Week 2, continued

This is CS50. Harvard University. Fall 2015.

Anna Whitney

Table of Contents

1. Introduction 1
2. More on Strings 1
3. Arrays 4
4. Command-Line Arguments 6
5. Cryptography 13

1. Introduction

- Rob Bowden teaches today, and makes fun of David for sending a vertical video to apologize for missing today's lecture.
- Today we'll talk more about data representation and cryptography, or scrambling information, but first a story from yesteryear.
  - Radio Orphan Annie's Secret Decoder Ring is a child-friendly form of cryptography, with two discs that rotates independently, the outer ring containing the letters `A` – `Z` and the inner ring the numbers `1` – `26` to encode a message by mapping letters to numbers.
  - This clip[1] from A Christmas Story[2] shows a child, Ralphie, excitedly decoding the secret message from the radio, only to find that it is an advertisement for Ovaltine, a beverage popular many years ago.

2. More on Strings

- Recall from Monday the following representation of a string containing Zamyla's name, with the individual \`char\`s split up into separate boxes.

[1] http://www.youtu.be/kEAH6u1ODNI?t=1m36s
[2] http://en.wikipedia.org/wiki/A_Christmas_Story

```
-------------------------
| Z | a | m | y | l | a |
-------------------------
```

- We can access these individual characters using bracket notation, so if we assign this string to a variable called `s`, we can get the first character using `s[0]`, the second with `s[1]`, and so on up to `s[5]` (the final `a`).
- This string is of length `6`, but positions in a string are 0-indexed, so it contains indices `0` through `5`.

• Now let's look at this string in a larger context of your computer's memory:

```
---------------------------------
| Z | a | m | y | l | a |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
```

- The computer's memory is basically one long string of bytes, but we'll represent it as a grid to save space.

• A volunteer from the audience acts as a computer implementing the following code and storing the strings in memory:

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

int main(void)
{
    // get four strings from the user
    string s1 = GetString();
    string s2 = GetString();
    string s3 = GetString();
    string s4 = GetString();

    // print the first string entered
    printf("string s1 is %s\n", s1);
}
```

Rob, as the user, provides the strings `DEAN`, `HANNAH`, `MARIA`, and `ROB`, and our volunteer fills in the computer's memory like so:

```
-------------------------
|...| D | E | A | N |   |
-------------------------
| H | A | N | N | A | H |
-------------------------
|   | M | A | R | I | A |
-------------------------
|   | R | O | B |   |...|
-------------------------
```

Our volunteer left spaces between each string so the computer can tell where each string ends (otherwise, when we try to print just `s1`, we would get `DEANHANNAHMARIAROB` rather than just `DEAN`!).

Rather than spaces, the computer actually uses a special terminator character, `\0`, to represent the end of a string:

```
-------------------------
|...| D | E | A | N |\0 |
-------------------------
| H | A | N | N | A | H |
-------------------------
|\0 | M | A | R | I | A |
```

```
-------------------------
|\0 | R | O | B |\0 |...|
-------------------------
```

- The `strlen` function also relies on the presence of `\0`—it just iterates over the characters of the string until it finds a `\0`.
- In memory, the character `\0` is actually represented by a byte of all `0`s (so 8 consecutive 0 bits). So what about the character `0`? Remember ASCII, the system that maps characters to underlying byte values, where `A` maps to `65` and so on; the number `0` is represented by ASCII `48`.
- We'll refer to the `\0` character as a **null-terminator** (aka **NUL**).

3. Arrays

- So this general idea of storing items in boxes, as we do under the hood in a string, is known as an **array**. An array is a type of **data structure**, with a continguous number of the same type of data, back-to-back. A `string` is just an array of `char` variables, but we can put any of our other data types in an array as well.
- Now say we wanted to get the ages of a number of people in the room. We might start with:

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

int main(void)
{
    int age0 = GetInt();
    int age1 = GetInt();
    int age2 = GetInt();

    // do something with those numbers ...
}
```

- But this will force us to completely rewrite our code, and copy-paste that `GetInt` line for every person we want to get an age from.
- We can solve this problem by using an array. The general format for declaring an array of a given `type` and `size` as a variable called `name` is:

```
type name[size];
```

- Let's look at how we do this in `ages.c` [3]:

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

int main(void)
{
    // determine number of people
    int n;
    do
    {
        printf("Number of people in room: ");
        n = GetInt();
    }
    while (n < 1);

    // declare array in which to store everyone's age
    int ages[n];

    // get everyone's age
    for (int i = 0; i < n; i++)
    {
        printf("Age of person #%i: ", i + 1);
        ages[i] = GetInt();
    }

