CS50x – Notes (Unofficial)

Courtesy of https://cs50.harvard.edu/lectures/

Acknowledgements

- Full credit to the students, teachers, staff, and volunteers at Harvard, CS50, and EdX who helped make this course possible
- This PDF is simply a quick reference to all Notes found on the website; the only changes are to format and the creation of this page. Nothing is added or removed from the website version of this information
- Share at will. This information is open to the public, for free
- **This PDF is in no way associated with Harvard, CS50, or EdX**. It is compiled for free, by a volunteer who is taking the course and wished to spread the resource to all. All of the information can be found at the aforementioned web address. This PDF simply saves you time

Table of Contents

Week 1

Andrew Sellergren

Table of Contents

Announcements and Demos

- Next time you're milling about the Science Center, take a gander at the Mark I, one of the very first electromechanical computers capable of long self-sustained computation. Check out this giant calculator. From this same Mark I computer comes the term "bug" that we take for granted. One of the engineers discovered an actual moth in the machine that was causing some incorrect calculations. The moth was then taped to a log book for posterity's sake.
- To help make the class feel a bit more intimate, we'll be gathering most Fridays at Fire and Ice in Harvard Square for a casual lunch. If you're interested, RSVP here.
- Avail yourself of the following resources at cs50.net/lectures to aid you in your quest through CS50:
    - videos
        - If you click the icon at the bottom right in the video player, you'll see a searchable full transcript of the lecture.
    - slides
    - examples
    - walkthroughs
        - To illuminate some of the more complex examples from lecture, we offer walkthroughs of those examples. Check out the first round here.
    - scribe notes

        - Welcome to the scribe notes! This is your canonical source of notes for each lecture so that you don't have to scribble down anything while listening to David. This is also your canonical source for jokes made at David's expense.

- Sectioning starts Wednesday. This Sunday [1] we'll offer a one-time supersection led by the course heads and available on video afterward.
- It's time to introduce our inimitable course heads: Lauren Carvalho, Rob Bowden, Joseph Ong, R.J. Aquino, and Lucas Freitas. Feel free to reach out to them at heads@cs50.net.
- Problem Set 0 has been released!
- Office Hours will begin soon! There really are no dumb questions. [2]
- CS50 Discuss is the course's forum where you can post any and all questions you have. We'll monitor it during lecture so that if you have a question about something David says, you can post it and we'll try to respond in realtime.
- If you come into this course with little or no prior background in computer science or you'd just like to have the safety net of being able to call it quits when you're 90% done with a problem set on a Thursday night, you should consider taking the course SAT/UNS. Trust us, you'll still learn plenty!

From Last Time

- In Scratch, we worked with purple puzzle pieces starting with verbs like "say" that represented *statements*.
- The blue hexagonal puzzle pieces that ended in question marks we deemed *boolean expressions*.
- We used these boolean expressions inside of *conditions*, which represented branches in our program's logic.
- The puzzle pieces that contain the words "repeat" and "forever" are examples of *loops*.
- We wrote our own *functions* using the purple curved puzzle pieces containing the word "define." A function is a chunk of code that we want to use over and over again without having to copy and paste.

From Scratch to C

- Recall that source code looks something like the following:

```
int main(void)
{
    printf("hello, world\n");
}
```

- The blue "say" puzzle piece from Scratch has now become `printf` and the orange "when green flag clicked" puzzle piece has become `main(void)`.
- However, source code is not something a computer actually understands. To translate source code into something the computer understands, we'll need a *compiler*. A compiler is a program that takes source code as input and produces 0's and 1's, a.k.a. object code, as output.
- We won't trouble ourselves with knowing the exact mapping between a series of 0's and 1's and the "print" command. Rather, we'll content ourselves with writing instructions at a higher level that can be translated to a lower level. This is consistent with one of the themes of the course: layering on top of the work of others.
- Statements are direct instructions, e.g. "say" in Scratch or `printf` in C.
- The "forever" loop from Scratch can be recreated with a `while (true)` block in C. The "repeat" loop from Scratch can be recreated with a `for` block in C.
- Note that in C just as in Scratch, there are multiple ways of achieving the same goals.
- In C, a loop that increments a variable and announces its value would look like so:

```
int counter = 0;
while (true)
{
    printf("%i\n", counter);
    counter++;
}
```

- Here we declare a variable named `counter` and then create an infinite loop that prints its value then increments it.
- Boolean expressions are much the same in C as in Scratch. The less-than (`<`) and greater-than (`>`) operators are the same. One difference is that the "and" operator is represented as `&&` in C.
- Conditions in C look much the same as they do in Scratch:

```
if (x < y)
{
    printf("x is less than y\n");
}
else if (x > y)
{
    printf("x is greater than y\n");
}
else
{
    printf("x is equal to y\n");
}
```

`hello, world!`

- The CS50 Appliance is software running inside of your normal computer's environment that simulates the environment of another operating system, namely Fedora Linux. At the bottom left of the Appliance window are three icons for gedit, Chrome, and Terminal. Since we can code in any text editor, let's start by opening gedit.
- In gedit, there are three main divisions of the window:
  - on the left, the source code pane
  - on the right, the actual text editor, where we write code
  - on the bottom, the terminal, where we run commands
- Note that all of your files by default save to the `jharvard` directory, which is unique to your Appliance and is not shared with other students. All of your files in the`Dropbox` subdirectory are automatically backed up in the cloud. In this directory, we'll save our file as `hello.c`.
- Now let's quickly rewrite that first program in C:

```
#include <stdio.h>

int main(void)
{
    printf("hello, world!\n");
}
```

- To actually run this program, we click on the terminal at the bottom of gedit. To begin, we're in the `jharvard` directory. On other operating systems, we can simply double click on another directory in order to open it. But in the terminal, we can only use the keyboard. So to get to the `Dropbox` folder, we instead type `cd Dropbox`. Now if we type `make hello`, we see a bit of cryptic syntax, but afterward, we can run `./hello` and see our program execute successfully.

- A single dot (`.`) refers to the current directory. Typing `./hello` instructs the computer to look for a program named `hello` in the current directory. Type `ls` to see the contents of the current directory. In green, you'll see `hello`, which is the executable program that we just compiled. Recall that we use a *compiler* to translate the source code above into the object code that the computer can actually understand.

Linux Commands

- As an aside, here's a short list of Linux commands that you'll find useful:
  - `ls`
    - stands for "list," shows the contents of the current directory
  - `mkdir`
    - stands for "make directory," creates a new folder
  - `cd`
    - stands for "change directory," the equivalent of double clicking on a folder
  - `rm`
    - stands for "remove," deletes a file
  - `rmdir`
    - stands for "remove directory," deletes a directory

Compiling

- When we type `make hello` in the terminal, the command that actually runs is as follows:
  ```
  clang -ggdb3 -O0 -std=c99 -Wall -Werror hello.c -lcs50 -lm -o hello
  ```
- `make` is not actually a compiler, but rather a program that shortcuts these options to the compiler, which in this case is `clang`. The shorter version of the command above is:
  ```
  clang -o hello hello.c
  ```
- `-o` is a switch or flag, an option that influences the behavior of the program. In this case, the value provided after `-o` is `hello`, which becomes the name of the executable that the compiler creates. We could've typed `-o hihihi` and our executable would then have been named `hihihi`. The flags that we pass to a program are special examples of *command-line arguments*.

User Input

- To make our program more interesting, let's try asking the user for a name and saying hello to her. To do this, we need a place to store the user's name, i.e. a variable. A variable that stores a word or a phrase is known as a *string*. Let's call this variable `name`:

```c
#include <stdio.h>

int main(void)
{
    string name;
    name = GetString();
    printf("hello, David\n");
}
```

- Before we ask the user for her name, the variable `name` has no value. We shouldn't print it out as such.
- `GetString` is a function provided in the CS50 Library written by the staff. `GetString` takes in user input and passes it back to your program as a string. The `=` in this case is an assignment operator, meaning place in the left side the value of the right side.
- Now when we try to compile this program, we get all sorts of errors. When the compiler prints out this many errors, it's a good idea to work your way through them from top to bottom because the errors at bottom might actually have been caused by the errors at the top. The topmost error is as follows:

```
hello.c:5:5 error: use of undeclared identifier 'string': did you mean
'stdin'?
```

- No, we didn't mean `stdin`! However, the variable type `string` is actually not built in to C. It's available via the CS50 Library. To use this library, we actually need to tell our program to include it like so:

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

int main(void)
{
    string name;
    name = GetString();
    printf("hello, David\n");
}
```

- When we compile and run this, the program appears to do nothing: the cursor simply blinks. This is because it's waiting for the user to type something. When we type "Rob," the program still prints out "hello, David," which isn't quite what we intended. Let's add a line to clarify to the user that he's supposed to type something:

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

int main(void)
{
    string name;
    printf("What is your name?");
    name = GetString();
    printf("hello, David\n", );
}
```

- When we run the program, nothing seems to have changed. Oops, we forgot to recompile.
- One thing we can improve is to add a newline between the question and the blinking cursor. We do this by adding the `\n` character.
- What we really want is to print out the user-provided name, which is stored in `name`. Thus, we nee to pass `name` to `printf` like so:

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

int main(void)
{
    string name;
    printf("What is your name?\n");
    name = GetString();
    printf("hello, %s\n" name);
}
```

- What's between the parentheses after `printf` are the *arguments* that we pass it. Here, we pass two arguments. `%s` is a placeholder for the second argument, `name`, which gets inserted into the first argument.
- In addition to the CS50 Library, we're including `stdio.h`, the library the contains the definition of `printf`.

Loops

- Let's write a silly little program with an infinite loop:

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

int main(void)
{
    while (true)
    {
        printf("I am a buggy program");
    }
}
```

- Since the loop condition `true` is always true, the loop continues executing indefinitely. Compiling and running this program prints a whole lot of text to the terminal! You don't need to restart your Appliance to stop the program, just type Ctrl+C.
- Now let's write a counter program:

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

int main(void)
{
    for (int i = 0; i < 100; i++)
    {
        printf("I can count to %i\n", i);
    }
}
```

- Ignore the cryptic syntax for now, but know that this program counts (very fast) to 100. What if we made a mistake and typed `i >= 0` instead of `i < 100` as the second loop condition? We would unintentionally induce an infinite loop. On Wednesday we'll see if this program has finished!

---

1. Sunday Sunday Someday!
2. Except for once when David asked me what's the deal with the internet and cats. Duh.

Last updated 2013-09-13 18:39:33 PDT

Week 1, continued

Andrew Sellergren

Table of Contents

Announcements and Demos

- Devices like Google Glass come with an API, an *application programming interface*, that allow developers to write software for it. For Final Projects, we'll do what we can to hook you up with loaner hardware so that if you're interested in writing, say, an Android or iOS app, you'll have a test device.
- Another such device that comes with an API is Leap Motion. Check out this video to see how it allows you to control your computer with 3D gestures.
- Section by Friday at noon at cs50.net/section.
- One-time supersections will be held this Sunday. They will be filmed.
    - Less comfortable
        - Sun 9/15, 2pm, Yenching Auditorium
    - More comfortable

        - Sun 9/15, 4pm, Yenching Auditorium

- Take advantage of Office Hours!
- Problem Set 0 is due on Friday. This is a day later than the usual deadline. For Problem Set 1, you'll have the opportunity to extend the deadline from Thursday to Friday by completing some warm-up exercises.
- Get the CS50 Appliance at cs50.net/appliance. Problem Set 1's specification will walk you through setting it up.

From Last Time

- To translate source code into object code, we used a compiler. Inside your computer, the CPU knows what the 0's and 1's actually mean, whether it be print, add, subtract, etc.
- We ported our "hello, world" Scratch program to C.

hello, world!

- Let's begin to tease apart the syntax of our very first C program:

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

- The first line that begins with a `#` is called a *preprocessor directive*. In this case, the directive says to include the contents of a file `stdio.h` inside our program. Inside of `stdio.h` is stored the definition of `printf`.
- `int main(void)` is the equivalent of the "when green flag clicked" puzzle piece in Scratch. We'll wave our hands at what it actually means for now.
- Next we have open and close curly braces. These encapsulate related lines of code.
- The interesting line of code in this program is the `printf` line. Recall that "hello, world\n" is an example of a *string*. The `\n` is a newline character.
- How would we print out a double quote? If we simply write a double quote inside the two double quotes that enclose our string, the compiler will freak out. We need to*escape* it using a backslash, so we write `\"`.

Programming Constructs in C

Conditions

- Conditions in C look like so:

```
if (condition)
{
    // do this
}
```

- Note the encapsulation provided by the curly braces again. The `//` denotes a comment, a line of English that explains the code.
- A two-way fork in logic looks like this:

```
if (condition)
{
    // do this
}
else
{
    // do that
}
```

- A three-way fork in logic looks like this:

```
if (condition)
{
    // do this
}
else if (condition)
{
    // do that
}
else
{
    // do this other thing
}
```

- The quote strings and variables we've been passing to `printf` between the parentheses are known as *arguments*. An argument is a value that influences the behavior of the function.

Boolean Expressions

- The Boolean operators "and" and "or" are written as `&&` and `||` in C:

```
if (condition && condition)
{
    // do this
}
if (condition || condition)
{
    // do this
}
```

- Note that `&` and `|` have different meaning!

Switches

- *Switches* are another way of implementing forks in logic. Instead of repeated "else if" blocks, you can handle cases like so:

```
switch (x)
{
    case 1:
        // do this
        break;

    case 2:
        // do that
        break;

    default:
        // do this other thing
        break;
}
```

For Loops

- for loops take the following general structure:

```
for (initializations; condition; updates)
{
    // do this again and again
}
```

- Within the parentheses after the `for` keyword, there are three parts. Before the first semicolon, we are initializing a variable which will be our iterator or counter, often named `i` by convention. Between the two semicolons, we're providing a condition which, if true, will cause another iteration of the loop to be executed. Finally, we provide code to update our iterator.
- A snippet of code to print "hello, world!" ten times would look something like:

```
for (int i = 0; i < 10; i++)
{
    printf("hello, world!\n");
}
```

- Recall that `i++` is shorthand for "increment the value of `i` by 1." This loop continues executing so long as `i < 10`, which happens 10 times.
- A while loop is functionally equivalent to a for loop, albeit with slightly different syntax. Consider this code that prints out "hello, world!" infinitely:

```
while (true) {
    print("hello, world\n");
}
```

- As soon as the condition between the parentheses after `while` evaluates to false, the loop will terminate. In this case, however, `true` will never be false, so the loop continues forever.
- A do while loop executes before checking the condition it depends on. That means it will always execute at least once. The general structure is as follows:

```
do
{
    // do this again and again
}
while (condition);
```

Variables

- Syntactically, declaring a variable looks like so:

```
int counter;
counter = 0;
```

- *Declaring a variable* means asking the computer for some number of bits in which to store a value, in this case the value of an integer. The second line of code above*assigns* the value 0 to the variable `counter`. `=` is the assignment operator and instructs the computer to put the right in the left.
- We can do variable declaration and assignment all in one line:

```
int counter = 0;
```

- This one is a little easier to read and should be considered best practice.

Functions

- A *function* is a piece of code that can take input and can produce output. In some cases, a function can be a so-called *black box*. This means that the details of its implementation aren't relevant. We don't care **how** it does what it does, just that it does it.

- Let's represent `printf` with an actual black box onstage. We can write "hello, world" on a piece of paper to represent an argument to `printf`. We then place this piece of paper in the black box and, by whatever means, the words "hello, world" appear on the screen!
- To make our `hello` program more dynamic, we asked the user for his or her name and passed that to `printf`:

```
string name = GetString();
printf("hello, %s\n", name);
```

- `printf` doesn't return anything; it only has the *side effect* of printing to the screen. `GetString`, on the other hand, returns what the user typed in.
- As with `printf`, we don't necessarily care how `GetString` is implemented. We know that when we call it, we'll be provided with a string after some amount of time. We can simulate this by retrieving from the black box a piece of paper with a student's name (Obasi) written on it. We actually then make a copy of this string before storing it in `name`.
- Now we have `name` written on one piece of paper which will act as the second argument to `printf`. Next we create the first argument by writing "hello, %s\n" on another piece of paper. Finally, we place these two pieces of paper in the black box and magically, "hello, Obasi" appears on the screen.
- Functions that we implemented in the CS50 Library (`cs50.h`) include:
    - `GetChar`
    - `GetDouble`
    - `GetFloat`
    - `GetInt`
    - `GetLongLong`
    - `GetString`
- Convention holds that C function names are lowercase, but we capitalized these just to make it clear that they belong to the CS50 Library.
- A float is a number with a decimal point. A double is a number with a decimal point but with more numbers after the decimal point. These types returned by CS50 Library function require different number of bits to be stored. A `char` requires 8 bits, a `float` requires 32 bits, and a `double` requires 64 bits. A `long long` is an integer that is twice as big in memory (64 bits) as an `int` (32 bits). More on these types later.
- The CS50 Library also contains two custom types:
    - `bool`
    - `string`
- For convenience, we have created the symbols `true` and `false` to represent 1 and 0. Likewise for convenience, we have created a `string` type to store strings.

- The actual types of variables available in C are as follows:
    - `char`
    - `double`
    - `float`
    - `int`
    - `long long`
- The `printf` function can take many different formatting characters. Just a few of them are:
    - `%c` for `char`
    - `%i` (or `%d`) for `int`
    - `%f` for `float`
    - `%lld` for `long long`
    - `%s` for `string`
- A few more escape sequences:
    - `\n` for newline
    - `\r` for carriage return (think typewriter)
    - `\'` for single quote
    - `\"` for double quote
    - `\\` for backslash
    - `\0` for null terminator

Hellos and More

hello-0.c

- As before, we'll open a file in gedit on the Appliance and save it to the `jharvard` directory. We'll call this file `hello-0.c` and write the following therein:

```
int main(void)
{
    printf("hello, world\n");
}
```

- When we type `make hello-0` in the terminal window at bottom, we get all sorts of compiler errors. At the top of these errors, which tend to compound each other, we see:
```
...implicitly declaring library function 'printf'...
```
- This error message may seem overwhelming, but try looking for keywords. Right away we notice `printf`. We forgot to include the library that contains the definition of `printf`:

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

`hello-1.c`

- To say hello to the user, we need a variable to store his or her name:

```
#include <stdio.h>

int main(void)
{
    string name = "David";
    printf("hello, %s\n", name);
}
```

- For now, we hardcode the value "David" into the variable `name`. This time when we compile (simply hit the up arrow to see previous commands in Linux), we get the following error:
  `...use of undeclared identifier 'string'`
- We need to also include the CS50 Library:

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

int main(void)
{
    string name = "David";
    printf("hello, %s\n", name);
}
```

- Sigh, still more compiler errors. The topmost one says:
  `...multi-character character constant...`
- That doesn't help us too much, but `clang` does point us to the problem are with a green caret. Turns out that in C, strings must be delimited by double quotes, not single quotes:

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

int main(void)
{
    string name = "David";
    printf("hello, %s\n", name);
}
```

`hello-2.c`

- Finally, let's actually accept dynamic input from the user:

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

int main(void)
{
    printf("State your name: ");
    string name = GetString();
    printf("hello, %s\n", name);
}
```

- This program compiles and functions correctly for normal names like Rob, Lauren, and Joseph. What about an empty name? It just prints out "hello, "; perhaps we should use a condition and a loop so that we keep prompting the user until he provides a non-empty name.

adder.c

- If we want to ask the user for integer input, we can use the `GetInt` function:

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

int main(void)
{
    // ask user for input
    printf("Give me an integer: ");
    int x = GetInt();
    printf("Give me another integer: ");
    int y = GetInt();

    // do the math
    printf("The sum of %i and %i is %i!\n", x, y, x + y);
}
```

- Notice that we don't need a separate variable to store the sum, we can inline `x + y`.

conditions-0.c

- `conditions-0.c` has a subtle bug in it:

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

int main(void)
{
    // ask user for an integer
    printf("I'd like an integer please: ");
    int n = GetInt();

    // analyze user's input (somewhat inaccurately)
    if (n > 0)
    {
        printf("You picked a positive number!\n");
    }
    else
    {
        printf("You picked a negative number!\n");
    }
}
```

- Although this program is syntactically correct (it compiles and runs), it is logically incorrect (it declares 0 a negative number).

conditions-1.c

- To handle the corner case of 0, we need another fork in logic:

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

int main(void)
{
    // ask user for an integer
    printf("I'd like an integer please: ");
    int n = GetInt();

    // analyze user's input
    if (n > 0)
    {
        printf("You picked a positive number!\n");
    }
    else if (n == 0)
    {
        printf("You picked zero!\n");
    }
    else
    {
        printf("You picked a negative number!\n");
    }
}
```

- Note that to test equality, we use `==`, not `=`, which is the assignment operator.

Teaser

- It turns out that computers cannot express some values perfectly precisely. The protagonists in the movie Office Space take advantage of this imprecision to rip off their company Initech. Consider that if banking software stores a number like 0.1 improperly, it could mean that there are fractions of a cent gained or lost. If you haven't seen Office Space, that's your homework for the weekend.

Last updated 2013-09-13 22:01:40 PDT

Week 2

Andrew Sellergren

Table of Contents

Announcements and Demos

- If you're interested in teaching computer science to local middle school students, either now or sometime in the future, head to DLP.io/volunteer.
- Sections begin this coming Monday, Tuesday, and Wednesday. You'll hear later this week via e-mail about your assignment and how to change it if necessary.
- Problem Set 1 is due this Thursday. Included in the specification are instructions for how to extend the deadline to Friday as well as how to use the CS50 Appliance and tools like `check50` and `style50`.
- Problem set scores are calculated as follows:
    - scope * (correctness * 3 + design * 2 + style * 1)

      Scope captures how much of the problem set you bit off. Correctness implies whether your program works as per the specification. Design is a subjective assessment of the approach you took to solve the problem. Style refers to the aesthetics of your code: are your variables named descriptively, is your code indented properly?

    - each axis ranges from 1 to 5:
        - 5 - best
        - 4 - better
        - 3 - good
        - 2 - fair
        - 1 - poor

          Never fear that a 3 out of 5 is a 60% and thus failing! A 3 out of 5 is just what it says: good.

- A word on academic honesty. This course has the distinction of having sent more students to the Ad Board than any other course. We're very good at detecting*dishonesty*. We compare all problem sets pairwise against each other across this year and all prior years. If you can Google a solution, so can we. This year, we've rephrased our statement on academic honesty to capture the spirit of appropriate collaboration:
    - This courses's philosophy on academic honesty is best stated as "be reasonable."
    - Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraits.
    - See cs50.net/syllabus for examples of "reasonable" and "unreasonable" behavior.
    - If you're absolutely stuck and at your breaking point, reach out to David or the heads directly. Don't resort to unreasonable behavior!

From Last Time

- We introduced a few data types in C:
    - `char`
    - `double`
    - `float`
    - `int`
    - `long long`
- A `char` is an ASCII character. Recall in Week 0 our 8 volunteers who came on stage and raised their hands to represent bits. Together, they represented an entire byte, which, thanks to ASCII, can represent an alphabetical character. With 8 bits, it is possible to represent `2^{8}`, or 256, different characters. To represent the whole of the alphabet in a

language such as Arabic, we need an encoding system other than ASCII. More on that later.

- An `int` is an integer. On most computers, an `int` requires 32 bits and thus can represent roughly `2^{32}`, or 4 billion, different numbers.
- A `long long` requires 64 bits and can represent even bigger numbers than an `int`.
- The `float` and `double` types, which require 32 bits and 64 bits, respectively, store numbers with decimal points.
- One problem you may have already picked up on is that we have a finite number of bits that we can use to represent an infinite number of numbers.

Floating Points

floats-0.c

- Let's write a short program we'll call `floats-0.c`:

```
#include <stdio.h>

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

- Here we're simply trying to store the value 0.1 in a `float` and print it out. Don't forget `stdio.h`! The `.1` in front of `f` means that we want to print only one decimal place.
- When we compile and run this program, however, we see 0.0 printed to the screen. Where is the bug? Let's try printing two decimal places by writing `%.2f`. Nope, we get 0.00.
- The problem is that we're dividing one integer by another. When you do this, the computer assumes that you want another integer in response. Since 0.1 is not an integer, the computer actually truncates it, throwing away everything after the decimal point. When we actually store the resulting integer in a `float`, it gets converted to a number that has a decimal point.

floats-1.c

- To fix this, we could turn the integers into floating points like so:

```
#include <stdio.h>

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

floats-2.c

- Alternatively, we could explicitly *cast*, or convert, the numbers 1 and 10 to floating points before the division:

```c
#include <stdio.h>

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

- Since characters are represented as numbers via ASCII, we can use casting to convert between `char` and `int`. Similarly, we can cast between different types of numbers.
- What if we change `%.1f` to `.10f` or `.20f`? We don't get exactly 0.1, but rather 0.1 with some numbers in the far right decimal places. Because the `float` type has a finite number of bits, we can only use it to represent a finite number of numbers. At some point, our numbers become imprecise.
- Don't think that this imprecision is a big deal? Perhaps this video will convince you otherwise.
- As we'll find out later in the semester, MySQL requires you to specify how many bits it should use to store values.

More From Last Time

- Check out last week's notes for the syntax we used for conditions, Boolean expressions, switches, loops, variables, and functions in C.
- Recall that `printf` was a function that had no return value (or at least none that we cared about), but only a *side effect*, that of printing to the screen.

do-while

- One type of loop we didn't look closely at is the do-while loop. Let's write a program that insists that the user give a positive number:

```
#include <stdio.h>

int main(void)
{
    printf("I demand that you give me a positive integer: ");
    int n = GetInt();
    if (n <= 0)
    {
        printf("That is not positive!\n")
    }
}
```

- But now if the user hasn't given us a positive number, we need to copy and paste the `GetInt()` call into another branch of logic to re-prompt her. But what if she *still* hasn't given us a positive number? Obviously this could go on forever, so we probably need some kind of loop instead. Let's try a do-while loop:

```
#include <stdio.h>

int main(void)
{
    do
    {
        printf("I demand that you give me a positive integer: ");
        int n = GetInt();
    }
    while (n <= 0);
    printf("Thanks for the %d!\n", n);
}
```

- Notice how much improved this is! When we try to compile, though, we get an "implicit declaration" error. We need to include the CS50 Library.

Scope

- Even after adding `#include <cs50.h>` we get "unused variable n" and "undeclared identifier n" errors. It would seem that we are in fact using the variable `n` when we check whether it's less than or equal to zero. Likewise it would seem that `n` is not "undeclared" since we initialized it within the do block. What's wrong then? Because we're declaring `n` inside the do block, within the curly braces, its *scope* is limited to that block. Outside of those curly braces, `n` effectively doesn't exist. What we need to do is declare `n` outside the loop but set its value within the loop like so:

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

int main(void)
{
    int n;
    do
    {
        printf("I demand that you give me a positive integer: ");
        n = GetInt();
    }
    while (n <= 0);
    printf("Thanks for the %d!\n", n);
}
```

- If we compile and run this program, we find that it works!
- Let’s try banging on it a little. What happens if we type a character instead of a number? We get a "Retry: " prompt. Since this doesn’t appear in our program above, it presumably comes from some error checking in the CS50 Library. When you call `GetInt()`, we at least make sure that what you get in return is an `int`, not a`string` or a `char`.
- A suboptimal solution to this problem of scope would have been to declare a *global variable* like so:

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

int n;

int main(void)
{
    do
    {
        printf("I demand that you give me a positive integer: ");
        n = GetInt();
    }
    while (n <= 0);
    printf("Thanks for the %d!\n", n);
}
```

- Scratch actually implemented global variables as variables declared for "all sprites."
- Global variables are generally considered poor design.
- Note that declaring a variable and not using it is not strictly an error. However, for CS50, we’ve cranked up the error checking of the compiler as a pedagogical exercise. You may have noticed a series of flags that are

passed to `clang` automatically when you type `make`. Two of those flags are `-Wall -Werror` which mean "make all warnings into errors."

