Week 2

This is CS50. Harvard University. Fall 2015.

Anna Whitney

Table of Contents

1. Bugs

- A **bug** is a mistake in a program that causes it to behave other than expected. Let's look at `buggy-0.c`[1]:

```
#include <stdio.h>

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

  # This sort of bug is called an **off-by-one error**—starting from `i = 0` and continuing while `i <= 10` results in 11 things counted total, rather than 10 as intended.

- So we can change the code like this:

[1] http://cdn.cs50.net/2015/fall/lectures/2/m/src2m/buggy-0.c

```
#include <stdio.h>

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

- This is more straightforward for humans, but various constructs in C and other programming languages are **0-indexed**, meaning they start from 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 ten asterisks, one 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@ide50:~/workspace/src2m $ ./buggy-1
***********
```

  # So even though we indented line 7 and 8, the compiler doesn’t understand indentation—whitespace in our program is entirely for human ease of reading. The compiler requires curly braces to know what’s the body of the loop and what isn’t.

[2] http://cdn.cs50.net/2015/fall/lectures/2/m/src2m/buggy-1.c

# If we only had one line in the body of the loop, however, then the braces would be optional. In the absence of curly braces, the compiler knows the loop must have something in the body, so it assumes the next line of code is the body of the loop. (But we prefer you always use curly braces, especially if you're new to programming, because it's so easy to make this mistake and add an extra line of code without remembering to add in the curly braces!)

```
#include <stdio.h>

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

- Let's look at the following, `loop.c`, in which the intent is to print the numbers 0 through 49:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 50; i++)
    {
        printf("i\n");
    }
}
```

# But `i` is just a character, so this just prints 50 of the letter `i`!

# We need a placeholder, rewriting the printf statement as `printf("%i\n", i)`, so we're getting the value of the integer variable `i`, rather than simply the character `i`.

# What if we accidentally type `for (int i = 0; i > 50; i++)` instead? Nothing happens—the body of the loop is never run—because the very first time the condition is checked, it's already false!

- How about this version, with a while loop?

```
#include <stdio.h>

int main(void)
{
    int i = 0;
    while (i >= 0)
    {
        printf("i\n");
    }
}
```

  # This results in an infinite loop because the while condition is initially met, and the value of i never changes, so the body of the loop is run again and again.

  # You can kill a program that's infinitely looping by pressing `ctrl-c`.

  # What if we add the line `i++;` in the body of the loop? Now `i` is changing, but it's still greater than 0, so we should still get an infinite loop, right? Not quite—once we get to the maximum possible integer, it will loop around to negative numbers, which will then be less than 0 and terminate the loop (but this takes a long time with 32-bit integers, and MUCH longer with 64-bit integers!)

2. Announcements

- If you're interested in participating in **YHACK** at Yale on November 6th-8th, see yhack.org[3] for details.
- **Sections** start next week; you should receive an email with your section assignment (please be patient—this is a very large class—but let us know if you haven't gotten a section assignment by next week).
- **Study50** provides resources from sections, if you'd like to review material from section, work ahead, or can't make it.
- **Office hours** may vary between weeks, so be sure to check the schedule.
- **CS50 Discuss** is another resource available to you when you have questions outside of office hours.

[3] http://yhack.org

- **CS50 Lunches**, weekly lunches with CS50 staff and industry friends (at both Harvard and Yale), will begin soon; you can sign up on the course website, first come first served.

- **Assessment** of problem sets will be along the axes of scope (how much of the problem set did you attempt?), correctness (does it work, per the specification, and without bugs? This can be tested using **check50**), design (is your code written well?), and style (is easy for another human to read, with appropriate indentation and variable names? See the CS50 Style Guide[4] for details).

  - We score these each on a 5 point scale, with 3 being a good score (and very few 5s given!).

  - We weigh things according to the following formula, which generally reflects how much time each aspect takes to get right:

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

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

  - The syllabus 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 help each other at office hours (and the overworked staff at office hours appreciate it!) but recall that you may show your broken code to others when looking for help, but may not view their working code.

    - For context and transparency, about 30 students were involved with the Ad Board 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 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

[4] http://manual.cs50.net/style

    include an unsatisfactory or failing grade for work submitted, but the course will not refer the matter for further disciplinary action except in cases of repeated acts."

  - We hope to turn moments of bad decisions into teaching opportunities, rather than drastic consequences.

  - Last year, when we introduced the regret clause, 19 students came forward.

    - 7 received no consequences, as the course staff decided they had not actually crossed a line with the actions they were concerned about.

    - 11 received a 0 score for the pset in question.

    - 1 student was asked to re-do a pset.

    - Conversations with these students often revealed academic or personal issues that we could then help students find the resources to solve, so this was one of the most successful policies the course has ever implemented.

- Let's break the tension with puppies![5]

3. Functions

- Recall that last time we moved beyond `main`, our default starting function, to write some of our own functions as well.

- Small programs can be written entirely in `main`, but as the problems we're solving get larger and more interesting, it's often worthwhile to factor out pieces of logic into separate functions.

- Let's open `function-0.c`[6]:

[5] http://cs50.harvard.edu/puppies

[6] http://cdn.cs50.net/2015/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);
}
```

