Week 2, continued

This is CS50. Harvard University. Fall 2014.

Cheng Gong

Table of Contents

Introduction

- Ask questions online at http://cs50.harvard.edu/discuss.
- 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, each with a set of letters, such that `A` might now map to `B`, `B` to something like `C`, etc.
    # 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.
- CS50 Lunch will again be Friday 1:15pm, RSVP at http://cs50.harvard.edu/rsvp.

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

Data Representation

Strings

- Let's say we want to represent a `string` for Zamyla's name. We can place each character, or `char`, in its own box:

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

- It's useful to think of strings as composed of `char` puzzle pieces since we can access individual characters more easily. Let's look at `string-0.c`[3]:

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

int main(void)
{
    string s = GetString();

    for (int i = 0; i < 6; i++)
    {
        printf("%c\n", s[i]); ❶
    }
}
```

- In line 10 we use `%c\n` to print each character on its own line, and to get each character, we use `s[i]`, as in the "box number" of the string `s`:

```
    ------------------------
 s: | Z | a | m | y | l | a |
    ------------------------
      0   1   2   3   4   5
```

- So `s[0]` would get us `Z`, `s[1]` `a`, and so on, and as `i` is increased by the `for` loop, we will move through the string.

[3] http://cdn.cs50.net/2014/fall/lectures/2/w/src2w/string-0.c

- But wait, this code won't compile, because we're missing header files. So let's add `stdio.h` for `printf`, and `cs50.h` to have `string` and `GetString`.

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

int main(void)
{
    string s = GetString();

    for (int i = 0; i < 6; i++)
    {
        printf("%c\n", s[i]);
    }
}
```

- Now let's run this program:

```
jharvard@appliance (~/Dropbox): ./string-0
Zamyla
Z
a
m
y
l
a
jharvard@appliance (~/Dropbox):
```

- Looks good. Let's try it again with Daven's full name:

```
jharvard@appliance (~/Dropbox): ./string-0
Davenport
D
a
v
e
n
p
jharvard@appliance (~/Dropbox):
```

- This bug happened because we put the 6 in the for loop, and we can actually determine the length of a string with `strlen`:

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

int main(void)
{
    string s = GetString();

    for (int i = 0; i < strlen(s); i++)
    {
        printf("%c\n", s[i]);
    }
}
```

- Now we get a compiler error:

```
jharvard@appliance (~/Dropbox): make string-0
clang -ggdb3 -O0 -std=c99 -Wall -Werror    string-0.c  -lcs50 -lm -o
 string-0
string-0.c:8:25: error: implicitly declaring library function 'strlen'
 with type 'unsigned int (const char *)'
      [-Werror]
    for (int i = 0; i < strlen(s); i++)
                        ^
string-0.c:8:25: note: please include the header <string.h> or explicitly
 provide a declaration for 'strlen'
1 error generated.
make: *** [string-0] Error 1
jharvard@appliance (~/Dropbox):
```

- We focus on `implicitly declaring library function`, which tells us we need to find the `strlen` function in another library.
- If we wanted to find out which library has this function, we can go into the terminal and run `man` for manual:

```
jharvard@appliance (~): man strlen
...
STRLEN(3)                  Linux Programmer's Manual
 STRLEN(3)

NAME
       strlen - calculate the length of a string
```

```
SYNOPSIS
       #include <string.h>
...
```

- So now we know to include `string.h`, and compiling and running now seem to work.

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

int main(void)
{
    string s = GetString();

    for (int i = 0; i < strlen(s); i++)
    {
        printf("%c\n", s[i]);
    }
}
```

- We've been taking for granted that our laptop, and the CS50 Appliance, has a large amount of memory, but if we type for long enough, we can type more characters than we have memory, much like integers running out of bits.