Strings

- There’s a lot more going on under the hood with strings than we’ve let on so far. Consider the following program:

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

int main(void)
{
    printf("Please give me a string: ");
    string s = GetString();
    for (int i = 0; i < strlen(s); i++)
    {
        printf("%c\n", s[i]);
    }
}
```

- `strlen` returns the length of the string it is passed. Thus, we seem to be looping through the characters of the string.
- It turns out that strings are stored as characters back-to-back in memory. We can access those characters using the square bracket notation, so `s[0]` gets the first letter, `s[1]` gets the second letter, and so on.
- This program prints the characters of the user-provided string, one line at a time!

Teaser

- Check out the first jailbreak of the iPhone, the winner of an obfuscated C contest, and a very pretty program!

Last updated 2013-09-19 08:08:49 PDT

Week 2, continued

Andrew Sellergren

Table of Contents

Announcements and Demos

Functions

function-0.c

- Thus far, we've used functions like `printf`, which prints to the screen, and `GetString`, which gets a string from the user, but what if we want to write our own function? Let's revisit this program that prompts a user for his name and try to factor out some of the logic:

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

void PrintName(string name)
{
    printf("hello, %s\n", name);
}

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

- Although this function is actually only one line of code, it's much nicer to write just the function name than to copy and paste that one line when we want to reuse it.
- Above the definition of `main`, we declared our function `PrintName` to have a return type of `void` (because it doesn't return anything but has a side effect) and a single argument `name` of type `string`.
- Notice that although we call the user's name `s` in `main`, we refer to it as `name` in `PrintName` because that was what we chose to call the argument.
- What if we had chosen to write the definition of `PrintName` after `main`? The compiler would have complained of "implicit declaration of function PrintName" because it reads top to bottom and we called `PrintName` before we defined it. Better than defining it above `main`, however, is declaring it above `main` and then defining it below. This keeps `main` at the top of the file:

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

void PrintName(string name);

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

void PrintName(string name)
{
    printf("hello, %s\n", name);
}
```

- This declaration of function `PrintName` is called a *prototype*.

function-1.c

- Let's rewrite our program that asks for a positive integer using a function:

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

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

- At this stage, we’re going to get an "implicit declaration" compiler error again because we haven’t defined `GetPositiveInt`. We need to put the prototype at the top like so:

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

int GetPositiveInt(void);

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

- As its name suggests, `GetPositiveInt` returns an `int`. Because it doesn’t take any arguments, we’ve explicitly declared it as taking `void`, meaning nothing. Now for the definition of `GetPositiveInt`:

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

int GetPositiveInt(void);

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

int GetPositiveInt(void)
{
    int n;
    do
    {
        printf("Give me a positive integer: ");
        n = GetInt();
    }
    while (n <= 0);
}
```

- When we compile this program, though, we get an error: "control reaches end of non-void function." What this means is that `GetPositiveInt` isn’t returning anything even though we’ve specified that it should return an `int`. To fix this, we add a return statement:

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

int GetPositiveInt(void);

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

int GetPositiveInt(void)
{
    int n;
    do
    {
        printf("Give me a positive integer: ");
        n = GetInt();
    }
    while (n <= 0);
    return n;
}
```

- Question: `main` has a return type of `int` too, so why aren't we returning anything from it? Technically, in the version of C we're using, 0 is returned by default at the end of `main`. 0 means nothing went wrong, whereas each non-zero return value could represent a different error code.

Strings

- Last time, we saw that we can think of strings as collections of contiguous boxes, each containing a single character requiring a single byte of memory. To access the individual characters/bytes of a string, we index into the string using the square bracket notation.
- Realize that there are two types of memory in your computer: disk, where you store your music and photos, etc., and RAM, or random access memory, which store information that needs to be accessed quickly while a program is running. The memory that stores a string like "hello" for your program is RAM.

`string-1.c`

- Consider again the program that takes a string from the user and prints it one character per line:

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

- What's the deal with the `s != NULL`? It turns out that the `GetString` will not always succeed in getting a string from the user. If it fails, perhaps because the user typed a string that was too long to hold in memory, `GetString` will return a special *sentinel value* named `NULL`. Without this check, other things we try to do with `s`might cause the program to crash.
- `s[i]`, where `i` is 0, 1, 2, is an individual character in the string, so we ask `printf` to substitute it into `%c`.

string-2.c

- There's at least one inefficiency in this `string-1.c` as it's currently written. `strlen(s)` will only ever return one value, yet we're calling it on every iteration of the loop. To optimize our program, we should call `strlen` once and store the value in a variable like so:

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

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

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

- Although computers are very fast these days and this optimization may not be immediately noticeable, it's important to look for opportunities to improve design. These little optimizations can add up over time. One of the problem sets we've done in years past was writing a spellchecker in C with the goal of making it as fast as possible. An optimization like this might save a few milliseconds of runtime!

capitalize-0.c

- In `capitalize-0.c`, we claim to capitalize all the letters in a string:

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

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

    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");
}
```

- The first condition within the loop is checking if the character is lowercase. If it is, then we subtract from it the value of `a - A`. Under the hood, characters are actually just numbers, which is why we can compare them with `>=` and subtract them from each other. The value of `a - A` is the offset between the lowercase and uppercase characters on the ASCII chart. By subtracting this offset from `s[i]`, we're effectively capitalizing the letter.

capitalize-1.c

- Thus far, we've worked with a few libraries of code that gave us convenient functions. Let's add one more to that list:
  - `stdio.h`
  - `cs50.h`
  - `string.h`
  - `ctype.h`
- Rather than write our own condition to check if a character is lowercase and our own logic to capitalize a character, let's use functions someone else already implemented:

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

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

    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");
}
```

capitalize-2.c

- We don’t strictly need the curly braces around if-else blocks so long as they are only a single line. However, there’s an even better way to shorten this program:

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

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

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

- Turns out that `toupper` handles both lowercase and uppercase characters properly, so we don’t even need the `islower` check. We know this because we checked the man page, or manual page, for `toupper` by typing `man toupper` at the command line. This page tells us that the return value is the converted letter or the original letter if conversion was not possible. Perfect!

The Null Terminator

- If strings are stored as characters back to back in memory, how do we know where one ends and another begins? When inserting a string into memory, the computer actually adds a special character called NUL after the last character of the string. NUL is represented as `\0`.

Arrays

`ages.c`

- Strings are actually a special case of a data type called an array. Arrays allow us to store related variables together in one place. For example, consider a program that stores and prints out the ages of everyone in the room:

```c
 1 #include <cs50.h>
 2 #include <stdio.h>
 3 
 4 int main(void)
 5 {
 6     // determine number of people
 7     int n;
 8     do
 9     {
10         printf("Number of people in room: ");
11         n = GetInt();
12     }
13     while (n < 1);
14 
15     // declare array in which to store everyone's age
16     int ages[n];
17 
18     // get everyone's age
19     for (int i = 0; i < n; i++)
20     {
21         printf("Age of person #%i: ", i + 1);
22         ages[i] = GetInt();
23     }
24 
25     // report everyone's age a year hence
26     printf("Time passes...\n");
27     for (int i = 0; i < n; i++)
28     {
29         printf("A year from now, person #%i will be %i years old.\n", i + 1, ages[i] + 1);
30     }
31 }
```

- The first lines should be familiar to you by now: we're prompting the user for a positive number. In line 16, we use that number as the number of places in our array called `ages`. `ages` is a bucket with room for `n` integers. Using an array is a better alternative than declaring an `int` for every single person in the room, especially since we don't even know how many there are until the user tells us!

- The rest of the program is pretty straightforward. We iterate through `ages` the same way we iterated through strings, accessing each element using square bracket notation.

Cryptography

- Can you guess what this encrypted string actually says?

  Or fher gb qevax lbhe Binygvar

- It says "Be sure to drink your Ovaltine." Each character of the string is changed to another character using an encryption technique known as ROT-13. Each letter is simply rotated by 13 around the alphabet. This isn't very sophisticated, of course, as there are only 25 different numbers to rotate by, so it can easily be cracked by brute force.

- On certain systems, your password might be stored as an encrypted string like so:

- Your password was encrypted using secret-key cryptography. That means it was translated from plaintext to so-called *ciphertext* using a secret. Only with that secret can the ciphertext be decrypted back to plaintext. In the Hacker Edition of the upcoming problem set, you'll be asked to decrypt some passwords without knowing the secret! In the Standard Edition, we'll introduce you to some ciphers, one called Caesar and one called Vigenère.

- Be sure to drink your Ovaltine!

Last updated 2013-09-21 20:44:38 PDT

Week 3

Andrew Sellergren

Table of Contents

Announcements and Demos

- Please welcome special guest lecturer Rob Bowden!
- No CS50 Lunch this Friday!

From Last Time

- We've seen how a string is a sequence of characters. More properly speaking, it is an array of characters with a null terminator (`\0`) at the end.
- We can also have arrays of other types, not just characters. In the `ages.c` program, we used an array of `int` to store the age of everyone in the room:

```
 1 #include <cs50.h>
 2 #include <stdio.h>
 3 
 4 int main(void)
 5 {
 6     // determine number of people
 7     int n;
 8     do
 9     {
10         printf("Number of people in room: ");
11         n = GetInt();
12     }
13     while (n < 1);
14 
15     // declare array in which to store everyone's age
16     int ages[n];
17 
18     // get everyone's age
19     for (int i = 0; i < n; i++)
20     {
21         printf("Age of person #%i: ", i + 1);
22         ages[i] = GetInt();
23     }
24 
25     // report everyone's age a year hence
26     printf("Time passes...\n");
27     for (int i = 0; i < n; i++)
28     {
29         printf("A year from now, person #%i will be %i years old.\n",
   i + 1, ages[i] + 1);
30     }
31 }
```

- In line 16, we use bracket notation to declare an array of `int` of size `n`. To access the element `i` of that array, we write `ages[i]`. Arrays are zero-indexed, so the first element is stored in `ages[0]`. There's no element in `ages[n]` even though the array has size `n`!

Command-line Arguments

- Thus far, we've started our programs with the line `int main(void)`. `void` means that we're not passing any arguments to the `main` function. However, it's possible to pass arguments to the `main` function using the following syntax:
- `int main(int argc, string argv[])`
- Here, we pass two arguments, `argc` and `argv`. `argc` is an `int` and `argv` is an array of `string`. `argc` actually indicates how many arguments we've passed to a program.
- Earlier, when we typed `./ages` at the command line, there was 1 command-line argument: the name of the program itself. If we had typed `./ages hello world`, there would have been 3 command-line

arguments. In these cases, `argc` would have taken the values 1 and 3, respectively. There is always at least 1 command-line argument.

- Whereas `argc` contains the number of command-line arguments, `argv` contains the command-line arguments themselves.

argv-0.c

- Listen to David describe a short program that uses command-line arguments:

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

int main(int argc, string argv[])
{
    printf("%s\n", argv[1]);
}
```

- We could modify this program to print out the second command-line argument as an integer:

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

int main(int argc, string argv[])
{
    int x = atoi(argv[1]);
    printf("%d\n", x);
}
```

- This works well as long as we actually send two command-line arguments. What happens when we don't? The program crashes with a *segmentation fault*. This is because we tried to access an element beyond the bounds of the array. We should add a check to protect against this:

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

int main(int argc, string argv[])
{
    if (argc < 2)
    {
        printf("Not enough command line arguments\n");
    }
    int x = atoi(argv[1]);
    printf("%d\n", x);
}
```

argv-1.c

- Let’s try printing out all of the command-line arguments:

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

int main(int argc, string argv[])
{
    // print arguments
    for (int i = 0; i < argc; i++)
    {
        printf("%s\n", argv[i]);
    }
}
```

argv-2.c

- Now, to go even deeper [1], let’s print each character of each command-line argument on its own line:

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

int main(int argc, string argv[])
{
    // print arguments
    for (int i = 0; i < argc; i++)
    {
        for (int j = 0, n = strlen(argv[i]); j < n; j++)
        {
            printf("%c\n", argv[i][j]);
        }
    }
}
```

- `argv[i][j]` represents character `j` in command-line argument `i`.
- How would we go about writing `strlen` if it weren’t provided to us in the library `string.h`? Recall that all strings end with the special `\0` character. If we iterate through each character of the string until we find `\0`, we’ll know the length of the string:

```c
int my_strlen(string s)
{
    int length = 0
    while(s[length] != '\0')
    {
        length++;
    }
    return length;
}
```

- We could even move this logic into our second loop and avoid a function call altogether:

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

int main(int argc, string argv[])
{
    // print arguments
    for (int i = 0; i < argc; i++)
    {
        for (int j = 0; argv[i][j] != '\0'; j++)
        {
            printf("%c\n", argv[i][j]);
        }
    }
}
```

- Each time we reach a `\0` within `argv`, our array of array of characters, we terminate the inner for loop and execute the outer for loop again (provided that we still have command-line arguments left, i.e. `i < argc`).

Debugging

debug.c

- The best way to learn to debug is to work with buggy code like the following:

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

void foo(int i)
{
    printf("%i\n", i);
}

int main(void)
{
    printf("Enter an integer: ");
    int i = GetInt();

    while (i > 10)
    {
        i--;
    }

    while (i != 0)
    {
        i = i - 3;
    }

    foo(i);
}
```

- Let’s assume that the user provides an integer greater than 10 (a bad assumption). The first while loop will then decrement `i` by 1 until it equals 10, at which point the loop condition `i > 10` will no longer be true and the loop will exit. The second while loop will decrement `i` by 3 until it equals 0. But if it starts at 10, `i` will go to 7, then 4, then -1, then -4, and so on to negative infinity. That’s not what we intended!
- One useful debugging technique is to add some `printf` statements:

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

void foo(int i)
{
    printf("%i\n", i);
}

int main(void)
{
    printf("Enter an integer: ");
    int i = GetInt();

    printf("Outside first while loop");
    while (i > 10)
    {
        printf("First while loop: %i\n", i);
        i--;
    }

    printf("Outside second while loop");
    while (i != 0)
    {
        printf("Second while loop: %i\n", i);
        i = i - 3;
    }

    foo(i);
}
```

- When we compile and run `debug.c` now, we can clearly see that the second while loop is infinite.
- As your programs get longer and more complicated, you’ll find more sophisticated debugging techniques more useful.

