Week 4

This is CS50. Harvard University. Fall 2015.

Anna Whitney

Table of Contents

1. Volkswagen and Trust in Software 1
2. Recursion 2
3. Sigma 4
4. Swap 8
5. Debugging with CS50 IDE 11
6. Pointers 15

1. Volkswagen and Trust in Software

- Volkswagen is in trouble for faking its emission tests using software.
- "A sophisticated software algorithm on certain Volkswagen vehicles detects when the car is undergoing official emissions testing and turns full emissions controls on only during the test."
- "The software produced by Volkswagen is a 'defeat device' as defined by the Clean Air Act."
- The software for faking emissions was discovered in an independent investigation at WVU.
- Based on various inputs (position of steering wheel, vehicle speed, duration of engine's operation, barometric pressure), the car's Electronic Control Module (ECM) would track whether the car was undergoing the standard emissions testing procedure.
- Although the actual VW source code has not been released, the algorithm essentially boils down to this:

```
if being tested
    turn full emissions controls on
```

```
else
    don't
```

- Perhaps more concretely:

```
if wheels are turning but steering wheel isn't
    turn full emissions controls on
else
    don't
```

- This video[1] describes the actual consequences this software has on the function of the cars.
- So how do we know that all the software we use is only doing what we think it is?
    # We can look at the source code of, e.g., the CS50 Library to make sure that the CS50 staff have not inserted code we don't want to run.
    # But even `clang` could have a "Trojan horse" built in that could add code to your programs when they're being compiled.
    # Even if we look at the source code for `clang`, we can't guarantee it doesn't have any malicious code, because compilers are themselves compiled with older versions of themselves!
    # Ken Thompson gave a talk[2] about trust in software that essentially comes down to the fact that we can't trust software - the best we can do is to trust the people who wrote it.
- To lighten the mood, watch this adorable Volkswagen ad[3] from the 2011 Superbowl that almost makes them likeable again.

2. Recursion

- Recall our pseudocode from back at the very beginning to find Mike Smith in the phone book:

[1] https://www.youtube.com/watch?v=CQ4irwe3ZDk
[2] http://cdn.cs50.net/2015/fall/lectures/4/m/p761-thompson.pdf
[3] https://www.youtube.com/watch?v=R55e-uHQna0

```
pick up phone book
open to middle of phone book
look at names
if "Smith" is among names
    call Mike
else if "Smith" is earlier in book
    open to middle of left half of book
    go to line 3
else if "Smith" is later in book
    open to middle of right half of book
    go to line 3
else
    give up
```

- Note that lines 8 and 11 have this `go to` construct, which creates a loop (C does actually have a `go to` statement, but its use is strongly discouraged because it is very easy to get wrong, makes it harder to reason about your program's behavior, and can easily be abused).
- Instead, we could write this program as follows:

```
pick up phone book
open to middle of phone book
look at names
if "Smith" is among names
    call Mike
else if "Smith" is earlier in book
    search for Mike in left half of book
else if "Smith" is later in book
    search for Mike in right half of book
else
    give up
```

- Now instead of inducing a loop with `go to`, lines 7 and 9 **recursively** call the entire algorithm, telling us to start back at the beginning - but with a smaller problem!

  # Eventually we'll reach the **base case**, where we have just one page left and either "Smith" is there and we call Mike, or he's not and we give up.

- Recall our algorithm for merge sort, which is also recursive:

```
On input of n elements
```

```
if n < 2
    return
else
    sort left half of elements
    sort right half of elements
    merge sorted halves
```

- An algorithm is **recursive** if it **calls** itself.
- More formally, in C, a function `foo()` is recursive if somewhere in the code of `foo()`, there is a call to the function `foo()` itself.

  # If all `foo()` ever does is call itself, we have a problem! But as long as the problem gets smaller and we have a base case, this is just fine and will end.

