` does not take any arguments (as implied by the presence of `void`):

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // TODO
}
```

  # This means that no other words can be typed after the program's name and accessed within `main`, and the only way to provide input is by a function running after the program is started, like with `GetString`.

- Commands like `cd` take arguments from the command-line, so `cd Dropbox` changes the directory to `Dropbox` without a separate prompt for input. `mkdir pset2` makes a directory called `pset2`, and `make hello` builds a program called `hello`.

- So far we've had `int main(void)` at the beginning of our programs, signifying that they don't take any arguments. Starting today, though, we'll start supporting even multiple command-line arguments.
  - `clang -o hello hello.c` has three such arguments (`-o`, `hello`, and `hello.c`).
- We will start adding code like this:

```
#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    // TODO
}
```

- We see that `main` now takes two arguments, `argc` which is an `int`, and `argv` which is an array of strings. We specify the name, `argv` (short for argument vector, or array of arguments), but not the size, so any array can be passed in to `main.`
- So command-line arguments look like this:

```
argc
-----------
|          |
-----------

argv
------------------------------------------------
|          |           |           |           |  ...
------------------------------------------------
```

- `argv` is a chunk of memory that stores one `string` after another, and `argc` is a single chunk of memory that holds an `int`.
- We can access each string individually:

```
argc
-----------
|          |
-----------
```

```
argv[0]    argv[1]     argv[2]     argv[3]
-------------------------------------------------
|          |           |           |           |  ...
-------------------------------------------------
```

- If we run a program with `./hello`, the computer will store this in `argc` and `argv[0]`:

```
argc
------------
| 1        |
------------

argv[0]     argv[1]     argv[2]     argv[3]
-------------------------------------------------
| ./hello   |           |           |           |  ...
-------------------------------------------------
```

- If we run `cd Dropbox`, the memory will look like this:

```
argc
------------
| 2        |
------------

argv[0]     argv[1]     argv[2]     argv[3]
-------------------------------------------------
| cd        | Dropbox   |           |           |  ...
-------------------------------------------------
```

- And likewise for `mkdir pset2`:

```
argc
------------
| 2        |
------------

argv[0]     argv[1]     argv[2]     argv[3]
-------------------------------------------------
| mkdir     | pset2     |           |           |  ...
-------------------------------------------------
```

- If we ran `clang -o hello hello.c`, however, we get:

```
argc
------------
| 4        |
------------

argv[0]     argv[1]     argv[2]     argv[3]
--------------------------------------------------
| clang     | -o        | hello     | hello.c   |  ...
--------------------------------------------------
```

- Since we don't know where `argv` will end by itself, we need `argc` to tell us where to stop looking.
- Let's write a program that uses these arguments. What about a program that says `hello` without using `GetString`? Instead, it'll take arguments like this:

```
argc
------------
| 2        |
------------

argv[0]     argv[1]     argv[2]     argv[3]
--------------------------------------------------
| ./hello   | Zamyla    |           |           |  ...
--------------------------------------------------
```

- We'll call this `hello-3.c` [1]:

```
#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    printf("hello, %s\n", argv[1]);
}
```

# `argv[1]` contains whatever `string` is passed in after the name of our program.

[1] http://cdn.cs50.net/2014/fall/lectures/3/m/src3m/hello-3.c

Memory Access

- But what happens if we don't type someone's name in?

```
jharvard@appliance (~/Dropbox): make hello-3
clang -ggdb3 -O0 -std=c99 -Wall -Werror    hello-3.c  -lcs50 -lm -o
 hello-3
jharvard@appliance (~/Dropbox): ./hello-3
hello, (null)
```

  - `printf` is printing `(null)` because there's nothing (well, technically, `NULL`) in `argv[1]`.
    - What about poking around even more? Let's try to access `argv[2]`:

```
#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    printf("hello, %s\n", argv[2]);
}
```

```
jharvard@appliance (~/Desktop): ./hello-3
hello, XDG_VTNR=7
```

    - What about `argv[3]`?

```
jharvard@appliance (~/Desktop): ./hello-3
hello, SSH_AGENT_PID=1616
```

    - Interesting. How about `argv[50]`?

