Problem 1-4: Skittles

This is CS50. Harvard University. Fall 2014.

Table of Contents

Questions? Feel free to head to CS50 on Reddit[1], CS50 on StackExchange[2], or the CS50 Facebook group[3].

Objectives

- Learn about pseudorandom numbers.
- Learn how to use online references to gain insight into using unfamiliar functions.
- Continue experimenting with CS50 IDE.
- Build your first game in C.

Recommended Reading

- Pages 1 – 7, 9, and 10 of http://www.howstuffworks.com/c.htm.

[1] https://www.reddit.com/r/cs50
[2] http://cs50.stackexchange.com
[3] https://www.facebook.com/groups/cs50

- Chapters 1 – 5, 9, and 11 – 17 of *Absolute Beginner's Guide to C*.
- Chapters 1 – 6 of *Programming in C*.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problems is not permitted (unless explicitly stated otherwise) except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code or writing to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on quizzes and tests is not permitted at all. Collaboration on the final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it until you solicit and receive approval in writing from your instructor. If a violation of this policy is suspected and confirmed, your instructor reserves the right to impose local sanctions on top of any disciplinary outcome that may include an unsatisfactory or failing grade for work submitted or for the course itself.

Reasonable

- Communicating with classmates about problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code, such as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.

- Reviewing past years' quizzes, tests, and solutions thereto.
- Sending or showing code that you’ve written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own solutions to problems online so that others might help you identify and fix a bug or other issue.
- Turning to the web or elsewhere for instruction beyond the course’s own, for references, and for solutions to technical difficulties, but not for outright solutions to problems or your own final project.
- Whiteboarding solutions to problems with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff’s solutions to problems.
- Failing to cite (as with comments) the origins of code, writing, or techniques that you discover outside of the course’s own lessons and integrate into your own work, even while respecting this policy’s other constraints.
- Giving or showing to a classmate a solution to a problem when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual’s work during a quiz or test.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problems to individuals who might take this course in the future.
- Searching for, soliciting, or viewing a quiz’s questions or answers prior to taking the quiz.

- Searching for or soliciting outright solutions to problems online or elsewhere.
- Splitting a problem's workload with another individual and combining your work (unless explicitly authorized by the problem itself).
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.
- Viewing another's solution to a problem and basing your own solution on it.

Assessment

Your work on this problem set will be evaluated along four axes primarily.

Scope
: To what extent does your code implement the features required by our specification?

Correctness
: To what extent is your code consistent with our specifications and free of bugs?

Design
: To what extent is your code written well (i.e., clearly, efficiently, elegantly, and/or logically)?

Style
: To what extent is your code readable (i.e., commented and indented with variables aptly named)?

To obtain a passing grade in this course, all students must ordinarily submit all assigned problems unless granted an exception in writing by the instructor.

Free Candy

So, one of the best things about our office is the free[4] candy machine.

[4] We might have been the ones to hack the machine to be free.

Well, that and the stuffed cat in the ceiling[5].

Anyhow, there's a whole lot of Skittles, Mike and Ike's, and M&M's in that machine. Want to guess how many Skittles? Glad you said yes! Your task in this problem is to implement, in a file called `skittles.c` inside of your `~/workspace/unit1` directory (remember how?), a program that first picks a (pseudorandom) number between 0 and 1023, inclusive, and then asks you (the human) to guess what it is.[6] The program should keep asking you to guess until you guess right, at which point it should thank you for playing.

Where to begin? Allow us to hand you some puzzle pieces.

To generate a random number, you can use a function called `rand`. Take a peek at its manual page, commonly called a `man` page, by checking it out on Reference50[7].

Under **Synopsis**, we see that the function is apparently declared in `stdlib.h`. So you'll want to put

```
#include <stdlib.h>
```

atop `skittles.c` along with

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

[5] https://www.google.com/search?q=ceiling+cat
[6] To be clear, that range includes 1024 integers, from and including 0, to and including 1023.
[7] https://reference.cs50.net/stdlib.h/rand

as usual. The order of such includes tends not to matter, but alphabetical is probably good style.

Also notice that rand "returns a pseudorandom integer between zero and `RAND_MAX`." It turns out that `RAND_MAX` is a **constant** (a symbol that represents some value) that's defined in `stdlib.h`. Its value can vary by server, and so it's not hard-coded into the manual. Let's assume that `RAND_MAX` is greater than 1023. How, though, do we map a number that's between 0 and `RAND_MAX` to a number that's between 0 and 1023?