GDB

- GDB, or GNU Debugger, is a program that allows your to step through your code line by line while it's executing. Whenever you have run `make` on your code thus far, the compiler `clang` has executed with a command-line argument `-ggdb3`. This instructs `clang` to compile your code so that you can debug it within GDB if you so choose.
- After you've compiled `debug.c`, you can open it in GDB by executing `gdb debug`. Note that the name of the program you want to walk through, in this case `debug`, is provided as a command-line argument to `gdb`.
- Once you've started GDB, you'll find yourself at a prompt. Type `run` to begin the execution of your program. This alone isn't very useful, as the program will finish executing just as it would outside of GDB.
- Before typing `run`, you can type `break main` to insert a *breakpoint* at the beginning of the `main` function. A breakpoint is a place at which execution of the program is paused. Once you've reached a breakpoint, you can type `next` repeatedly to step through your code line by line. Hitting Enter will redo the last command you typed.
- Whenever we want to view the value of `i`, we can type `print i`.
- `list` will display the code before and after your current position.
- After stepping forward through many more executions of the second loop, printing `i` will give a negative number.
- Let's move the infinite loop inside the `foo` function for more practice with GDB:

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

void foo(int i)
{
    while (i != 0)
    {
        i = i - 3;
    }
    printf("%i\n", i);
}

int main(void)
{
    printf("Enter an integer: ");
    int i = GetInt();

    while (i > 10)
    {
        i--;
    }

•     foo(i);
• }
```

- After starting GDB by executing `gdb debug`, we'll again set a breakpoint at the beginning of `main` by typing `break main`. If we `next` over the `foo` function call, the program will finish executing. Note that although the second while loop appears infinite, it will eventually terminate for reasons we'll wave our hands at for now.
- If we want to examine what `foo` is doing, we need to type `step` instead of `next`. `step` is identical to `next` if the next line of code is not a function call.
- We can also set a breakpoint at the `foo` function by typing `break foo`. Now when we type `run`, we'll first stop at `main`. If we type `continue`, execution will resume until we hit the next breakpoint at `foo`.

Security

- In Problem Set 2, you'll be working with encrypting and decrypting passwords. Of course, the strength of the encryption doesn't matter all that much if you choose aweak password.

- Consider the code that implements the login prompt on your laptop. If it's working properly, it will check that what the user types matches your password before letting the user in. However, if it's working maliciously, it might check that what the user types matches some master password that lets **anyone** in. We can hope that this *backdoor* might be caught by at least one of the many people who have reviewed it.

- But what if the malicious code is in the compiler? Then the compiler might actually insert the backdoor into the login prompt even though the code for the login prompt seems safe and has been reviewed. We can hope again that this would be caught by one of the people who reviewed the code for the compiler.

- But what if the malicious code is in the compiler that is used to compile the compiler? Well, then, the backdoor might get inserted without anyone knowing.

- If you think this scenario is unlikely, consider the speech that Ken Thompson gave, Reflections on Trusting Trust, when he accepted the Turing Award (more or less the Nobel Prize of computer science). In it, he describes this exact technique for compromising a compiler so that it would introduce a backdoor into a login program. The login program he refers to, however, is not some toy program, but rather the login program for all of UNIX. Since delivering this speech, Thompson has confirmed that this exploit was actually implemented and released to at least one company, BBN Technologies.

---

1. Even deeper.

Last updated 2014-03-19 09:24:33 PDT

Week 3, continued

Andrew Sellergren

Table of Contents

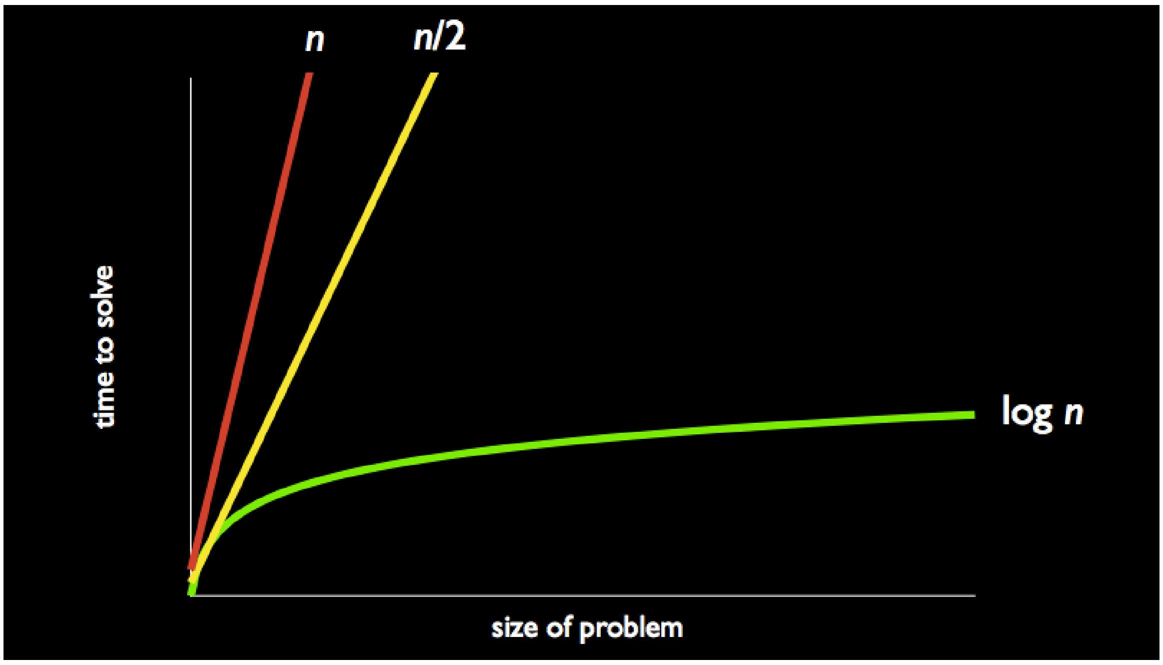
Announcements and Demos

- If you’re struggling with the C syntax, don’t worry, it’s normal! In a few weeks, you’ll look back on Problem Set 2 and be amazed at how far you’ve come.

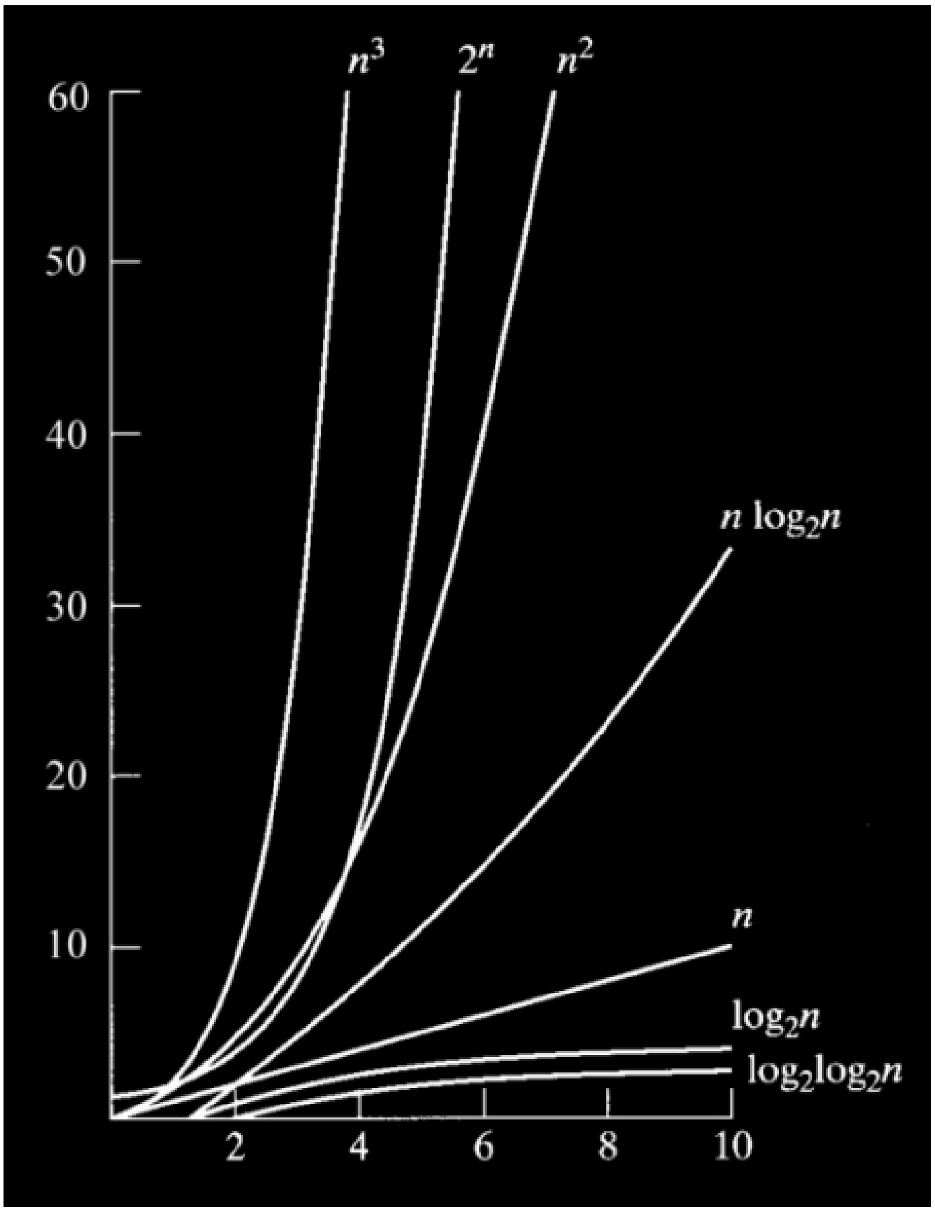
Searching

- Imagine there are 7 doors with numbers behind them and you want to find the number 50. If you know nothing about the numbers, you might just have to open the doors one at a time until you find 50 or until all the doors are opened. That means it would take 7 steps, or more generally, *n* steps, where *n* is the number of doors. We might call this a *linear* algorithm.
- How might our approach change if we know that the numbers are sorted? Think back to the phonebook problem. We can continually

divide the problem in half! First we open the middle door. Let's say it's 16. Then we know that 50 should be in the doors to the right of the middle, so we can throw away the left. We then look at the middle door in the right half and so on until we find 50! This algorithm has *logarithmic* running time, the green line in the graph below:



- Note that there are plenty of algorithms that are much worse than linear, as this graph shows:



- Although it looks like n^3 is the worst, 2^n is much worse for large inputs.

Sorting

Bubble Sort

- If the numbers aren't sorted to begin with, how much time will it take to sort them before we search?
- To start figuring this out, let's bring 7 volunteers on stage and have them hold pieces of paper with the numbers 1 through 7 on them. If we ask

them to sort themselves, it seems to only take 1 step as they apply an algorithm something like "if there’s a smaller number to my right, move to the right of it." In reality, though, it takes more than 1 step as there are multiple moves going on.

- In order to count the number of steps this algorithm takes, we’ll slow it down and allow only 1 move to happen at a time. So walking left to right among the volunteers, we examine the two numbers next to each other and if they’re out of order, we swap them. We may have to walk left to right more than 1 time in order to finish sorting. How do we know when they’re sorted? As a human, you can look at it and know, but we need a way for the computer to know. If we walk left to right among the volunteers and make 0 swaps, then we can be sure that all the numbers are in the right order. That means we’ll need to store the number of swaps made in a variable that we check after each walkthrough.
- This algorithm we just described is called *bubble sort*. To describe its running time, let’s generalize and say that the number of volunteers is *n*. Each time we walk through the volunteers, we’re taking *n*-1 steps. Let’s just round that up and call it *n*. How many times do we walk left to right through the volunteers? In the worst case scenario, the numbers will be perfectly out of order, that is, arranged left to right largest to smallest. In order to move the 1 from the right side all the way to the left side, we’re going to have to walk through the volunteers n times. So that’s *n* steps per walkthrough and *n* walkthroughs, so the running time is n^2.

Selection Sort

- Another approach we might take is to walk through the volunteers left to right and keep track of the smallest number we find. After each walkthrough, we then swap that smallest number with the leftmost number that hasn’t been put in its correct place yet. On each walkthrough, we start one position to the right of where we started on the walkthrough before so as not to swap out a number we already put in its correct position.
- This algorithm is called *selection sort*. Here’s what our numbers look like after each walkthrough:

```
• 4 2 6 1 3 7 5
• 1 2 6 4 3 7 5
• 1 2 6 4 3 7 5
• 1 2 3 4 6 7 5
• 1 2 3 4 6 7 5
• 1 2 3 4 5 7 6
  1 2 3 4 5 6 7
```

- So which algorithm is faster, bubble sort or selection sort? With selection sort, we're again going to have to do *n* walkthroughs in the worst case. On the first walkthrough, we take *n*-1 steps. On the second walkthrough, because we know that the leftmost number is in its correct position, we start at the second lefmost and we take *n*-2 steps. On the third walkthrough, we take *n*-3 steps. And so on. So our total running time is *n*-1 + *n*-2 + *n*-3 + *n*-4... which works out to be (*n*(*n*-1)) / 2. Although this is less than n^2, as *n* gets very large, the difference between the two is negligible. So we say that selection sort's running time is also n^2 and thus it has the same running time as bubble sort.

Insertion Sort

- Our next approach will be to sort the numbers in place. When examining each element, we place it in its correct position in a smaller list on the left that we'll call the sorted list. Observe:

- `  4 2 6 1 3 7 5`
- `4   2 6 1 3 7 5`
- `2 4   6 1 3 7 5`
- `2 4 6   1 3 7 5`
- `1 2 4 6   3 7 5`
- `1 2 3 4 6   7 5`
- `1 2 3 4 5 6   7`
  `1 2 3 4 5 6 7`
- Note that the space is merely to delimit the sorted list from the rest of the list, but the total list is still only size 7.
- This algorithm might appear to be faster because we only walk through the list once. However, consider what happens when we have to insert 1 at the beginning of the sorted list. We have to move 2, 4, and 6 to the right to make room. In the worst case, we'll have to shift the entire sorted list every time we need to insert a number. When you consider this, you can deduce that the running time of this algorithm, *insertion sort*, is also n^2.
- To see these algorithms in action, check out this visualization. You may need to use Firefox and zoom in to see the control buttons.

Big O Notation

- Let’s summarize the running times of the algorithms we’ve discussed so far using a table. In so-called big O notation, O represents the worst-case running time and Ωrepresents the best-case running time.

	Ω	O
linear search	1	n
binary search	1	$\log n$
bubble sort	n	n^2
selection sort	n^2	n^2
insertion sort		n^2

- In the best case for linear and binary search, the number you’re looking for is the first one you examine, so the running time is just 1. In the best case for our sorting algorithms, the list is already sorted, but in order to verify that in bubble sort, we need to walk through the list at least once. Unfortunately, to verify that in selection sort, we still have to do n^2 walkthroughs, each of which confirms that the smallest number is in the correct position.
- What about the best case for insertion sort? We’ll fill in that blank next time.
- Are we doomed to n^2 running time for sorting? Definitely not. Check out this visualization to see how fast merge sort is compared to bubble sort, selection sort, and insertion sort. Merge sort leverages the same "divide and conquer" technique that binary search does.

Last updated 2013-09-27 19:50:45 PDT

Week 4

Andrew Sellergren

Table of Contents

Announcements and Demos

- We seem to have caused some confusion with our coupon code policy. Apologies! What we meant was to incentivize you to start your problem sets early. To explain it more clearly:
  - by default, psets are due on Thu at 12pm
  - if you start early, finishing part of pset by Wed at 12pm (and receive a coupon code), you can extend your deadline for rest of pset to Fri at 12pm
  - coupon-code problem still required even if not completed by Wed at 12pm

From Last Time

- We talked a little more high level about searching and sorting. Bubble sort, for example, gets its name from the way that large numbers bubble up toward the end of the list. We visualized bubble sort using this website.

- We quantified the efficiency of searching and sorting algorithms using big O notation. Bubble sort was said to take n^2 steps in the worst case, where n was the size of the input.
- Selection sort gets its name from selecting the smallest number on each walkthrough of the list and placing it at the front. Unfortunately, selection sort also took n^2steps in the worst case.
- Insertion sort involved sorting the list in place, but required shifting elements of the "sorted list" to the right whenever a smaller number needed to be placed in the middle of it. It too took n^2 steps.
- What's the best sorting algorithm? Why not ask President Obama?
- When talking about the running time of algorithms, O represents the upper bound, the worst case, whereas `\Omega` represents the lower bound, the best case. For bubble sort, selection sort, and insertion sort, the worst case was that the list was in exactly reverse order and the best case was that the list was already sorted. Whereas bubble sort (with a variable tracking the number of swaps) and insertion sort only took n steps in the best case, selection sort still took n^2 steps.
- So we say that bubble sort, selection sort, and insertion sort are all $O(n^2)$. Linear search is $O(n)$. Binary search is $O(\log n)$. Finding the length of an array is in $O(1)$, a.k.a. constant time, if that length is stored in a variable. `printf` is also $O(n)$ since it takes n steps to print n characters.
- Bubble sort and insertion sort are `\Omega`(n). Linear search and binary search are `\Omega`(1) because the element we're looking for might just be the first one we examine.
- One algorithm we mentioned briefly was *merge sort*. Merge sort is both `\Omega`($n \log n$) and $O(n \log n)$, so we say it is `\Theta`($n \log n$).

Merge Sort

- To see how merge sort compares to the other algorithms we've looked at so far, check out this animation. Notice that bubble sort, insertion sort, and selection sort are the three worst performers! The flip side is that they are relatively easy to implement.
- We can describe merge sort with the following pseudocode:

```
On input of n elements:
    If n < 2
        Return.
    Else:
        Sort left half of elements.
        Sort right half of elements.
        Merge sorted halves.
```

- If *n* is less than 2, then it's either 0 or 1 and the list is already sorted. This is the trivial case.
- If *n* is greater than or equal to 2, then what? We seem to be copping out with a circular algorithm. Two of the steps begin with the command "sort" without giving any indication as to how we go about that. When we say "sort," what we actually mean is reapply this whole algorithm to the left half and the right half of the original list.
- Will this algorithm loop infinitely? No, because after you've halved the original list enough times, you will eventually have less than 2 items left.
- Okay, so we're halving and halving and halving until we have less than 2 items and then we're returning. So far, nothing seems sorted. The magic must be in the "Merge sorted halves" step.
- One consideration with merge sort is that we need a second list for intermediate storage. In computer science, there's generally a tradeoff between resources and speed. If we want to do something faster, we may need to use more memory.
- To visualize merge sort, let's bring 8 volunteers on stage. We'll hand them numbers and sit them down in chairs so that they're in the following order:

  4 2 6 1 3 7 5 8
- The bold numbers are the ones we're currently focusing on. Merge sort says to first sort the left half, so let's consider:

  4 2 6 1 3 7 5 8
- Now we again sort the left half:

  4 2 6 1 3 7 5 8
- And again:

  4 2 6 1 3 7 5 8
- Now we have a list of size 1, so it's already sorted and we return. Backtracking, we look at the right half of the final two-person list:

  4 **2** 6 1 3 7 5 8
- Again, a list of size 1, so we return. Finally, we arrive at a merge step. Since the elements are out of order, we need to put them in the correct order as we merge:

  _ _ 6 1 3 7 5 8
  2 4 _ _ _ _ _ _

- From now on, the red numbers will represent the second list we use for intermediate storage. Now we focus on the right half of the left half of the original list:

  ```
  _ _ 6 1 3 7 5 8
  2 4 _ _ _ _ _ _
  ```
- We insert these two numbers in order into our intermediate list:

  ```
  _ _ _ _ 3 7 5 8
  2 4 1 6 _ _ _ _
  ```
- Now we merge the left and right half of the intermediate list:

  ```
  _ _ _ _ 3 7 5 8
  1 2 4 6 _ _ _ _
  ```
- Finally, we can insert the intermediate list back into the original list:

  ```
  1 2 4 6 3 7 5 8
  ```
- And we're done with the "Sort left half" step for the original list!
- Repeat for the right half of the original list, skipping to the "Sort left half" step:

  ```
  1 2 4 6 _ _ 5 8
  _ _ _ _ 3 7 _ _
  ```
- Sort right half:

  ```
  1 2 4 6 _ _ _ _
  _ _ _ _ 3 7 5 8
  ```
- Merge:

  ```
  1 2 4 6 _ _ _ _
  _ _ _ _ 3 5 7 8
  ```
- Move the right half back to the original list:

  ```
  1 2 4 6 3 5 7 8
  ```
- Now, merge the left half and the right half of the original list:

  ```
  _ _ _ _ _ _ _ _
  1 2 3 4 5 6 7 8
  ```
- And ta-da!

  ```
  1 2 3 4 5 6 7 8
  ```
- Merge sort is $O(n \log n)$. As before, the $\log n$ comes from the dividing by two. The n thus must come from the merging. You can rationalize this by considering the last merge step. To figure out which number to place in the intermediate array next, we point our left hand at the leftmost number of the left half and our right hand at the leftmost number of the

right half. Then we walk each hand to the right and compare numbers. All told, we walk through every number in the list, which takes *n* steps.

- Check out Rob's visualization of merge sort. You can even hear what sorting algorithms sound like.
- A function that calls itself is using *recursion*. In the above pseudocode, we implemented merge sort using recursion.

A Little Math

- To show mathematically that merge sort is $O(n \log n)$, let's use the following notation:

```
T(n) = 0, if n < 2
```

- So far, all this says is that it takes 0 steps to sort a list of 1 or 0 elements. This is the so-called *base case*.

```
T(n) = T(n/2) + T(n/2) + n, if n > 1
```

- This notation indicates that the rest of the algorithm, the *recursive case*, i.e. sorting a list of *n* elements, takes as many steps as sorting its two halves, each of *n* / 2 elements, plus an extra *n* steps to do the merging.
- Consider the case where *n* = 16:

```
T(16) = 2 * T(8) + 16
T(8)  = 2 * T(4) + 8
T(4)  = 2 * T(2) + 4
T(2)  = 2 * T(1) + 2
T(1)  = 0
```

- Since the *base case*, where a list of 0 or 1 is already sorted, takes 0 steps, we can now substitute 0 in for `T(1)` and calculate `T(2)`:

```
T(2) = 2 * 0 + 2
     = 2
```

- Now we can substitute 2 in for `T(2)` and so on until we get:

```
T(16) = 2 * 24 + 16
T(8)  = 2 * 8  + 8
T(4)  = 2 * 2  + 4
T(2)  = 2 * 0  + 2
T(1)  = 1
```

- Thus, `T(16)` is 64. This number is actually $n \log n$. Dividing the list successively accounts for $\log n$, but the additional *n* factor comes from the merge step.
- Here again with merge sort we've returned to the idea of "divide and conquer" that we saw in Week 0 with the phonebook example.
- In case you want to know what recursion is, try Googling it and checking out the "Did you mean" suggestion. Hooray for geek humor!

More with Recursion

`sigma-0.c`

- Let's write a program that sums up the numbers 0 through n, where n is provided by the user. We start with some boilerplate code to get a positive integer from the user using a do-while loop:

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

int main(void)
{
    int n;
    do
    {
        printf("Positive integer please: ");
        n = GetInt();
    }
    while (n < 1);
}
```

- Recall the sigma symbol (`\Sigma`) which stands for sum. It makes sense, then, to call our summing function `sigma`:

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

int main(void)
{
    int n;
    do
    {
        printf("Positive integer please: ");
        n = GetInt();
    }
    while (n < 1);

    int answer = sigma(n);

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

- Now we need to define `sigma`:

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

int sigma(int m);

int main(void)
{
    int n;
    do
    {
        printf("Positive integer please: ");
        n = GetInt();
    }
    while (n < 1);

    int answer = sigma(n);

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

int sigma(int m)
{
    if (m < 1)
    {
        return 0;
    }

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

- This is pretty straightforward. First, we do some error checking, then we iterate through all numbers 1 through `m`, summing them up as we go. `sum += i` is functionally equivalent to `sum = sum + i`.
- Don't forget that we need to declare `sigma` before `main` if we want to implement `sigma` after `main`!
- When we compile and run this, it seems to work! What happens when we mess with it by inputting a very large number? Turns out that if our sum becomes so large that an `int` doesn't have enough bits for it, it will be confused for a negative number.

sigma-1.c

- Let's try to approach the same problem using recursion.
- Our implementation of `main` doesn't change. `sigma`, however, now looks like this:

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

- You might worry that this implementation will induce an infinite loop. However, the first if condition represents a base case in which `sigma` doesn't call itself.

Teaser

`noswap.c`

- Consider the following code that claims to swap the values of two integers:

```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);
}

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

- Seems reasonable, right? Unfortunately, when we compile and run this, we get this output:

```
x is 1
y is 2
Swapping...
Swapped!
x is 1
y is 2
```

- Obviously, the numbers haven't really been swapped. We'll find out why next time!

Last updated 2013-10-03 00:26:15 PDT

Week 4, continued

Andrew Sellergren

Table of Contents

Announcements and Demos

- Fifth Monday is on 10/7! This is the deadline for changing your grading status in the course. Switching to SAT/UNS or Pass/Fail requires a signature, so please do approach David, Rob, or Lauren if you need.

Pointers

noswap.c

- Pointers are one of the more complex topics we cover, so don't feel bad if your mind feels stretched in the next few weeks. That's a good thing!
- Recall last time we ended with a function that didn't live up to its name:

```
#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);
}

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

- Though we expected to see `x` and `y` have the values 2 and 1, respectively, we actually saw that they still had their original values 1 and 2.
- To see why this doesn't work, let's bring a volunteer onstage. We'll ask her to pour orange juice and milk into two separate glasses representing two different `int`. If we ask her to swap the orange juice and milk, she wisely chooses to use another glass. This glass represents some temporary storage which we call `tmp` in the `swap`function above. Interestingly, if we implement the same code directly in `main`, the swapping actually works:

```
#include <stdio.h>

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

    printf("x is %i\n", x);
    printf("y is %i\n", y);
    printf("Swapping...\n");

    int tmp = x;
    x = y;
    y = tmp;

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

- So why does this logic work in `main` but not in `swap`? `a` and `b` are actually copies of `x` and `y`, so when we swap `a` and `b`, `x` and `y` are unchanged.

- One way to fix this would be to make `x` and `y` global variables, declaring them outside of `main`. In `fifteen.c`, it made sense to make certain variables global because they were to be used by the whole program. However, in a small program like `noswap.c`, using global variables is sloppy design.

swap.c

- How can we change the definition of `swap` to work as intended? Turns out we just need to add asterisks:

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

- What is an `int*`? It's the memory address of an `int`. More properly speaking, it is a *pointer* to an `int`. If your computer has 2 gigabytes of RAM, then there are 2 billion bytes, each of which has a memory address. Let's say the `int` that `a` points to is stored at the 123rd byte of RAM. The value of `a` then, is 123. To get at the actual integer value that's stored at byte 123, we write `*a`. `*a = *b` says "store at location `a` whatever is at location `b`."
- Now that we've changed `swap`, we need to change how we call `swap`. Instead of passing `x` and `y`, we want to pass the address of `x` and the address of `y`:

```
swap(&x, &y)
```

- `&` is the "address-of" operator and `*` is the dereference operator.
- Let's assume our integers 1 and 2 are stored next to each other in memory and 1 is stored at byte 123. That means 2 is stored 4 bytes away (since an `int` requires 4 bytes), so we'll assume that it's stored at byte 127. The values of `a` and `b`, then, are 123 and 127. We can simulate passing those to `swap` by writing them on pieces of paper and putting them in a black box.
- We ask a volunteer to come onstage and retrieve the pieces of paper from the black box. Next he needs to allocate a little bit of memory for variable `tmp`. In `tmp`, he stores the value of the `int` whose address is in `a`. This is 1.
- Next, at address 123, he erases the number 1 and writes in the number 2. This corresponds to the `*a = *b` line, which says "store at location `a` whatever is at location`b`."
- Finally, at address 127, he erases the number 2 and writes in the number 2, which was stored in `tmp`. `tmp` is a local variable, but goes away when `swap` returns.

compare-0.c

- For the first few weeks, we have worked with `string` as a data type. However, this is a type that we defined for you in the CS50 Library. A `string` is really a`char*`. It's the address of a `char`. In fact, it's the address of the first `char` in the string.
- Consider the following program which claims to compare two strings:

```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");
    }
}
```

- Here, we simply ask the user for two strings and store them in `s` and `t`. Then we ask if `s == t`. Seems reasonable, no? We've used the `==` operator for all the other data types we've seen thus far.
- But if we compile and run this program, typing "hello" twice, we always get "You typed different things!"
- Recall that a string is just an array of characters, so "hello" looks like this in memory:

```
h e l l o \0
```

- Although we're able to access the first character "h" using bracket notation, under the hood it's really located at one of 2 billion or so memory addresses. Let's call it address 123 again. Then "e" is at address 124, "l" is at address 125, and so on. A `char` only takes 1 byte, so this time the memory addresses are only 1 apart.

- If `GetString` is getting us this string, then what does it actually return? The number 123! Before it does so, it allocates the memory necessary to store "hello" and inserts those characters along with the null terminator.
- But if we only know the memory address of the first character, how do we know how long the string is? Recall that strings end with the special `\0` character, so we can just iterate until we find it.
- `compare-0.c` is buggy because it's comparing the memory addresses of the two strings, not the strings themselves. Maybe `s` is stored at memory address 123 and `t`is stored at memory address 200. Since 123 does not equal 200, our program says they're different strings.

`copy-0.c`

- Let's take a look at a program that tries, but fails to copy a string:

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

int main(void)
{
    // get line of text
    printf("Say something: ");
    string s = GetString();
    if (s == NULL)
    {
        return 1;
    }

    // try (and fail) to copy string
    string t = s;

    // change "copy"
    printf("Capitalizing copy...\n");
    if (strlen(t) > 0)
    {
        t[0] = toupper(t[0]);
    }

    // print original and "copy"
    printf("Original: %s\n", s);
    printf("Copy:     %s\n", t);
}
```

- We check that `s` isn't `NULL` in case the user has given us more characters than we have memory for. `NULL` is actually the memory address 0. By convention, no user data can ever be stored at byte 0, so if a program tries to access this memory address, it will crash.