3. Sigma

- Let's take a look at `sigma-0.c` [4]:

[4] http://cdn.cs50.net/2015/fall/lectures/4/m/src4m/sigma-0.c

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

// prototype
int sigma(int);

int main(void)
{
    // ask user for a positive int
    int n;
    do
    {
        printf("Positive integer please: ");
        n = GetInt();
    }
    while (n < 1);

    // compute sum of 1 through n
    int answer = sigma(n);

    // report answer
    printf("%i\n", answer);
}

/**
 * Returns sum of 1 through m; returns 0 if m is not positive.
 */
int sigma(int m)
{
    // avoid risk of infinite loop
    if (m < 1)
    {
        return 0;
    }

    // return sum of 1 through m
    int sum = 0;
    for (int i = 1; i <= m; i++)
    {
        sum += i;
    }
    return sum;
}
```

- The program adds the numbers `1` through `n`. Notice that there is a prototype on line 5, and all that does is say that there will be a function later on in the program, named `sigma`, that takes an `int` in the parentheses, and returns an `int`.

  # We need this so the compiler knows this function will be defined below, because it compiles the code in order.

- Now let's look at `main`, where we ask the user for an integer until we get a positive one, using a `do-while` loop as we have before. Then on line 19 we create a variable called `answer`, and store the return value of the `sigma` function to it, after we pass it the `n` from the user.

- Before moving on, let's run it:

```
jharvard@ide50:~/workspace/src4m $ ./sigma-0
Positive integer please: 2
3
```

  # And 2 + 1 is indeed 3.

- What if we give it `3`? 3 + 2 + 1 = 6.

```
jharvard@ide50:~/workspace/src4m $ ./sigma-0
Positive integer please: 3
6
```

- And bigger numbers should give us bigger sums:

```
jharvard@ide50:~/workspace/src4m $ ./sigma-0
Positive integer please: 50
1275
```

- So how does the `sigma` function actually work?

```
int sigma(int m)
{
    // return sum of 1 through m
    int sum = 0;
    for (int i = 1; i <= m; i++)
    {
        sum += i;
    }
    return sum;
}
```

- Remember that we declare `sum` outside the loop, so that we can access it outside of the `for` loop, and also so that we don't reset it to `0` every pass of the loop.
- Variables are generally scoped to the curly braces that encompass them, so we need to put them outside the curly braces of the `for` loop in order to `return` it after.
- Finally, in `main` we simply call the `sigma` function and print the value it returns. In this case, `sigma` is written with an **iterative** approach where it does the same thing, over and over again.
- But we can implement it differently, as in `sigma-1.c`[5]:

```
int sigma(int m)
{
    if (m <= 0)
        return 0;
    else
        return (m + sigma(m - 1));
}
```

- Here we start by returning `0` if `m # 0`, which is the base case. This is equivalent to knowing what to do when we got down to one page of the phone book in our previous example.
- The beauty is in the `else` condition: the sum of the numbers from `1` to `m` is the same as the sum of `m`, and the sum of the numbers from `1` to `m - 1`. So we can follow this logic, passing each smaller value back to `sigma`, from `sigma(n)` to `sigma(n - 1)` to `sigma(n - 2)` until we get to `sigma(0)`, which is added back up to all those other questions.

[5] http://cdn.cs50.net/2015/fall/lectures/4/m/src4m/sigma-1.c

    # At each step, we keep around the last value while we make the next function call, and hold on to it until we get back the result from that function call.

    # We "stack" these results in order, so when we get back each call, we can rewind in time and add them up in the correct order (the order doesn't matter for addition, but it might for other recursive algorithms).

- Volunteer Sam comes up to Google "recursion", which results in Google prompting us, "Did you mean: *recursion*".

    # For entertainment's sake: try Googling **anagram**, **askew**, or **do a barrel roll**.

    # Clearly Google has a few `if` conditions under the hood to check if the user typed any of these search terms!

4. Swap

- Before we move on, a demonstration from a volunteer from the audience, Lauren. We have some orange juice and milk, each in a glass, and to swap them, Lauren needed a third cup, using it to store the orange juice. Then she poured the milk into the cup originally containing the orange juice, and finally the orange juice into the cup that originally had the milk.

    # We can think of the third cup as a **temporary variable** to store the value of the orange juice cup or the milk cup. Here's the corresponding C code:

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

    # Lauren also tries this without a third cup using oil and water, taking advantage of the fact that they don't mix. We can actually swap two values without using a temporary variable, using the magic of bitwise operators:

```
void swap(int a, int b)
{
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}
```

`^` is XOR, or exclusive OR. Try working through this by hand with a couple of 8-bit values (it works with any bitstring, but it'll take a while by hand!), and verify that it does in fact switch the values.

This sort of micro-optimization is cute, but not particularly useful in most cases (since generally the 32-bit overhead of assigning one extra integer variable is negligible relative to overall memory usage of your software).

- Let's open an example, `noswap.c`[6]:

[6] http://cdn.cs50.net/2015/fall/lectures/4/m/src4m/noswap.c

```
#include <stdio.h>

void swap(int a, int b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %i\n", x);
    printf("y is %i\n", y);
    printf("Swapping...\n");
    swap(x, y);
    printf("Swapped!\n");
    printf("x is %i\n", x);
    printf("y is %i\n", y);
}

/**
 * Fails to swap arguments' values.
 */
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

- We call this `noswap` because it doesn't actually work. In `main`, we declare `x` and `y`, print out messages for us to see their values, call the `swap` function, and print their values again.
- But when we run it:

```
jharvard@ide50:~/workspace/src4m $ ./noswap
x is 1
y is 2
Swapping...
Swapped!
x is 1
y is 2
```

- What's going on? Let's look at the values of our variables inside the `swap` function as follows:

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
    printf("a is %i\n", a);
    printf("b is %i\n", b);
}
```

  # This is an example of debugging with `printf`, a simple technique for figuring out what's going on inside.

- Now when we run it:

```
jharvard@ide50:~/workspace/src4m $ ./noswap
x is 1
y is 2
Swapping...
a is 2
b is 1
Swapped!
x is 1
y is 2
```

- Variables `x` and `y` are **local** to `main`. We pass `x` and `y` to the function `swap`, where we're calling them `a` and `b` (just so it's clear that we can pass the function values other than `x` and `y` specifically). But somehow the versions of `x` and `y` inside `swap` are different from the versions inside `main`.

5. Debugging with CS50 IDE

- Let's debug this, not with `printf` statements like we've done so far, but using the built-in debugger in the CS50 IDE. This lets us get inside our program in real time.
- We start by clicking on the **Debugger** tab on the right edge of the IDE.

  # Under **Local Variables**, we'll be able to see the values of all our local variables at a particular point in the execution of our program.

    - We can look at a particular point by setting a **breakpoint**, which we can set by clicking to the left of the line number at the point in the code we want to inspect, and all our breakpoints will appear under **Breakpoints** in the debugger tab. When the program gets to that line, it will pause so you can look at what's going on.
    - **Call Stack** describes the functions that have been called, starting with `main`.
- Let's start by setting a breakpoint at `main` in `noswap.c` - a red dot should appear to the left of the line.
- Now we click **Debug** at the top of the page, and we'll see a debugger window at the bottom of the page (where the Terminal usually is).
    - You should see the first line of code inside main highlighted in yellow, indicating that execution has paused there (or really immediately before the highlighted line).
    - We can see local variables `x` and `y` have been created, but don't have values yet (so they initially have value `0`, in this particular case).
    - At this point we could click **Resume** (the play button in the debugger tab) to continue executing the program (until the next breakpoint, but so far we've only set one breakpoint).
    - We can also **Step Over** - i.e., run just the following line of code, without descending into any functions that are called there - or **Step Into** - i.e., run the following line of code and descend into any functions that are called there.
- For now, we'll click **Step Over**, which runs just the following line of code:

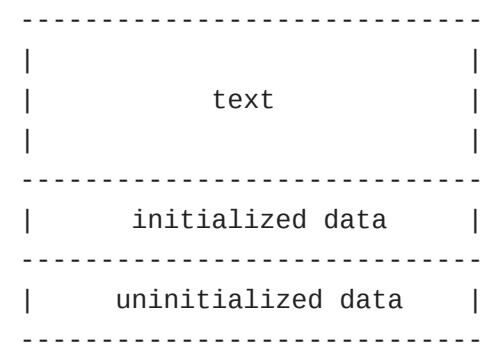
```
int x = 1;
```

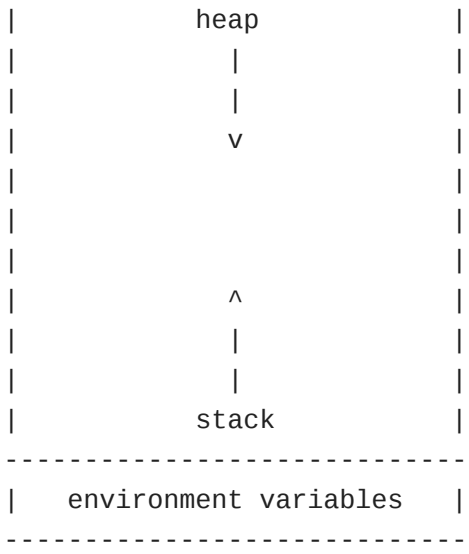
    - Now if you look in the **Local Variables** list, you'll see that `x` has the value `1`.
- Doing the same with the next line:

```
int y = 2;
```

    - Now `y` has the value `2` in the list of local variables as well.
- When we step over the `printf` lines, the output is printed in the debugger console.
    - If we were to step into the `printf` lines, the debugger would take us into the code for `printf` itself - not what we want, since our bug is certainly not caused by a bug in `printf`!

- Once we get to a line that calls a function we actually wrote - namely, `swap` - we click **Step Into** to see what's happening inside our function.
    # The first line of code inside `swap` is now highlighted, and our local variables are `a`, which has a value of `1`, passed from `x`; `b`, which has a value of `2`, passed from `y`; and `tmp`, which has a garbage value, since it has not been assigned a value yet.
        # This time we didn't get `0` as the uninitialized value, which is important to note - there's no guarantee that a variable you haven't assigned a value to will start with a value of `0`, or any other particular value.
        # The value of an uninitialized variable is whatever value was in the chunk of memory that will be used to store the variable, which is just junk left over from whatever that memory was last used for.
    # As we step over the three lines in `swap`, we can see `tmp` assigned the value of `1` from `a`, `a` assigned the value of `2` from `b`, and `b` assigned the value of `1` from `tmp`.
    # If we look at the **Call Stack**, we can see that now there are two entries: `swap` and `main`. By clicking on them, we can change contexts and see what the local variables are in each **scope**.
        # Inside `swap`, `a` and `b` have been swapped, but in `main`, `x` and `y` have not been affected.
- It turns out that when we pass arguments to a function like this, we're really passing copies of the variables - so what `swap` gets from `main` is two new locations in memory that contain the same values as `x` and `y`, and changing the values at those new locations doesn't change the original variables `x` and `y`.
- This is one way of thinking about how our computer's memory is laid out:

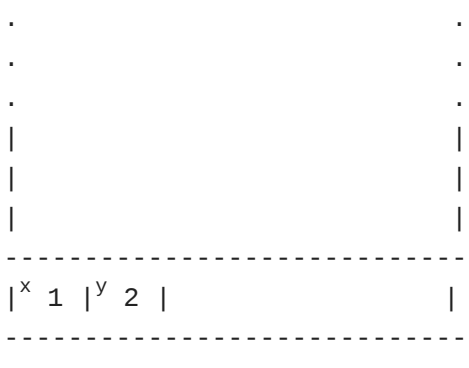




- The **stack** is just a chunk of memory used every time a function is called. The operating system takes some amount of bytes and lets you run your function with places for variables and other things you need. If you call another function, and another function, and another function, you get more pieces of memory.

  # Each function call gets its own layer of memory, and when a function calls another function (as `main` called `swap`), the next layer of memory is put down on top of the first one.

- So the stack in our program might start to look like this:



- `main` has this slice of memory, called a **stack frame**, and it contains two local variables - `x`, containing the value `1`, and `y`, containing the value `2`. Each variable gets its own 32-bit (because they're integers) chunk of memory within the layer that belongs to `main`.
- When `main` calls `swap`, `swap` gets another layer on top, where it puts its own local variables:

```
.                            .
.                            .
.                            .
|                            |
-----------------------------
|a 1 |b 2 |tmp   |           |
-----------------------------
|x 1 |y 2 |                  |
-----------------------------
```

- Once the code inside `swap` has run, it'll look like this:

```
.                            .
.                            .
.                            .
|                            |
-----------------------------
|a 2 |b 1 |tmp 1 |           |
-----------------------------
|x 1 |y 2 |                  |
-----------------------------
```

- So the swap has happened inside the section of memory allocated to `swap`, but the memory layer belonging to `main` is untouched.
- How can we give a function "secret access" to a section of memory belonging to another function?

6. Pointers

- Let's look at `compare-0.c`[7] to see what we've actually been working with this whole time:

[7] http://cdn.cs50.net/2015/fall/lectures/4/m/src4m/compare-0.c

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

int main(void)
{
    // get line of text
    printf("Say something: ");
    string s = GetString();

     // get another line of text
     printf("Say something: ");
     string t = GetString();

      // try (and fail) to compare strings
      if (s == t)
      {
          printf("You typed the same thing!\n");
      }
      else
      {
          printf("You typed different things!\n");
      }
}
```

- When we run it, what happens?

```
jharvard@ide50:~/workspace/src4m $ ./compare-0
Say something: mom
Say something: MOM
You typed different things!
jharvard@ide50:~/workspace/src4m $
```

Alright, those are different, as expected. What about if we type the same string twice?

```
jharvard@ide50:~/workspace/src4m $ ./compare-0
Say something: Mom
Say something: Mom
You typed different things!
jharvard@ide50:~/workspace/src4m $
```

- Even though those strings are identical, `compare-0` is still telling us that they're different.
- To figure out why, let's think about what `GetString()` is actually doing.
  # When `GetString()` is called, it gives us a chunk of memory, and fills in what the user types:

```
-----------------
| M | o | m |\0 |
-----------------
```

  # If we store the return value of `GetString()` in a variable `s`, what's actually in that variable?
  # The string `Mom` is stored somewhere in memory. Our computers have some number of bytes of memory (a very large number, likely in the billions), and we have some way of numbering them all. Let's imagine that our string `Mom` is stored in bytes `1` through `4`:

```
1   2    3   4
-----------------
| M | o | m |\0 |
-----------------
```

  # So what `GetString()` is actually returning is not the string `Mom` itself, per se, but the address in memory where we can find it - so what's being stored in `s` is actually the memory address `1`.

```
s       1    2   3    4
-----  -----------------
| 1 |  | M | o | m |\0 |
-----  -----------------
```

  # Now if we call `GetString` a second time, storing the result in the variable `t`, we get another string somewhere else in memory (let's say addresses `9` through `12`, assuming some memory in between is being used for something else), and we again store the address:

```
s       1    2   3    4
-----  -----------------
| 1 |  | M | o | m |\0 |
```

```
-----  -----------------
t       9     10    11    12
-----  -----------------
| 9 |  | M | o | m |\0 |
-----  -----------------
```

- So when we compare `s` to `t`, as we did with the condition `if (s == t)` in our original code, we see that they are not in fact equal, because `1` and `9` are not equal!
- We're only given the address of the beginning of the string, but we can figure out where it ends by looking for the `\0`.
- We're now taking off the training wheels and revealing that what we've been calling a `string` this whole time is actually a `char*`, where the `*` denotes a pointer.
- We'll discuss pointers in more detail next time, but for now this clip[8] gives a brief preview of what we're in for.

[8] https://youtu.be/SadMsthVUBM?t=3024