Turns out that C includes an operator—alongside the usual suspects of addition, subtraction, multiplication, and division—called the **modulo** operator. Modulo (`%`) gives you the remainder after dividing its operands. But it can be useful for more than arithmetic remainders alone! Consider this line of code:

```
int skittles = rand() % 1024;
```

The effect of that line is to divide the **return value** of `rand` by 1024 and store the remainder in `skittles`. What might the remainder be, though, when dividing some integer by 1024? Well, there might be no remainder, in which case the answer is 0. Or there might be a big remainder, in which case the answer is 1023. And, of course, a whole bunch of other remainders are possible in between those bounds. Well there you have it, a way of generating a pseudorandom number between 0 and 1023, inclusive!

That's So Pseudorandom!

There's a catch, though. It turns out that, by default, `rand` always returns the same number (odds are it's `1804289383`) the first time it's called in a program, in which case your program's always going to be filled with the same number of Skittles. Why is that? On the aforementioned Reference50 page, take a second to click that "More Comfy" radio button in the top-right corner of the page.

Whoa! That's a lot more detail. That's what the real `man` page looks like—a bit overwhelmin. Still, some useful information seems to have been abstracted away in "Less Comfy" mode. In particular, there's a note under the Description that

> If no seed value is provided, the `rand()` function is automatically seeded with a value of 1.

A **seed** is simply a value that influences the sequence of values returned by a **pseudorandom number generator** (PRNG) like `rand`. To be clear, it's not the first number returned by a PRNG but, rather, an influence thereon.

This is why we say "pseudorandom" all the time instead of "random". Computers can't really generate numbers that are truly random: they have to start somewhere. How can you override this default seed of 1? Before you call `rand`, call `srand` with your choice of seed (e.g., 2):

```
srand(2);
```

Better yet, call `srand` with a seed that actually changes over time (literally), without your having to recompile your code each time you want it to change:

```
srand(time(NULL));
```

Never mind what `NULL` is for now, but know that `time(NULL)` returns the current time in seconds; that's not a bad seed. No need to store the return value of `time` in some variable first; we can pass it directly to `srand` between those parentheses. It's worth noting, though, that time is declared in `time.h`, so you'll need to include that header file too.

Guesstimating

Alright, what other puzzle pieces do we need? Well, your program will need to tell the user what to do, for which `printf` should be helpful. And you'll want to allow the user an infinite number of guesses, for which some looping construct is probably necessary. And you'll also want to get integers from the user, for which `GetInt`, declared in `cs50.h`, is definitely handy.

Okay, where to begin? Allow us to suggest that you begin by filling `skittles.c` with this code:

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

int main(void)
{
    // seed PRNG
    srand(time(NULL));

    // pick pseudorandom number in [0, 1023]
    int skittles = rand() % 1024;

    // TODO
}
```

We'll leave the `TODO` to you! Remember, don't try to implement the whole program at once. Perhaps start by printing (with `printf`) the value of `skittles`, just to be sure that you didn't make any typos. Then save your code and proceed to compile it with[8]

```
make skittles
```

To run your program (assuming it compiles with no errors), execute

```
./skittles
```

to see how many Skittles there are. Wait one second and then run your program again: odds are the number of Skittles will differ. Then continue editing `skittles.c` and take another bite out of this problem. Perhaps next implement your program's instructions that explain to the user how to play this guessing game.

What should your program's output be, once fully implemented? We leave your program's personality entirely up to you, but below's one possible design. Assume that the underlined text is what some user has typed.

[8] If only it were that easy to make Skittles.

```
username@ide50:~/workspace/unit1 $ ./skittles
O hai! I'm thinking of a number between 0 and 1023. What is it?
0
Nope! There are way more Skittles than that. Guess again.
1
Nope! There are way more Skittles than that. Guess again.
-1
Nope! Don't be difficult. Guess again.
1023
Nope! There are fewer Skittles than that. Guess again.
42
That's right! Nom nom nom.
```

Your program should end once the user has guessed right. The above design happens to respond to the user's input in a few different ways, but we leave it to you to decide how much to vary your program's output.

Incidentally, know that you can generally force a program to quit prematurely by hitting ctrl-c. And you may be able to extrapolate from the phone book example: Finding a value between 0 and 1023 doesn't actually require that many guesses. Odds are you can test your program fairly efficiently. You can certainly use temporarily a modulus less than 1024 to save even more time; just be sure that your final version does pick a number in [0, 1023].

If you'd like to play with the our own implementation of `skittles` you may execute the below.

```
~cs50/unit1/skittles
```

This was Problem 1-4.