  # So we need to anticipate this problem by making sure `s` has some value that was indeed returned by `GetString`. If something goes wrong, `GetString` will return a special value, `NULL`, as in there is no value. We check if `s` is `NULL` before we use it, in `string-1.c`[4]:

[4] http://cdn.cs50.net/2014/fall/lectures/2/w/src2w/string-1.c

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

int main(void)
{
    string s = GetString();

    if (s != NULL)
    {
        for (int i = 0; i < strlen(s); i++)
        {
            printf("%c\n", s[i]);
        }
    }
}
```

`!=` means not equal, so we only print the characters in `s` if there is a string in `s`.

- Let's look at `string-2.c`[5]:

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

int main(void)
{
    // get line of text
    string s = GetString();

    // print string, one character per line
    if (s != NULL)
    {
        for (int i = 0; i < strlen(s); i++)
        {
            printf("%c\n", s[i]);
        }
    }
}
```

[5] http://cdn.cs50.net/2014/fall/lectures/2/w/src2w/string-2.c

    # Remember in line 13 we initialize an `int i = 0` and increment it by `i++` every time. The middle part checks that `i < strlen(s)` is true in order to continue the loop, meaning we keep printing the character for the length of the string. But every time `i` is changing while `s` is not, so every time we are calculating the length of `s` over and over again unnecessarily. We can do a little better:

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

int main(void)
{
    // get line of text
    string s = GetString();

    // print string, one character per line
    if (s != NULL)
    {
        for (int i = 0, n = strlen(s); i < n; i++)
        {
            printf("%c\n", s[i]);
        }
    }
}
```

        # Now we initialize two variables, `i` and `n`, with `n` holding the length of the string. Though this version is equally as correct as the first, it is better design as we don't need to answer the same question multiple times, and thus improved its efficiency.

        # Note that we don't have to say `int n` since it's the same type as `i` and it is in the same statement.

- As an aside, there is no functional difference in this particular context between `i++` and `++i`, but `i++` is more clear that `i` is being incremented. Alternatively, you can write `i += 1`.

Typecasting

- **Typecasting** is the ability to convert one datatype to another. Recall that ASCII maps letters to numbers. Let's look at `ascii-0.c`[6]:

```
#include <stdio.h>

int main(void)
{
    // display mapping for uppercase letters
    for (int i = 65; i < 65 + 26; i++)
    {
        printf("%c: %i\n", (char) i, i);
    }

    // separate uppercase from lowercase
    printf("\n");

    // display mapping for lowercase letters
    for (int i = 97; i < 97 + 26; i++)
    {
        printf("%c: %i\n", (char) i, i);
    }
}
```

Let's go through this program. Line 6 runs 26 times, starting from `65` because `A` is `65` in ASCII, and in line 8 we print a `char` and an `int`. It turns out by using `(char) i` we can print `i` out as a char. The loop below, starting at `97`, prints the lowercase characters in a similar way:

```
jharvard@appliance (~/Dropbox/src2w): make ascii-0
clang -ggdb3 -O0 -std=c99 -Wall -Werror    ascii-0.c  -lcs50 -lm -o
 ascii-0
jharvard@appliance (~/Dropbox/src2w): ./ascii-0
A: 65
B: 66
C: 67
...
```

[6] http://cdn.cs50.net/2014/fall/lectures/2/w/src2w/ascii-0.c

```
X: 88
Y: 89
Z: 90

a: 97
b: 98
c: 99
...
x: 120
y: 121
z: 122
```

- Let's look at `ascii-1.c`[7]:

```
#include <stdio.h>

int main(void)
{
    // display mapping for uppercase letters
    for (char c = 'A'; c <= 'Z'; c++)
    {
        printf("%c: %i\n", c, (int) c);
    }

    // separate uppercase from lowercase
    printf("\n");

    // display mapping for lowercase letters
    for (char c = 'a'; c <= 'z'; c++)
    {
        printf("%c: %i\n", c, (int) c);
    }
}
```

We can compare `c` to a character directly in lines 6 and 15 since the underlying data is still stored in bits and the computer will just compare the numbers. We can also see that `char c` can be converted back to an `int` in lines 8 and 17.

- Let's look at `capitalize-0.c`[8]:

[7] http://cdn.cs50.net/2014/fall/lectures/2/w/src2w/ascii-1.c
[8] http://cdn.cs50.net/2014/fall/lectures/2/w/src2w/capitalize-0.c

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

int main(void)
{
    // get line of text
    string s = GetString();

    // capitalize text
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        if (s[i] >= 'a' && s[i] <= 'z')
        {
            printf("%c", s[i] - ('a' - 'A'));
        }
        else
        {
            printf("%c", s[i]);
        }
    }
    printf("\n");
}
```

First, we get a string from the user. In line 11 we iterate over the string character by character, storing the length of `s` in `n`. In line 13 we access each character, and determine if it's lowercase by comparing its value to the boundaries of the values of lowercase characters. In line 15, we notice that `a` is `97`, `A` is `65`, `b` is `98`, `B` is `66`, and so on, meaning the difference between lowercase and uppercase is a constant 32. So we subtract that difference from the lowercase `s[i]`, which then gives us an uppercase character.