```
jharvard@appliance (~/Desktop): ./hello-3
hello, INSTANCE=
```

    - Let's be reckless, and try `argv[5000]`:

```
jharvard@appliance (~/Desktop): ./hello-3
Segmentation fault (core dumped)
```

  - This message might look familiar. We've caused a **segmentation fault** where we've touched memory we shouldn't have, and segments are just chunks of memory. The computer gives us the number of arguments in `argv`, and if we try to go farther there is physically other memory there and data stored, so we might have seen anything else. Indeed, it might even be possible for hackers to use programs in a powerful language like C to go through memory and find passwords or other interesting things.
    - We should have gotten errors as soon as we went even one index past the end of `argv`, but since a computer allocates memory in chunks for efficiency, our program got a big enough chunk to look at `argv[3]` and `argv[50]` without going past its boundaries. Once we tried to access `argv[5000]`, the operating system noticed and enforced the segmentation of memory.
    - Another note, `core dumped` means that the program's memory when it crashed is saved as a file called `core`, so we can use it diagnostically after the fact.
  - In this case, however, we went too far and the program crashed. Let's try to fix this.

Return Values

- Starting over, we notice that `main` is a function that returns an `int`:

```
#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    // TODO
}
```

  - Even though we haven't specified `return 0`, the compiler has been putting that in for you. In the case of `main`, returning a `0` signifies successful completion, whereas any non-zero return value signifies an error.
- So let's look at `hello-4.c`[2]:

[2] http://cdn.cs50.net/2014/fall/lectures/3/m/src3m/hello-4.c

```
#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    if (argc == 2)
    {
        printf("hello, %s\n", argv[1]);
        return 0;
    }
    else
    {
        return 1;
    }
}
```

# On line 6 we make sure that `argc` has a value of `2`, and `return 0` on line 9 if all went well. Otherwise, we `return 1` on line 13 if something went wrong.

# Now if we pass in any number of arguments beside 2, the program will exit, and though the user can't see the return value of `1`, we will later use a debugger that can:

```
jharvard@appliance (~/Desktop): ./hello-4 Zamyla
hello, Zamyla
jharvard@appliance (~/Desktop): ./hello-4
jharvard@appliance (~/Desktop): ./hello-4 Rob is a proctor in Thayer
jharvard@appliance (~/Desktop):
```

- Also, `string` is the same as `char*`, but we'll remove those training wheels soon and see what they actually are.
- And the computer knows the value of `argc` because the blinking prompt is a program in itself that, working with the operating system, takes the words typed, and figures out `argc` and `argv` for you.
- We can name `argc` and `argv` in our `main` function to be anything, but by convention they are what they are.

More on argv

- Let's look a little more closely at `argv`:

```
argv[0]     argv[1]
-------------------------
| ./hello   | Zamyla    |  ...
-------------------------
```

- Remember that a string is an array of characters, so we should really draw this picture:

```
argv[0]                 argv[1]
---------------------------------------------
|  ------------------  |  ----------------  |
|  |.|/|h|e|l|l|o|\0|  |  |Z|a|m|y|l|a|\0|  |  ...
|  ------------------  |  ----------------  |
---------------------------------------------
```

  - Indeed, the little white boxes of each string are actually outside of the bigger boxes, but we won't worry about that for now.

- Let's look at `argv-1.c`[3]:

```
#include <cs50.h>
#include <stdio.h>

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

  - This will print each argument, one per line:

```
jharvard@appliance (~/Dropbox/src3m): make argv-1
clang -ggdb3 -O0 -std=c99 -Wall -Werror    argv-1.c  -lcs50 -lm -o
 argv-1
```

[3] http://cdn.cs50.net/2014/fall/lectures/3/m/src3m/argv-1.c

```
jharvard@appliance (~/Dropbox/src3m): ./argv-1
./argv-1
jharvard@appliance (~/Dropbox/src3m): ./argv-1 foo
./argv-1
foo
jharvard@appliance (~/Dropbox/src3m): ./argv-1 foo bar
./argv-1
foo
bar
```

- We can take this further in `argv-2.c`[4]:

```
#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(int argc, string argv[])
{
    // print arguments
    for (int i = 0; i < argc; i++)
    {
        for (int j = 0, n = strlen(argv[i]); j < n; j++)
        {
            printf("%c\n", argv[i][j]);
        }
        printf("\n");
    }
}
```

  # Now we go through each argument with line 8, but in line 10 we check the length of the argument stored in `argv[i]`, store it in `n`, and use `j` as a counter to iterate through `argv[i]` since `i` was already used. Then in line 12 we use this new syntax, `argv[i][j]` that gets the i'th string and the j'th character in that string, and indeed this works as we expect:

```
jharvard@appliance (~/Desktop): make argv-2
clang -ggdb3 -O0 -std=c99 -Wall -Werror    argv-2.c  -lcs50 -lm -o
 argv-2
jharvard@appliance (~/Desktop): ./argv-2
.
```

