Week 2

This is CS50. Harvard University. Fall 2014.

Cheng Gong

Table of Contents

Bugs

- A **bug** is a mistake in a program. Let's look at `buggy-0.c`[1]:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i <= 10; i++)
        printf("*");
}
```

# The bug here is called an **off-by-one error** since the start value for `i` is 0 and we continue while `i <= 10`, resulting in 11 things counted total.

- So we can change the code like this:

```
#include <stdio.h>

int main(void)
{
    for (int i = 1; i <= 10; i++)
        printf("*");
}
```

[1] http://cdn.cs50.net/2014/fall/lectures/2/m/src2m/buggy-0.c

- This is more straightforward for humans, but computer scientists generally start counting at 0, so we prefer this version:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 10; i++)
        printf("*");
}
```

- In `buggy-1.c`[2] we want to print one asterisk per line:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i <= 10; i++)
        printf("*");
        printf("\n");
}
```

  - But this program prints all the stars on the same line:

```
jharvard@appliance (~/Dropbox/src2m): ./buggy-1
***********
```

  - So even though we indented line 7 and 8, we need curly braces to signify that those are part of the same block. If we only have one line in the body of the loop, however, then the braces would be optional. (But we prefer you always use curly braces, especially if you're new to programming!)

[2] http://cdn.cs50.net/2014/fall/lectures/2/m/src2m/buggy-1.c

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i <= 10; i++)
    {
        printf("*");
        printf("\n");
    }
}
```

- In the real world, bugs can have serious effects. In February, Apple's OS X and iOS operating systems had a bug in its SSL, Secure Sockets Layer, software. SSL is involved with visiting websites that begin with `https` and use encryption, and Apple's implementation of SSL originally looked like this:

```
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

- Note that line 9 is accidentally repeated, and the lack of curly braces means that the code actually looks like this:

```
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

  # So this means that line 9 is executed no matter what, and the last check for SSL will never be done.

- If you have an older version of Mac OS, gotofail.com[3] will test whether you're vulnerable.

Administrative Details

- **Sections**[4] start Sunday 9/21, and resectioning will be accomodated in the days to come.
- **Office hours**[5] may vary between weeks, so be sure to check the schedule.
- **Assessment** of problem sets will be along the axes of scope (how much of the problem set did you attempt?), correctness (does it work, and without bugs?), design (is your code written well?), and style (is easy for another human to read, with appropriate indentation and variable names?).
  # We score these each on a 5 point scale, with 3 being good.
  # We weigh things according to the following formula:

```
scope x (correctness x 3 + design x 2 + style x 1)
```

- **Academic honesty**: CS50 has the most Ad Board cases because the work is electronic, and as computer scientists we can look for and find cases more easily.

[3] http://gotofail.com
[4] http://cs50.harvard.edu/sections
[5] http://cs50.harvard.edu/hours

  - # The syllabus[6] gives the bottom line as "be reasonable" and has further guidelines, but remember that "the essence of all work that you submit to this course must be your own."
  - # "… you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints" allows students to help each other at office hours and beyond, but remember that other aspects of the policy also need to be followed.
    - # For context and transparency, about 20 students were involved last fall, and a range of 0 - 5% of students in the course in the past years.
    - # We compare current submissions to past submissions, code repos, discussion forums, etc.
- We have a **regret clause** in the syllabus[7] also:
  - # "If you commit some act that is not reasonable but bring it to the attention of the course's heads within 72 hours, the course may impose local sanctions that may include an unsatisfactory or failing grade for work submitted, but the course will not refer the matter to the Administrative Board."
  - # We hope to turn moments of bad decisions into teaching opportunities, rather than drastic consequences.
- Let's break the tension with Tiny Hamsters Eating Tiny Burritos - Episode 1[8].

Functions

- We can write our own **functions**. Let's open `function-0.c`[9]:

[6] http://cs50.harvard.edu/syllabus#academic_honesty
[7] http://cs50.harvard.edu/syllabus#academic_honesty
[8] http://youtu.be/JOCtdw9FG-s
[9] http://cdn.cs50.net/2014/fall/lectures/2/m/src2m/function-0.c

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

// prototype
void PrintName(string name);

int main(void)
{
    printf("Your name: ");
    string s = GetString();
    PrintName(s);
}

/**
 * Says hello to someone by name.
 */
void PrintName(string name)
{
    printf("hello, %s\n", name);
}
```

 - # We first see the main function in lines 7-12. A good way to start reading someone else's program is to start with the main function and follow its logic.
 - # This program asks for your name, and then we see `PrintName` in line 11. `PrintName` isn't in `stdio` or another library, but in the same file in lines 17-20. Lines 14-16 is the description of the function as a comment, and lines 17-20, `PrintName`, is a simple function that takes one argument in the parentheses, `string name`. That just means this function as something that takes in a `string` called `name`.

- The use of a function like this is called **abstraction**, in this case calling it `PrintName` that describes what it does. `void` in line 17 just means it doesn't return any value. Instead, `PrintName` has a **side effect** of printing to the screen.
- Here's another example, `return.c`[10]:

[10] http://cdn.cs50.net/2014/fall/lectures/2/m/src2m/return.c

```
#include <stdio.h>

// function prototype
int cube(int a);

int main(void)
{
    int x = 2;
    printf("x is now %i\n", x);
    printf("Cubing...\n");
    x = cube(x);
    printf("Cubed!\n");
    printf("x is now %i\n", x);
}