# There are two functions here: `main` and `PrintName`. We pulled out the functionality that `PrintName` implements into a function as an example of **functional decomposition**—breaking a program down into its constituent bits—and **abstraction**—building higher-level functionality on top of lower-level code, and hiding the implementation behind a readable function name.

- Now let’s look at `function-1.c`[7]:

[7] http://cdn.cs50.net/2015/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;
}
```

- Unlike `PrintName` in the previous example, `GetPositiveInt` is not a one-liner - there's a little more legwork here, with a do-while loop, which we're hiding behind the abstraction `GetPositiveInt`.

- 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 wouldn't be able to use `n` outside of the do-while loop (or even in the condition of the do-while loop!) if it were declared inside. Every variable has a **scope**, or area of the program in which it's usable, limited to the area it is declared in, whether it's an entire function or a loop.

- Trying to use a variable outside of its scope results in compiler errors!

- We can declare a variable **globally** (so that it's in scope everywhere) by putting it at the top of the file, outside all curly braces, but we frown upon that for now.
  - Think about the fact that declaring `n` globally would mean that it's defined not just throughout `PrintName` but also throughout `main`, where we have a different variable called `n`!
  - Best design is to choose the narrowest possible scope for your variables.
- The `int` in `int GetPositiveInt(void)` indicates that the **return type** of the `GetPositiveInt` function is an integer. The `void` indicates that this function does not take any input of any type.
- Also, note that at the top we have a **prototype** on line 5, which declares that `GetPositiveInt` exists somewhere (and specifies its return and input types). Without this, `main` would not be able to refer to the function in line 9. This is because the compiler goes top to bottom, so without the prototype, the compiler complains that we're trying to use a function that hasn't been declared yet.
  - One possible 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.
  - There may also be situations where two functions both call each other, so it’s not possible to simply put one on top of the other! Prototypes solve this problem by letting you declare the function’s signature before actually defining what the function does.
  - The header files `stdio.h` and `cs50.h` contain prototypes for the functions they provide.

- In `cough-0.c`[8], we have a C implementation of what we went through in Scratch before:

[8] http://cdn.cs50.net/2015/fall/lectures/2/m/src2m/cough-0.c

```
#include <stdio.h>

int main(void)
{
    // cough three times
    printf("cough\n");
    printf("cough\n");
    printf("cough\n");
}
```

  # This involves copy-pasting the `printf` line three times - bad design!

- In `cough-1.c`[9], we improve upon this using a loop:

```
#include <stdio.h>

int main(void)
{
    // cough three times
    for (int i = 0; i < 3; i++)
    {
        printf("cough\n");
    }
}
```

  # This could also be implemented using a `while` loop, but either way, it avoids having to copy-paste a line of code to make it happen multiple times.

- We can go further and abstract out the `printf` statement into a function called `cough`, in `cough-2.c`[10]:

[9] http://cdn.cs50.net/2015/fall/lectures/2/m/src2m/cough-1.c

[10] http://cdn.cs50.net/2015/fall/lectures/2/m/src2m/cough-2.c

```
#include <stdio.h>

// prototype
void cough(void);

int main(void)
{
    // cough three times
    for (int i = 0; i < 3; i++)
    {
        cough();
    }
}