Abstraction

- Now let's look at `capitalize-1.c`[9]:

[9] http://cdn.cs50.net/2014/fall/lectures/2/w/src2w/capitalize-1.c

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

int main(void)
{
    // get line of text
    string s = GetString();

    // capitalize text
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        if (islower(s[i]))
        {
            printf("%c", toupper(s[i]));
        }
        else
        {
            printf("%c", s[i]);
        }
    }
    printf("\n");
}
```

Here we can use a `toupper` function declared in `ctype.h` which we also included, and we call it by passing it `s[i]` within the parentheses. We also use `islower` to check if a character is lowercase. Notice that these functions were probably implemented with code similar to the previous example, but are nicely named and already exist for us to use. This follows along with the idea of abstracting away lower level details and using these functions to help us.

If we look at the man page for `toupper`, we see this:

```
...
RETURN VALUE
       The value returned is that of the converted letter, or c if the
 conversion was not possible.
...
```

    # So now we can improve the code in `capitalize-2.c`[10] by removing the `if` condition and allowing `toupper` to do the work:

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

int main(void)
{
    // get line of text
    string s = GetString();

    // capitalize text
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        printf("%c", toupper(s[i]));
    }
    printf("\n");
}
```

        # Remember that we should also make sure that `s` is not `NULL`, or have a `do-while` loop prompt the user for a string until an acceptable one is given.

- For functions in the various libraries, `stdio.h`, `cs50.h`, `string.h`, `ctype.h`, reference.cs50.net[11] has user-friendly explanations that are quite helpful.

Arrays

- A volunteer from the audience, Alex, writes `Zamyla` on the screen, simulating `GetString`. He then writes `Belinda`, and then `Gabe`. If we think about the screen as all the memory we have, we notice that Alex wrote them with some spacing between the names. A computer has a grid of memory as well:

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

[10] http://cdn.cs50.net/2014/fall/lectures/2/w/src2w/capitalize-2.c
[11] http://reference.cs50.net

```
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
```

- The computer wants to be efficient, and use as much as the memory as possible:

```
---------------------------------
| Z | a | m | y | l | a |\0 |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
```

- Strings end with a `\0` which, in binary, is eight 0s in a row. And this tells a computer that this is the end of a string in memory. So Belinda is added like so:

```
---------------------------------
| Z | a | m | y | l | a |\0 | B |
---------------------------------
| e | l | i | n | d | a |\0 |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
```

- And we can continue with even more names:

```
---------------------------------
| Z | a | m | y | l | a |\0 | B |
---------------------------------
| e | l | i | n | d | a |\0 | G |
---------------------------------
| a | b | e |\0 | D | a | v | e |
---------------------------------
| n |\0 |   |   |   |   |   |   |
---------------------------------
```

- So this general idea of storing items in boxes 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, and we can even put numbers in an array. Arrays will generally have this format:

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

- 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 age1 = GetInt();
    int age2 = GetInt();
    int age3 = GetInt();

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

- But this will limit us to only ever storing exactly 3 ages. So we can solve this problem with `ages.c` [12]:

[12] http://cdn.cs50.net/2014/fall/lectures/2/w/src2w/ages.c

```c
#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);
    }
}
```

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.

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.

    # 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.

Command-Line Arguments

- It turns out, instead of using `GetInt` or `GetString` to get input, we can use **command-line arguments** to get words typed into the blinking prompt, after your program's name. For example, you might type:

```
jharvard@appliance (~): ./caesar 13
...
jharvard@appliance (~): ./caesar 7
```

    # The number after `./caesar` is the command-line argument that can change every time you run the program, and the program will use it as input.

Cryptography

- In **Problem Set 2** we introduce you to cryptography, scrambling information. In particular, Caesar and Vigenere ciphers will rotate letters to another letter. Caesar adds a particular number to all the letters, while Vigenere adds a different number to each letter. In the end, you'll see that you use the same **key** to turn **plaintext** into **ciphertext**, and back to plaintext.

    # 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.

- A clip from Spaceballs[13] humorously has "the stupidest combination I've ever heard in my life": `12345`.

[13] http://youtu.be/GjcHmboOyrs?t=48m19s