/**
 * Cubes argument.
 */
int cube(int n)
{
    return n * n * n;
}
```

- In line 8 we make a variable `x`, give it the value `2`, say what it is in line 9, and call the function `cube` in line 11, saving the result in the same variable `x`. We overwrite the value of `x` with whatever `cube` returns.
- In line 19, we give the `cube` function `int n` as an argument, meaning it takes in an integer. The beginning is also now `int` rather than `void`, which means it returns an integer instead of nothing.

- Now let's look at `function-1.c`[11]:

[11] http://cdn.cs50.net/2014/fall/lectures/2/m/src2m/function-1.c

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

// prototype
int GetPositiveInt(void);

int main(void)
{
    int n = GetPositiveInt();
    printf("Thanks for the %i!\n", n);
}

/**
 * Gets a positive integer from a user.
 */
int GetPositiveInt(void)
{
    int n;
    do
    {
        printf("Please give me a positive int: ");
        n = GetInt();
    }
    while (n < 1);
    return n;
}
```

- In this example, we write the `GetPositiveInt` function, using a do-while loop to keep getting an integer from the user until `n` is a positive number.
- Notice that `n` is declared on line 18 rather than 22, so it will be accessible within the entire function. A simple rule of thumb is that a variable can only be used within the most recent curly braces it's declared in, so we won't be able to use `n` outside of the do-while loop if it's declared inside. `n` has a **scope**, or usability, limited to the area it is declared in, whether it's an entire function or a loop.
- Also, note that the top we have a **prototype** on line 5, which is declaring that the `GetPositiveInt` exists somewhere. Otherwise, `main` would not be able to refer to the function in line 9. This is because the compiler goes top to bottom, and the `GetPositiveInt` function didn't exist by the time it is called, so it would cause

an error, unless we declare it with a prototype as in line 5. `#include` is done at the top of the file for the exact same reason, so those functions are declared before they are called.

- A clever solution would be moving the function above `main`:

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

/**
 * Gets a positive integer from a user.
 */
int GetPositiveInt(void)
{
    int n;
    do
    {
        printf("Please give me a positive int: ");
        n = GetInt();
    }
    while (n < 1);
    return n;
}

int main(void)
{
    int n = GetPositiveInt();
    printf("Thanks for the %i!\n", n);
}
```

  - But stylistically, longer programs will benefit from having `main` at the top for convenience and readability.

- We can declare a variable **globally** by putting it at the top of the file, outside all curly braces, but we frown upon that for now.

Data Representation

- Another way to look at implementation is how we represent information.
  - Recall that we have various types like `char`, which is 1 **byte**, or 8 bits in size. (8 bits would represent 256 unique patterns, counting from 0 and ending at 255.) Since

other languages have more characters, other standards have been adopted, but for English in C we default to the `char` type.

- An `int` is 4 bytes, with about 4 billion possible values, including negative numbers.
- A `float` is also 4 bytes, but has a decimal point somewhere to represent floating-point values, as well as a limited amount of precision.
- We can slightly avoid that problem with a `double` with 8 bytes of memory for a larger, but still finite, number of floating-point values.
- A `long long` is also 8 bytes, but stores integer values.
- Fun fact, a `long` in C is 4 bytes, the same as an `int`, for historical reasons.

• **Integer overflow** is a problem with limited sizes of variables. If we had one byte to store an integer that looked like this:

```
128     64     32     16      8      4      2      1

  1      1      1      1      1      1      1      1
```

• This number represents 255, but if we added 1 to this number, the value would carry over to become 0:

```
128     64     32     16      8      4      2      1

  0      0      0      0      0      0      0      0
```

• There is no bit to hold the final, left-most `1`, so with one byte, counting up from 255 will result in 0.

• We also have **floating-point imprecision**. Let's look at `floats-0.c`[12]:

```
#include <stdio.h>

int main(void)
{
    float f = 1 / 10;
    printf("%.1f\n", f);
}
```

[12] http://cdn.cs50.net/2014/fall/lectures/2/m/src2m/floats-0.c

# The program should print out `.1`, but instead:

```
jharvard@appliance (~/Dropbox/src2m): ./floats-0
0.0
jharvard@appliance (~/Dropbox/src2m):
```

# This is because the value of `1` and `10` in line 5 is assumed to be of an integer, and the decimal value we expect is **truncated**, or thrown away. We still see `0.0`, but only because `%.1f`, for one decimal point, is specified in line 6 to `printf`.

- If we fix the code to:

```
#include <stdio.h>

int main(void)
{
    float f = 1.0 / 10.0;
    printf("%.1f\n", f);
}
```

- We get:

```
jharvard@appliance (~/Dropbox/src2m): ./floats-0
0.1
jharvard@appliance (~/Dropbox/src2m):
```

- Let's see what happens when we print 28 decimal places:

```
#include <stdio.h>

int main(void)
{
    float f = 1.0 / 10.0;
    printf("%.28f\n", f);
}
```

- We expect something like `0.100000000000…` but we get:

```
jharvard@appliance (~/Dropbox/src2m): ./floats-0
0.10000000014901161193847656250
jharvard@appliance (~/Dropbox/src2m):
```

- Since the computer can’t represent an infinite number of real numbers, it gives us the closest number it can represent, which in this case is the number above.
- Though this seems insignificant, in software where we add and multiply and use numbers over and over again, these tiny mistakes add up.
- A clip from Modern Marvels[13] demonstrates a tragic mistake with floating-point imprecision, causing a missile to crash unintentionally.
- Colton Ogden plays another song[14] for Week 2.

[13] http://youtu.be/rOZrNQIFpAA?t=42m6s
[14] http://www.youtube.com/watch?v=mRVn_80SI5k