/**
 * Coughs once.
 */
void cough(void)
{
    printf("cough\n");
}
```

# Printing something to the screen is a **side effect**, so the `cough` function has a `void` return type, and it doesn’t take any input.

- We can make our cough function take an argument and allow it to loop any given number of times:

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

// prototypes
void cough(int n);

int main(void)
{
    // cough three times
    cough(3);
}

/**
 * Coughs n times.
 */
void cough(int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("cough\n");
    }
}
```

- We can also generalize further, as in `cough-4.c`[11]:

[11] http://cdn.cs50.net/2015/fall/lectures/2/m/src2m/cough-4.c

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

// prototypes
void cough(int n);
void say(string word, int n);
void sneeze(int n);

int main(void)
{
    // cough three times
    cough(3);

    // sneeze three times
    sneeze(3);
}

/**
 * Coughs n times.
 */
void cough(int n)
{
    say("cough", n);
}

/**
 * Says word n times.
 */
void say(string word, int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%s\n", word);
    }
}

/**
 * Sneezes n times.
 */
void sneeze(int n)
{
    say("achoo", n);
}
```

  # This abstracts out the entire idea of saying a word a given number of times.

  # The `say` function takes multiple arguments, separated by a comma.

4. Representing 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 |
-------------------------
```

  # Each of these boxes represents a one-byte block of memory, and we can access them individually using some syntax you'll see in a moment.

- Let's look at `string-0.c`[12]:

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

- In line 8 we use `strlen`, a function declared in the header `string.h`, to get the length of the string.

- 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`:

```
    -------------------------
```

[12] http://cdn.cs50.net/2015/fall/lectures/2/m/src2m/string-0.c

```
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.

- Now let's run this program:

```
jharvard@ide50:~/workspace/src2m $ ./string-0
Zamyla
Z
a
m
y
l
a
jharvard@ide50:~/workspace/src2m $
```

- What happens if instead of using `strlen`, we just assume that nobody will ever have a name longer than, say, 50 characters?

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

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

    for (int i = 0; i < 50; i++)
    {
        printf("%c\n", s[i]);
    }
}
```

# Now if we type in Zamyla, with this loop we're asking the program to give us Zamyla's name plus a whole bunch of extra bytes of memory. Let's see what we get:

```
jharvard@ide50:~/workspace/src2m $ ./string-0
Zamyla
Z
a
```

```
m
y
l
a
[... many lines of whitespace interspersed with funky characters ...]
jharvard@ide50:~/workspace/src2m $
```

# What if we go really reckless and use `500000` instead of `50`?

```
jharvard@ide50:~/workspace/src2m $ ./string-0
Zamyla
Z
a
m
y
l
a
[... many, MANY lines of whitespace interspersed with funky
 characters ...]
Segmentation fault
jharvard@ide50:~/workspace/src2m $
```

# A **segmentation fault** means we touched a segment of memory that doesn't belong to us. C gives us access to our computer's entire memory, but a lot of it is being used for other things!

- There's a major inefficiency in our `for` loop above. Let's look at `string-2.c`[13]:

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

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

- Remember in line 13 we initialize an `int i = 0` and increment it by `i++` every time. The condition in `string-1.c` checks that `i < strlen(s)` is true in order to continue the loop, meaning we keep calculating the length of `s` over and over again unnecessarily (since the length of Zamyla's name isn't changing!)

- 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.

5. 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`[14]:

[14] http://cdn.cs50.net/2015/fall/lectures/2/m/src2m/ascii-0.c

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

# Line 6 runs through values for 26 letters, 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. A type in parentheses prior to the name of a variable means that variable **cast** to the type in parentheses.

# Changing from `A` to `B` can be done by thinking of `A` as `65` and `B` as `66` and adding 1.

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

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

6. References

- We’ve introduced a few **header files** for libraries at this point, including `stdio.h`, `cs50.h`, `string.h`, and `ctype.h`.
- A useful source of information about the functions in these headers is **Reference50**[15], which provides both less comfortable explanations and official Linux man page explanations.
- Next time, we’ll discuss cryptography!

[15] http://reference.cs50.net