    // report everyone's age a year hence
    printf("Time passes...\n");
    for (int i = 0; i < n; i++)
    {
        printf("A year from now, person #%i will be %i years old.\n", i
 + 1, ages[i] + 1);
    }
}
```

[3] http://cdn.cs50.net/2015/fall/lectures/2/w/src2w/ages.c

  - In line 16, we declare an array that stores exactly `n` integers. The number in this case is how big we want the array to be, whereas earlier when we used `s[i]` we were retrieving that particular item in the array since it was already declared.
    - This means that somewhere in memory, we have `n` integer-sized (32-bit, or 4-byte) boxes in a row.
  - Then we `GetInt` for each person, storing it in `ages[i]` as we go through the loop, meaning the ages will be placed in the first box, second box, and so on of the `ages[]` array.
    - Now it should make a little more sense why the convention is to start `for` loops from `0` rather than `1`—we very often use a `for` loop to iterate over the indices of an array, and arrays are 0-indexed.
  - Finally, we iterate through the array again and print out each age, with 1 added to demonstrate what we can do after we retrieve the `int` from the array.
  - What happens if we try to store another `int` in `ages[n+1]`? Just as we saw when we tried to read many bytes past the end of a string, this results in a **segmentation fault**, an error that indicates we've touched a segment of memory that doesn't belong to us.
  - What if we didn't check whether the user entered a positive number for the number of people in the room? An array cannot have a negative size—if you declare an array with a negative size directly in your code, it will not compile—so if the user gets away with entering a negative size for the array, it also results in a segmentation fault.

4. Command-Line Arguments

- Now that we've seen arrays, we can start to work with command-line arguments, which we'll need for Problem Set 2.
- Commands like `cd` don't ask for input; instead they take arguments from the command-line, so `cd pset1` changes the directory to `pset1` without a separate prompt for input. `mkdir pset2` makes a directory called `pset2`, and `make hello` builds a program called `hello`.
  - `clang -o hello hello.c` has three such arguments (`-o`, `hello`, and `hello.c`).

- 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`.

- We will start adding code like this:

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

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

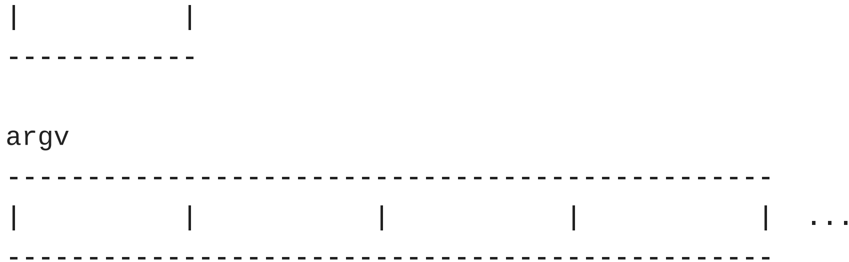
- We see that `main` now takes two arguments, `argc` which is an `int`, and `argv` which is an array of strings. We specify the name, `argv` (short for argument vector, or array of arguments), but not the size, so any array can be passed in to `main.`

  # This is a slightly different use of the bracket notation we haven't seen before—rather than indicating the size of an array or a position within an array, these empty brackets just mean that `argv` is an array (of unspecified size), and we know it's an array of strings because it's declared as `string argv[]`.

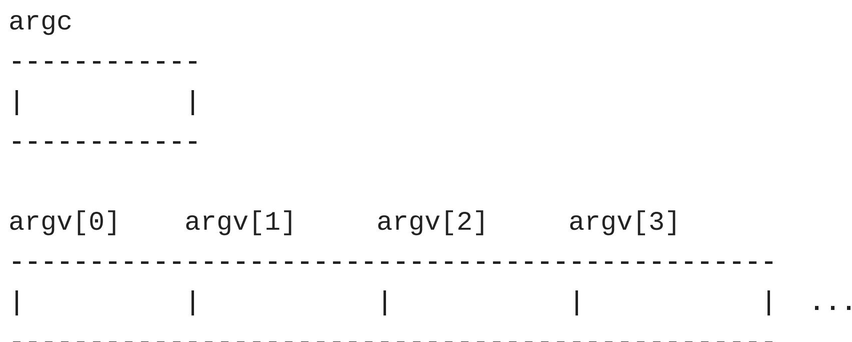
  # We need the size of `argv` to be unspecified because many programs (such as `clang`) can take different numbers of command-line arguments depending on what you're trying to do with them.

- So command-line arguments look like this in memory:

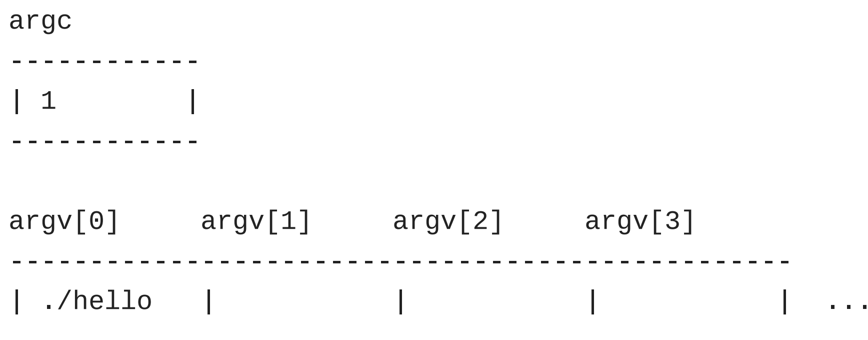
```
argc
------------
```



- `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:



- If we run a program with `./hello`, the contents of `argc` and `argv[0]` would be as follows:



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