[4] http://cdn.cs50.net/2014/fall/lectures/3/m/src3m/argv-2.c

```
/
a
r
g
v
-
2

jharvard@appliance (~/Desktop): ./argv-2 foo bar
.
/
a
r
g
v
-
2

f
o
o

b
a
r
```

Sorting

- So we’re just taking the same ideas and applying them. If we think about a phone book as an array with pages in each box, we can start to see a solution in another way.
- A volunteer from the audience, Ajay, looks for the number 50 among 7 doors on the screen, and luckily finds it on the first try. The numbers were in random order, so the best we could have done, if we didn’t get lucky, is continue searching randomly until we find it.
- We have another set of doors, this time sorted, and Ajay finds it luckily again. (After class, we sent Ajay to buy a lottery ticket on behalf of CS50, but haven’t heard from him since …)

- Time to play a clip[5] from the past of this same demonstration, with Sean who wasn't quite as lucky.
- With the second set of doors, that are sorted, we can do much better. Much like the phonebook example, we can start in the middle, and divide the problem in half by moving in the direction we expect the number we want to be.

Bubble Sort

- Let's take a closer look at sorting with 8 volunteers from the audience. We line them up with numbers on stage in this order:

```
4     2     6     8     1     3     7     5
```

- Let's make the first swap at the beginning, since 4 and 2 were out of order:

```
2     4     6     8     1     3     7     5
```

- Continuing by going down and comparing numbers one pair at a time, swapping them as necessary:

```
2     4     6     1     8     3     7     5
2     4     6     1     3     8     7     5
2     4     6     1     3     7     8     5
2     4     6     1     3     7     5     8
```

- So 8 has bubbled up to the end and we're getting closer to the solution. Let's run this again:

```
2     4     6     1     3     7     5     8
2     4     1     6     3     7     5     8
2     4     1     3     6     7     5     8
2     4     1     3     6     5     7     8
```

- Now both 7 and 8 are correct. One more time:

```
2     1     4     3     6     5     7     8
2     1     3     4     6     5     7     8
2     1     3     4     5     6     7     8
```

[5] http://youtu.be/nEdFfBbmlp4?t=26m24s

- And again:

```
2     1     3     4     5     6     7     8
1     2     3     4     5     6     7     8
```

- How many steps did we take to sort these numbers? It may have seemed inefficient, because it was, so we take a loot at other algorithms.
- This online demo[6] shows various sorting algorithms, in particular bubble sort, moving bars in order of length. The longer bars "bubble" to the right, and the shorter bars "bubble" to the left.

Selection Sort

- Let's try another algorithm, where we keep looking for the smallest element, and switch it (in constant time) with the first unsorted element:

```
4     2     6     8     1     3     7     5
1     2     6     8     4     3     7     5    // 1 is moved toward the front
1     2     3     8     4     6     7     5    // 2 is moved toward the front
1     2     3     4     8     6     7     5    // 3 is moved toward the front
1     2     3     4     5     6     7     8    // 4 is moved toward the front
```

# This is called selection sort, and we see how it sorted differently again with this online demo[7].

Insertion Sort

- Let's reset one more time:

```
4     2     6     8     1     3     7     5
```

- We take the first element, declare it to be sorted, and move the next one to its correct placement within the sorted part of the array.

```
4     2     6     8     1     3     7     5    // 4 is sorted
2     4     6     8     1     3     7     5    // we move 2 to in front of 4
```

[6] http://cs.smith.edu/~thiebaut/java/sort/demo.html
[7] http://cs.smith.edu/~thiebaut/java/sort/demo.html

```
2    4    6    8    1    3    7    5    // 6 is sorted
2    4    6    8    1    3    7    5    // 8 is sorted
1    2    4    6    8    3    7    5    // we move 1 to the beginning
1    2    3    4    6    8    7    5    // we move 3 to its location
1    2    3    4    6    7    8    5    // we move 7 to its location
1    2    3    4    5    6    7    8    // we move 5 to its location
```

# And the demo has another "feel" with little gaps that we fix over time.

- It turns out that all of these algorithms are fundamentally equivalent in efficiency once *n* (the number of items to sort) is sufficiently large.
- Finally, we watch what different sorting algorithms sound like[8] to get a feel for them in another way.

[8] http://youtu.be/t8g-iYGHpEA