- Now that we have the user-provided string in `s`, we assign the value of `s` to `t`. But if `s` is just a memory address, say 123, then `t` is now the same memory address. Both `s` and `t` are pointing to the same chunks of memory.
- To prove that this program is buggy, we'll try to capitalize `t`, but not `s`. The output, though, shows that both `s` and `t` are capitalized.
- To emphasize that their role is to *point* to other variables, pointers are often represented as arrows.

compare-1.c

- Finally, a program that truly compares two strings:

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

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

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

    // try to compare strings
    if (s != NULL && t != NULL)
    {
        if (strcmp(s, t) == 0)
        {
            printf("You typed the same thing!\n");
        }
        else
        {
            printf("You typed different things!\n");
        }
    }
}
```

- Now that we know a `string` is really just a `char*`, we need to be careful it's not `NULL`.
- `strcmp` is short for "string compare." It's a function that comes in `string.h`, which, according to the man page, returns 0 if two strings are identical, a negative number if the first string argument comes before the second string alphabetically, or a positive number if the first string argument comes after the second string alphabetically.

copy-1.c

- Copying a string is a little more complicated than just using the assignment operator:

```
 1 #include <cs50.h>
 2 #include <ctype.h>
 3 #include <stdio.h>
 4 #include <string.h>
 5 
 6 int main(void)
 7 {
 8     // get line of text
 9     printf("Say something: ");
10     char* s = GetString();
11     if (s == NULL)
12     {
13         return 1;
14     }
15 
16     // allocate enough space for copy
17     char* t = malloc((strlen(s) + 1) * sizeof(char));
18     if (t == NULL)
19     {
20         return 1;
21     }
22 
23     // copy string, including '\0' at end
24     int n = strlen(s);
25     for (int i = 0; i <= n; i++)
26     {
27         t[i] = s[i];
28     }
29 
30     // change copy
31     printf("Capitalizing copy...\n");
32     if (strlen(t) > 0)
33     {
34         t[0] = toupper(t[0]);
35     }
36 
37     // print original and copy
38     printf("Original: %s\n", s);
39     printf("Copy:     %s\n", t);
40 
41     // success
42     return 0;
43 }
```

- In line 17, we're declaring a pointer `t` and initializing it with the return value of a function named `malloc`. `malloc` takes a single argument, the number of bytes of memory requested, and returns the address in memory of the first of those bytes or `NULL` if the memory couldn't be allocated.

- In this case, we're allocating enough memory for all the characters in `s` plus 1 extra for the null terminator. We multiply this number of characters by `sizeof(char)`, which gives the size in bytes of a `char` on this particular operating system. Normally it will be 1, but we're handling other cases correctly, too.
- Once we have enough memory, we iterate through all of the characters in `s` and assign them one at a time to `t`.

Teaser

- Let's analyze some seemingly innocuous lines of code:

```
 1 int main(void)
 2 {
 3     int* x;
 4     int* y;
 5
 6     x = malloc(sizeof(int));
 7
 8     *x = 42;
 9
10     *y = 13;
11
12     y = x;
13
14     *y = 13;
15 }
```

- First, we declare two pointers `x` and `y`. We allocate enough memory to store an `int` and assign its address to `x`. We store the value 42 in this memory. Then we store in the memory address `y` the value 13. But wait, we didn't allocate memory for a second `int`, so what does `y` point to? Who knows! That's the problem. Line 10 is pretty painful for Binky.

Last updated 2013-10-25 12:28:03 PDT

Week 5

Andrew Sellergren

Table of Contents

Announcements and Demos

- On your seats you'll find a real-life plastic version of the Game of Fifteen! Consider it your reward for implementing it in C.
- Problem Set 4 involves writing a GUI for the game of Breakout. In the staff solution, you'll see that God mode allows you to play with no hands. This mode is actually trivial to implement, as you simply need to set the paddle's horizontal position to be the same as the ball's.
- If you haven't already, check out the Shorts!
- So much is possible with computer graphics! Check out what The Great Gatsby looks like before and after visual effects.

From Last Time

- GDB is a debugger that allows you to walk through your programs step by step, printing variables and examining the stack as you go. Although the interface is somewhat arcane, it will save you hours of time in the long run. Note that you can use this for Problem Set 4, as the GUI part of Breakout will pop up in a GWindow separate from your terminal.
- A `string` is really just a `char*`. This pointer stores the address of the first character of the string. We only need to know the address of the first character because we can iterate through the rest of the characters until we hit the null terminator. Memory addresses are often prefixed with 0x, which indicates that the number is in*hexadecimal*, or base 16.

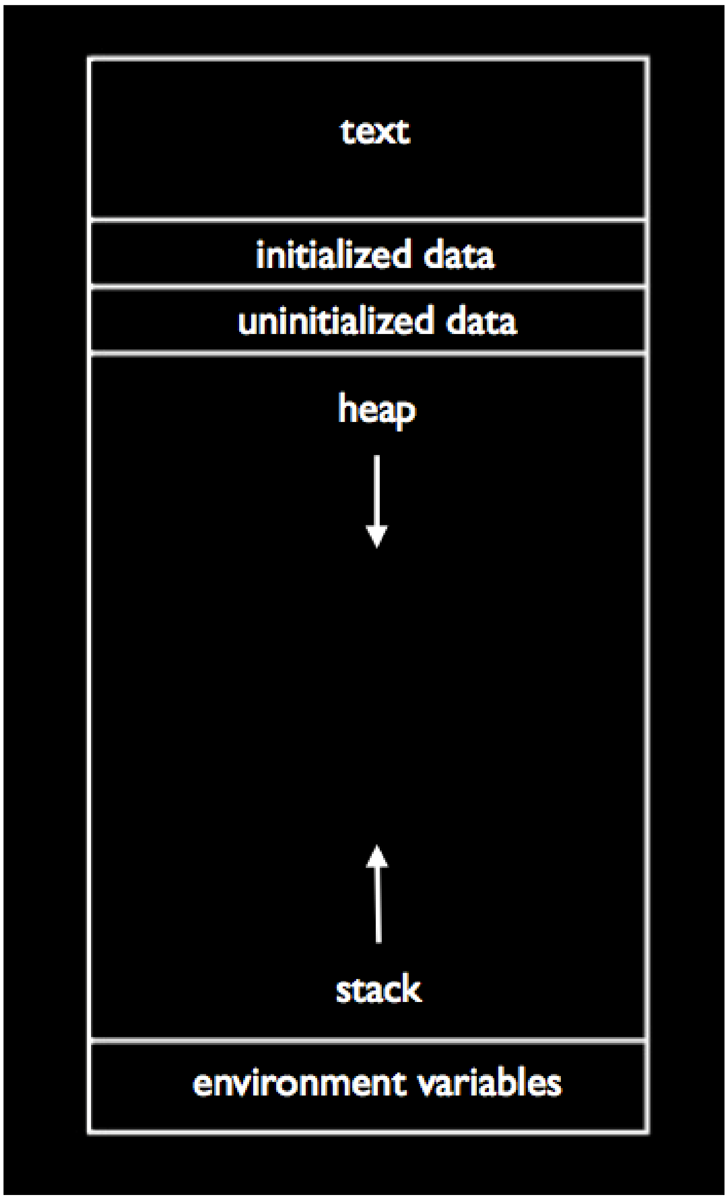
Hexadecimal makes use of 16 possible digits, 0-9 as well as A-F, to represent very large numbers quite concisely.

- With `compare-0.c`, we discovered that we can't compare strings using `==`. This operator only compares their memory addresses. For example, a string `s` might have the value 0x123 and a string `t` might have the value 0x456 even though they both point to strings that read "hello." In that case, "hello" is actually stored twice in memory. We can represent pointers as arrows pointing to chunks of memory. To compare strings as a human would, character for character, we can use a function like`strcmp`.
- *Segmentation faults* are memory errors that can result from iterating past the bounds of an array or dereferencing pointers that haven't been initialized.
- `copy-0.c` demonstrated that we can't copy a string simply by assigning it to a new variable. In effect, this only creates two pointers that store the same memory address and thus point to the same string. Changes to that string, then, will be visible from both pointers. We fixed this in `copy-1.c` by actually allocating memory using `malloc` and copying the string character by character into that new memory.
- Note that when you allocate memory, you can't trust what it contains until you've initalized it. Before it is initialized, newly allocated memory contains garbage values that we'll often denote with question marks.
- Question: why aren't we using the dereferencing operator to access the characters of a string? The square bracket notation is just syntactic sugar. When we write `*s`, it's equivalent to `s[0]`. However, if we wanted to access character `i` of `s`, we'd have to write `*(s+i)`, which is less intuitive but equivalent to `s[i]`. `s` is just a number representing the memory address of the first character, so adding `i` to it gives us the address of character `i`.
- Our first attempt at implementing a `swap` function failed because, by default, arguments to functions are passed as copies. When we defined `swap` so that it took two pointers as arguments, it worked as intended.

Memory

The Stack

- To help visualize your program's memory, consider the following diagram:



- The text segment contains the actual 0's and 1's of your program. The initialized data and unitialized data segments contain global variables. The *stack* is used to store local variables and function parameters.
- Each time a function is called in a program, a new *frame*, or section of memory, is added to the stack. This is where the stack gets its name, as it's just like a stack of trays in the dining hall.
- In `noswap.c`, `main` would have the bottommost frame and `swap` would have a frame on top of it. Inside `swap`'s frame, there is space for the variables `a`, `b`, and `tmp`. As we go through the logic of `swap`, the values of `a` and `b` are indeed switched. However, as soon as `swap` returns, its

entire frame is wiped away from the stack, so all our work was for nothing! `x` and `y`, which live in `main`'s frame, are unchanged.
- In `swap.c`, the version that actually worked, `a` and `b` still exist on `swap`'s frame, but now they point to the values of `x` and `y`. When we dereference them, we are actually changing memory on `main`'s frame.

Binky

- We left off last time with some code that put Binky in a tough spot:

```
 1 int main(void)
 2 {
 3     int* x;
 4     int* y;
 5
 6     x = malloc(sizeof(int));
 7
 8     *x = 42;
 9
10     *y = 13;
11
12     y = x;
13
14     *y = 13;
15 }
```

- What gets stored in `x` at line 6? The address of the first byte of memory allocated by `malloc`.
- Unfortunately, in line 10, we dereference the pointer `y` before it has been initialized. `y` contains some garbage value that we interpret as a memory address, so when we try to access it, bad things happen. In the video, this meant decapitation for Binky. In C programs, this usually means a segmentation fault.

Stack Overflow

- You may know this as a popular website, but it actually has a specific technical meaning. If a programmer forgets to check the boundaries of an array, he or she leaves his program vulnerable to an attack that can take over control of the program. Consider the following code:

```
#include <string.h>

void foo(char* bar)
{
    char c[12];
    memcpy(c, bar, strlen(bar));
}

int main(int argc, char* argv[])
```

```
{
    foo(argv[1]);
}
```

- For a thorough discussion of this attack, check out the Wikipedia article.
- In short, this program passes the first command-line argument to a function `foo` that writes it into an array of size 12. If the first command-line argument is less than 12 characters long, everything works fine. If the first command-line argument is greater than 12 characters long, then it will overwrite memory past the bounds of `c`. If the first command-line argument is greater than 12 characters long and actually contains the address in memory of some malicious code, then it could potentially overwrite the return address of `foo`. When `foo` returns, then, it will give control of the program over to this malicious code rather than `main`.
- Instead of ending on a scary note, let's end with a joke.

Last updated 2013-10-10 00:17:08 PDT

Week 5, continued

Andrew Sellergren

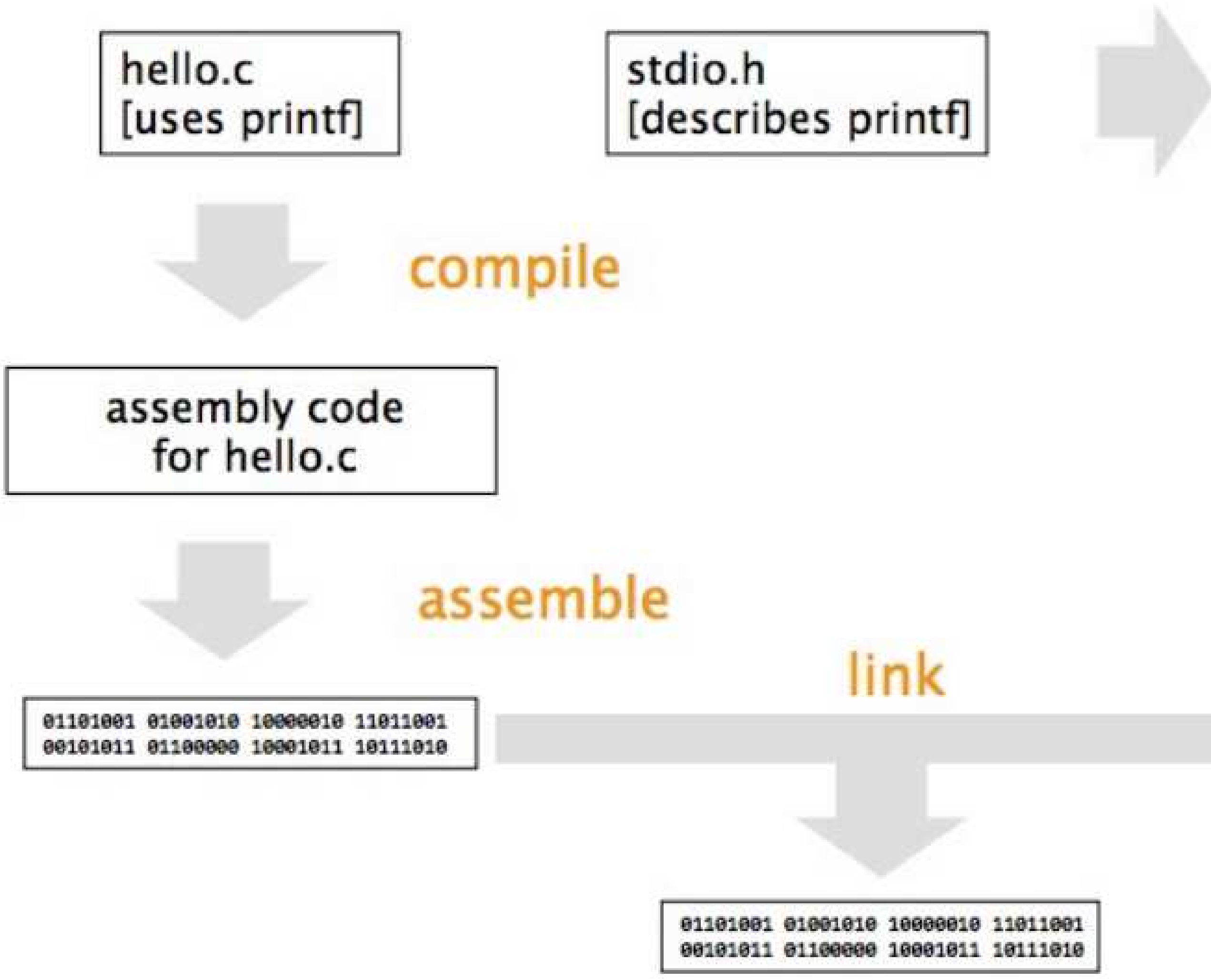
Table of Contents

Announcements and Demos

- No lecture on Monday 10/14 or Friday 10/18 (even though it says so in the syllabus)! There will be a quiz review session on Monday, though, which we'll announce by e-mail and on the course website. Sections will meet on Monday, but you can attend another section or watch the video online if you want.
- Quiz 0 is on Wednesday 10/16.
- After Quiz 0, we'll dive into the world of forensics. David and a few of the teaching fellows will walk around campus taking pictures, but unfortunately delete them from their memory card. It will be your job to recover them!
- In the coming weeks, we'll also talk more about graphics and images. We'll learn that "link:zoom and enhance" is not really as useful as TV shows and movies would suggest.

Compiling

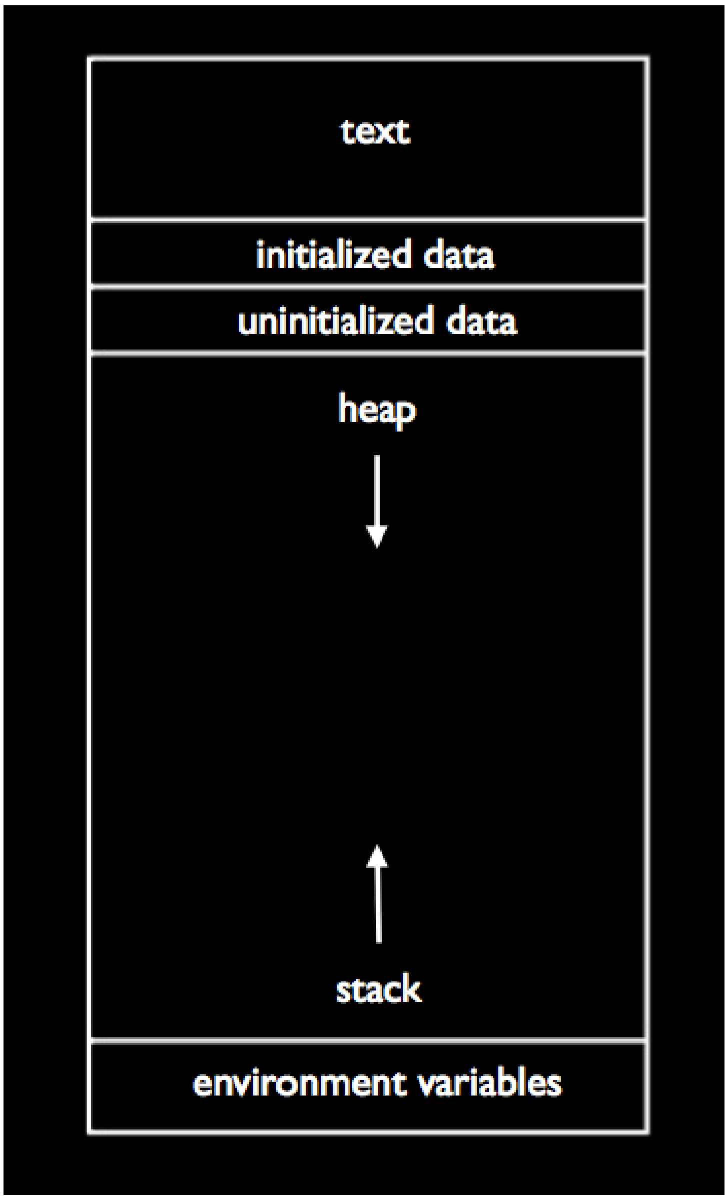
- What we call "compiling" a program actually consists of four steps:
    - pre-processing
    - compiling
    - assembling
    - linking
- Lines of code that begin with `#`, such as `#define` and `#include` are *pre-processor directives*. When you write `#include <stdio.h>`, it instructs the compiler to fetch the contents of `stdio.h` and paste them into your program before it begins translating into 0s and 1s. This occurs during the pre-processing step.
- Compiling actually involves translating C into assembly language. Assembling translates assembly language to binary. Finally, linking combines the 0s and 1s of your program with 0s and 1s of other people's code.
- To see what's going on during the compiling step, let's run `clang -S` on our `hello.c` program. This creates a file named `hello.s` written in assembly language. If you open this up, you'll see instructions like `pushl` and `movl` that manipulate *registers*, very small memory containers. These instructions vary between CPUs.
- The following diagram shows how these four steps of compiling connect with each other:



Memory

The Stack

- Recall from last time our picture of a program's memory:



- At the top, the text segment contains the actual 0s and 1s of the program. Below that are the initialized data and uninitialized data segments that contain global variables. We'll talk more about the heap later.
- The stack is the segment of memory on which frames are layered for each function call, including `main`. In `swap.c`, we saw that we could manipulate the frame of`main` while within `swap` if we passed it pointers to variables in the scope of `main`.
- We also learned that if we don't check the bounds of our arrays, we leave our programs susceptible to stack overflows, or buffer overrun attacks.

This is an exploit by which an adversary passes input to a program that overwrites memory it shouldn't have access to.

The Heap

- Let's look underneath the hood of a simple program:

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

int main(void)
{
    printf("State your name: ");
    string name = GetString();
    printf("hello, %s\n", name);
}
```

- First, we know that a `string` is really just a `char *`:

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

int main(void)
{
    printf("State your name: ");
    char* name = GetString();
    printf("hello, %s\n", name);
}
```

- Where is the memory for the string coming from? `GetString` is a function, so it gets its own frame on the stack. However, if we stored the string there, it would disappear as soon as `GetString` returned. Instead, we will store it in the *heap*. This is where `malloc` allocates memory.
- You can tell from the diagram that this design isn't perfect. If the stack grows upward and the heap grows downward, there's a chance that they'll collide. We can see this with a simple program like the following:

```
#include <stdio.h>

void foo(void)
{
    foo();
}

int main(void)
{
    foo();
}
```

- Obviously, this is a poorly designed program since it has a function that simply calls itself. If we compile and run this, we get a segmentation

fault. Every time `foo` is called, a new stack frame is allocated. Eventually, the program runs out of memory with which to allocate these frames.
- Normally, recursive functions have a *base case* in which they stop calling themselves. Even if a base case is defined, though, a recursive function can still exhaust all available memory if it calls itself too many times before hitting the base case.

Valgrind

- One gotcha with allocating memory on the heap is that we need to explicitly free it up when we're not using it anymore. Thus far, we've not been doing this when we call `GetString`. We can see that this memory is not being freed by running a tool called Valgrind. Valgrind executes programs and assesses them for these so-called*memory leaks*. You may have witnessed the effects of memory leaks in your daily life if you've ever left many programs running on your computer for a long time and noticed that the computer seems to slow down.
- When we run `valgrind ./hello-2` and type "David," we get a lot of output, but one line in particular stands out:

```
HEAP SUMMARY:
    in use at exit: 6 bytes in 1 blocks
```

- Those 6 bytes are the ones that were allocated to store the string "David."
- If we pass the command-line flag `--leak-check=full` to Valgrind, we get a more detailed report of where the memory leaks in our code are.
- How do we fix this memory leak? We just need to add one line of code:

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

int main(void)
{
    printf("State your name: ");
    char* name = GetString();
    printf("hello, %s\n", name);
    free(name);
}
```

- Now when we run this through Valgrind, we see the following output:

```
All heap blocks were freed -- no leaks are possible
```

- Valgrind can also help identify programming errors related to overstepping the bounds of arrays:

```
#include <stdlib.h>

void f(void)
```

```
{
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;
}

int main(void)
{
    f();
}
```

- Note that although `x` points to a chunk of memory (probably of size 40 bytes) that can store 10 `int` values, the array is zero-indexed, so `x[10]` is actually overstepping the bounds of the array. When we run this program through Valgrind, we get a line of output like so:

```
Invalid write of size 4
```

- This line refers to the fact that we tried to write 4 bytes (an `int`) to a chunk of memory that doesn't really belong to our program.
- Valgrind also tells us that we're not freeing the 40 bytes of memory that `x` points to.

The CS50 Library

- The CS50 Library is a set of functions and types we provided you to make it easier to get user input. In it, we also defined a `string` to be a `char*` until we could reveal to you what a pointer is:

```
typedef char* string;
```

- Let's peek at the definition of `GetChar`:

```
/**
 * Reads a line of text from standard input and returns the equivalent
 * char; if text does not represent a char, user is prompted to retry.
 * Leading and trailing whitespace is ignored.  If line can't be read,
 * returns CHAR_MAX.
 */
char GetChar(void)
{
    // try to get a char from user
    while (true)
    {
        // get line of text, returning CHAR_MAX on failure
        string line = GetString();
        if (line == NULL)
        {
            return CHAR_MAX;
        }

        // return a char if only a char (possibly with
        // leading and/or trailing whitespace) was provided
        char c1, c2;
        if (sscanf(line, " %c %c", &c1, &c2) == 1)
        {
            free(line);
            return c1;
        }
        else
        {
            free(line);
            printf("Retry: ");
        }
    }
}
```

- Why do we return `CHAR_MAX` if we fail to get a line of text from the user? `GetChar` returns a `char` according to its definition, so we need one `char` value that signals failure. By convention, this *sentinel value* is 255, the maximum possible value of a `char`. That means that we can't distinguish the case of an error from the case of the user typing the `char` 255, but since 255 is not something you can type on your keyboard, that's okay.
- `sscanf` is a function used for scanning formatted strings. Here, we pass it the line of text we got from the user, a format string, and the address of two `char` variables.`sscanf` will try to interpret the line of text as two characters in a row and fill in `c1` and `c2` accordingly. What we're hoping is that only `c1` will actually be populated. If both `c1` and `c2` are populated, it means the user typed more than one character, which is not what we asked for. If only `c1` is populated, then `sscanf` will return 1 and we can return `c1`.

Structs

- Just like we used `typedef` to create the `string` type in the CS50 Library, you can use it to define your own types:

```
#include <cs50.h>

// structure representing a student
typedef struct
{
    int id;
    string name;
    string house;
}
student;
```

- Here we're defining a variable type named `student`. Inside of this type, which is actually a *struct*, there are three variables representing the ID, name, and house of the student. To access these variables within a `student`, we use dot notation:

```
 1 #include <cs50.h>
 2 #include <stdio.h>
 3 #include <string.h>
 4
 5 #include "structs.h"
 6
 7 // class size
 8 #define STUDENTS 3
 9
10 int main(void)
11 {
12     // declare class
13     student class[STUDENTS];
14
15     // populate class with user's input
16     for (int i = 0; i < STUDENTS; i++)
17     {
18         printf("Student's ID: ");
19         class[i].id = GetInt();
20
21         printf("Student's name: ");
22         class[i].name = GetString();
23
24         printf("Student's house: ");
25         class[i].house = GetString();
26         printf("\n");
27     }
28
29     // now print anyone in Mather
30     for (int i = 0; i < STUDENTS; i++)
31     {
32         if (strcmp(class[i].house, "Mather") == 0)
33         {
34             printf("%s is in Mather!\n\n", class[i].name);
35         }
36     }
37
38     // free memory
39     for (int i = 0; i < STUDENTS; i++)
40     {
41         free(class[i].name);
42         free(class[i].house);
43     }
44 }
```

- In line 13, we declare an array of `student`. We then loop through that array and populate it with data from the user.

Images

- There are many different file formats used to store images. One such format is a *bitmap*, or BMP. A very simple bitmap might use 0 to represent black and 1 to represent white, so a series of 0s and 1s could store a black-and-white image.

- More sophisticated file formats like JPEG store 0s and 1s for the image itself but also metadata. In Problem Set 5, you’ll use this fact to detect JPEGs that have been lost on David’s memory card.