```
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`[4]:

```
#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.

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

```
jharvard@ide50:~/workspace/src2w $ make hello-3
clang -ggdb3 -O0 -std=c99 -Wall -Werror    hello-3.c  -lcs50 -lm -o
 hello-3
jharvard@ide50:~/workspace/src2w $ ./hello-3
hello, (null)
```

[4] http://cdn.cs50.net/2015/fall/lectures/2/w/src2w/hello-3.c

`printf` is printing `(null)` because there's nothing (well, technically, `NULL`) in `argv[1]`.

When we run just `./hello-3`, `argc` is 1, so the length of the array `argv` is 1, so the only valid index in `argv` is `argv[0]`.

- So let's look at how we can prevent something like this happening in `hello-4.c`[5]:

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

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

On line 6 we make sure that `argc` has a value of `2`, and if so, we know we have a single name string to plug in and say hello to the user. If not, we instead substitute a generic message of "hello, you".

```
jharvard@ide50:~/workspace/src2w $ ./hello-4
hello, you
jharvard@ide50:~/workspace/src2w $ ./hello-4 Rob
hello, Rob
jharvard@ide50:~/workspace/src2w $ ./hello-4 Rob Maria
hello, you
jharvard@ide50:~/workspace/src2w $
```

- Let's look at `argv-1.c`[6]:

[5] http://cdn.cs50.net/2015/fall/lectures/2/w/src2w/hello-4.c
[6] http://cdn.cs50.net/2015/fall/lectures/2/w/src2w/argv-1.c

```
#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]);
    }
}
```

We're iterating over the `argv` array, using its length, `argc`, to know when to stop.

This will print each argument, one per line:

```
jharvard@ide50:~/workspace/src2w $ make argv-1
clang -ggdb3 -O0 -std=c99 -Wall -Werror    argv-1.c  -lcs50 -lm -o
 argv-1
jharvard@ide50:~/workspace/src2w $ ./argv-1
./argv-1
jharvard@ide50:~/workspace/src2w $ ./argv-1 Rob
./argv-1
Rob
jharvard@ide50:~/workspace/src2w $ ./argv-1 Rob Maria Hannah
./argv-1
Rob
Maria
Hannah
```

Note that `argc` can never be less than 1, because there will always at least be the name of the program itself.

- We can take this further in `argv-2.c`[7]:

[7] http://cdn.cs50.net/2015/fall/lectures/2/w/src2w/argv-2.c

```
#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. This relies on the fact that `argv` is an array of strings, which are themselves arrays of characters, so `argv` is a nested array of arrays.

```
jharvard@ide50:~/workspace/src2w $ make argv-2
clang -ggdb3 -O0 -std=c99 -Wall -Werror    argv-2.c  -lcs50 -lm -o
 argv-2
jharvard@ide50:~/workspace/src2w $ ./argv-2
.
/
a
r
g
v
-
2

jharvard@ide50:~/workspace/src2w $ ./argv-2 foo bar
.
/
```

```
a
r
g
v
-
2

f
o
o

b
a
r
```

- One thing to note is that command-line arguments are generally separated by spaces, as you'd expect, but if you want to pass an argument that itself contains a space, you can do it as follows:

```
jharvard@ide50:~/workspace/src2w $ ./argv-2 Rob Maria 'Hannah Blumberg'
```

    # By enclosing `Hannah Blumberg` in quotes, we tell the program that it's not two separate arguments, but rather only one.

5. Cryptography

- In **Problem Set 2** we introduce you to cryptography, specifically **secret-key crypto**, which can only be decoded by someone who knows the secret key that was used to encode the message.
- In the Hacker Edition, we'll give you some usernames and encrypted (well, "hashed") passwords that look like `{crypt}$1$LlBcWwQn$pxTB3yAjbVS/HTD2xuXFI0`, challenging you to crack them and finding the original passwords.
- After this problem set, you'll be able to decode what this means:

```
or fher gb qevax lbhe Binygvar
```

- As well as this URL, which you may remember from Week 0:

```
uggcf://lbhgh.or/bUt5FWLEUN0
```