Last updated 2013-10-12 23:33:02 PDT

Week 7

Andrew Sellergren

Table of Contents

Announcements and Demos

- Sign up for CS50 Lunch this Friday!
- Final Projects are nigh! The specification has already been released, detailing the following checkpoints:
    - Pre-Proposal
    - Proposal
    - Status Report
    - CS50 Hackathon
    - Implementation

    - CS50 Fair

From Last Time

- By now, you're hopefully getting comfortable with the concept of a pointer, a memory address.
- We learned that Valgrind is a useful tool for detecting memory leaks and abuses. A lot of its output is cryptic, but you should look for phrases like "invalid write" and "definitely lost" as hints to your mistakes.

User Input

- `sscanf` is what the CS50 Library uses to get input from the user in functions like `GetString`.

`scanf-0.c`

- Take a look at a simple example of using `scanf`, which is quite similar to `sscanf`:

```c
#include <stdio.h>

int main(void)
{
    int x;
    printf("Number please: ");
    scanf("%i", &x);
    printf("Thanks for the %i!\n", x);
}
```

- The first argument to `scanf` resembles an argument we might pass to `printf` (The "f" in both denotes "formatted"). The second argument is the address of `x`, thus empowering `scanf` to actually modify the memory in which `x` is stored.
- This program behaves as expected if the user provides a number as input. However, if the user provides a string or any other non-numeric input, the program behaves strangely. One of the things the CS50 Library provides is some error checking so that if the user provides bad input, he or she will be prompted to retry.

`scanf-1.c`

- `scanf-1.c` closely resembles `scanf-0.c`, but introduces one major bug:

```c
#include <stdio.h>

int main(void)
{
    char* buffer;
    printf("String please: ");
    scanf("%s", buffer);
    printf("Thanks for the %s!\n", buffer);
}
```

- A buffer is just a generic name for a chunk of memory, a place to store information.
- The problem here is that `buffer` is uninitialized. We didn't ask the operating system for a chunk of memory in which to store the string the user gives us. If we run this program, it will probably crash with a segmentation fault.

`scanf-2.c`

- One solution to the bug in `scanf-1.c` would be to allocate memory for `buffer` on the stack, as we do in `scanf-2.c`:

```
#include <stdio.h>

int main(void)
{
    char buffer[16];
    printf("String please: ");
    scanf("%s", buffer);
    printf("Thanks for the %s!\n", buffer);
}
```

- Here, you can see that `scanf` treats the array `buffer` as a memory address. We know that the address is for a chunk of memory of size 16 bytes.
- In what scenario might this program also be buggy? If the user provides a string longer than 15 characters (not 16 because we need at least one character for the null terminator), the program may crash with a segmentation fault.
- How do we know in advance how much memory to request for user input? We don't! The CS50 Library has some logic that reads user input one character at a time with `scanf` and requests more memory whenever it runs out.

Structs

- Let's revisit the problem of storing information about a number of students. We might start off just declaring a few variables like so:

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

int main(void)
{
    string name = GetString();
    string house = GetString();
}

```

- * What if we want to store another student's information? Well I guess we need some more variables:
- +
- [source]

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

int main(void) { string name = GetString(); string house = GetString();

```
string name2 = GetString();
string house2 = GetString();
    string name3 = GetString();
    string house3 = GetString();
}
```

- Hopefully, this strikes you as bad design. In prior weeks, we solved the problem of storing numerous variables of the same types by using arrays:

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

int main(void)
{
    string names[3];
    string house[3];
}
```

- This solves the problem of repetitive code, but introduces the problem of names no longer being directly associated with houses.

structs.h

- To reduce code repetition but keep information tightly coupled, we can introduce a new variable type using syntax like the following:

```
#include <cs50.h>

// structure representing a student
typedef struct
{
    string name;
    string house;
}
student;
```

structs-0.c

- Now we have a type called `student` that contains both pieces of information about a student. Filling in this information is quite straightforward:

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

#include "structs.h"
```

```

// number of students
#define STUDENTS 3

int main(void)
{
    // declare students
    student students[STUDENTS];

    // populate students with user's input
    for (int i = 0; i < STUDENTS; i++)
    {
        printf("Student's name: ");
        students[i].name = GetString();

        printf("Student's house: ");
        students[i].house = GetString();
    }

    // now print students
    for (int i = 0; i < STUDENTS; i++)
    {
        printf("%s is in %s.\n", students[i].name, students[i].house);
    }

    // free memory
    for (int i = 0; i < STUDENTS; i++)
    {
        free(students[i].name);
        free(students[i].house);
    }
}
```

- `students` is an array of variables of type `student` ad defined in the `structs.h` header file.
- We access element `i` of `students` by writing `students[i]`, as with any array. To access the pieces of information within any given `student`, we use dot notation:`students[i].name` and `students[i].house`.
- Don't forget that we need to free the memory that we allocated when we called `GetString`! Technically, we should also be checking if `students[i].name` and`students[i].house)` are not `NULL` before we free them.

structs-1.c

- Before we examine the code, let's just make and run `structs-1.c`. After we enter in some data and the program exits successfully, a file named `students.csv` is created. CSV stands for comma-separated values, a very simple version of a table like you may have worked with in Excel.
- The code that creates this CSV file looks like this:

```
 1 #include <cs50.h>
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <string.h>
 5 
 6 #include "structs.h"
 7 
 8 // number of students
 9 #define STUDENTS 3
10 
11 int main(void)
12 {
13     // declare students
14     student students[STUDENTS];
15 
16     // populate students with user's input
17     for (int i = 0; i < STUDENTS; i++)
18     {
19         printf("Student's name: ");
20         students[i].name = GetString();
21 
22         printf("Student's house: ");
23         students[i].house = GetString();
24     }
25 
26     // save students to disk
27     FILE* file = fopen("students.csv", "w");
28     if (file != NULL)
29     {
30         for (int i = 0; i < STUDENTS; i++)
31         {
32             fprintf(file, "%s,%s\n", students[i].name, students[i].house);
33         }
34         fclose(file);
35     }
36 
37     // free memory
38     for (int i = 0; i < STUDENTS; i++)
39     {
40         free(students[i].name);
41         free(students[i].house);
42     }
43 }
```

- Line 27 does the work of actually opening a file, specifying "w" as an argument to `fopen` to indicate that we want to write to this file (whereas "r" would indicate read mode). `fopen` returns a pointer to a `FILE` object. Instead of `printf`, we use `fprintf` to write to our file.
- Question: what happens when you try to free a `NULL` pointer? Your program will probably segfault.

Storage

Hard Drives

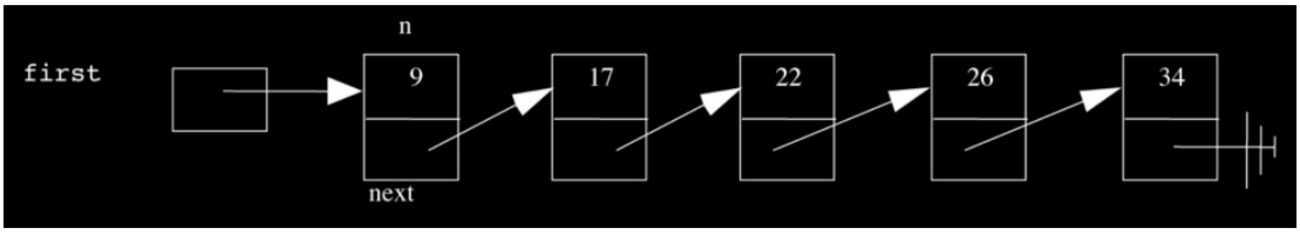
- Hard drives that aren't SSDs (solid-state drives with no moving parts) consist of circular metal platters and magnetic heads that read and write bits on them. The 0s and 1s of files are stored by magnetic particles that are flipped with either their north or their south poles sticking up.
- Somewhere on the hard drive there exists a table that maps filenames to their memory addresses. As you can with RAM, you can number all of the bytes of a hard drive so that each has a memory address. When you delete a file, say by dragging it to the trash can or even by emptying the trash can, the contents of the file may not actually be deleted. Rather, the file's entry in the location table is simply erased so that the operating system forgets where the file was stored. Not until the 0s and 1s of the file are actually overwritten will the file's contents truly be gone. In the meantime, the file can be recovered by software like Norton or by a program like the one you'll write for Problem Set 5. Having been provided with the raw bytes of an SD card, you'll be tasked with searching through them to look for the particular pattern of bits that identifies the start of a JPEG file.

Floppy Disks

- Back in David's day [1], another type of storage called floppy disks was popular. Functionally, these are very similar to hard drives in that inside their plastic casing, there is a circular magnetic platter. You can get your hands on it just by ripping off the metal tab. Be careful, there's a spring in there!
- These days, the size of hard drives is measured in terabytes. A so-called "high-density" floppy disk can only store 1.44 megabytes, or roughly 1 millionth of a terabyte.

Linked Lists

- Arrays are useful because they enable the storage of similar variables in contiguous memory. One downside of arrays is that they have a fixed size. Another downside is that there's no easy way to insert something in the middle of an array. To do so, we would have to allocate memory for a copy of the array and then shift all the elements to the right.
- To solve the problem of fixed size, we'll relax the constraint that the memory we use be contiguous. We can take a little bit of memory from here and a little bit of memory from there just so long as we can connect them together. This new data structure is called a *linked list*:



- Each element of a linked list contains not only the data we want to store, but also a pointer to the next element. The final element in the list has the `NULL` pointer.
- To implement a linked list, we'll borrow some of the syntax we used for structs:

```
typedef struct node
{
    int n;
    struct node *next;
}
node;
```

- Pictorially, `next` is the bottom box that points to the next element of the linked list. Why do we have to declare it as a `struct node*` then? The compiler doesn't yet know what a `node` is, so we have to call it a `struct node` in the meantime.
- There are a few linked list operations that will be of interest to us:
    - insert
    - delete
    - search
    - traverse
- The search operation is actually pretty easy to implement:

```
bool search(int n, node* list)
{
    node* ptr = list;
    while (ptr != NULL)
    {
        if (ptr->n == n)
        {
            return true;
        }
        ptr = ptr->next;
    }
    return false;
}
```

- `search` takes two arguments, the number to be searched for and a pointer to the first node in the linked list. We then declare a pointer `ptr` that we'll use to walk through the list. Since `ptr` is a pointer to a struct, we use the arrow syntax (`->`) to access the elements within the struct. To advance to the next node in the linked list, we assign `ptr->next` to `ptr`. More on this on Wednesday!

---

1. The turn of the 20th century?

Last updated 2013-10-23 21:15:57 PDT

Week 7, continued

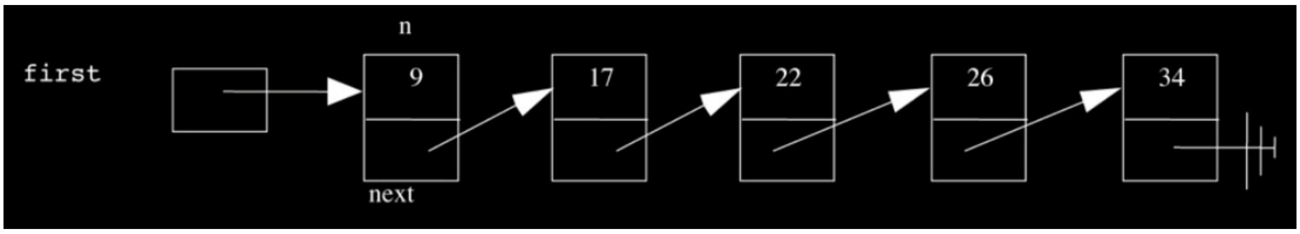
Andrew Sellergren

Table of Contents

Linked Lists

- Arrays are of fixed size, which is both an advantage and a disadvantage. It's an advantage because it means you know exactly how much space you'll be using, but it's a disadvantage because it means you have to allocate an entirely new array if you need more space. Arrays are stored as a single chunk of memory, which means that we have *random access* to all of their elements.

- We introduced the linked list as a data structure of expandable size:



- We called each of the elements of a linked list a *node*. Each node is implemented as a struct:

```
typedef struct node
{
    int n;
    struct node* next;
}
node;
```

- In the example above, the struct consists of an integer and a pointer to the next node. Using these pointers, we can traverse the entire linked list given only a pointer to the first node. The last node in the linked list has a NULL pointer. Adding a node to the linked list is as simple as allocating memory for it and rearranging pointers.

`list-0.c`

- Although this file isn't very complex, it's conventional to put type definitions in a separate header file. When you include that file, you use double quotes instead of angle brackets because the file is local:

```
#include "list-0.h"
```

- Our `main` function is simply a menu interface for getting a command from the user:

```
int main(void)
{
    int c;
    do
    {
        // print instructions
        printf("\nMENU\n\n"
            "1 - delete\n"
            "2 - insert\n"
            "3 - search \n"
            "4 - traverse\n"
            "0 - quit\n\n");

        // get command
        printf("Command: ");
        c = GetInt();

        // try to execute command
        switch (c)
        {
            case 1: delete(); break;
            case 2: insert(); break;
            case 3: search(); break;
            case 4: traverse(); break;
        }
    }
    while (c != 0);

    // free list before quitting
    node* ptr = first;
    while (ptr != NULL)
    {
        node* predptr = ptr;
        ptr = ptr->next;
        free(predptr);
    }
}
```

- We have a do-while loop to prompt the user for non-negative input. Once the user provides input, we switch on it to execute one of the four different functions. Finally, we free the linked list before exiting.

Search

- Let’s take a look at how the `search` function is implemented:

```
void search(void)
{
    // prompt user for number
    printf("Number to search for: ");
    int n = GetInt();

    // get list's first node
    node* ptr = first;

    // search for number
    while (ptr != NULL)
    {
        if (ptr->n == n)
        {
            printf("\nFound %i!\n", n);
            sleep(1);
            break;
        }
        ptr = ptr->next;
    }
}
```

- We declare a pointer to a node called `ptr` and point it to the first node in the list. To iterate through the list, we set our while condition to be `ptr != NULL`. The last node in the linked list points to NULL, so `ptr` will be `NULL` when we've reached the end of the list. To access the integer within the node that `ptr` points to, we use the `->` syntax. `ptr->n` is equivalent to `(*ptr).n`. If the integer within the node is the one we're searching for, we're done. If not, we update `ptr` to be the `next` pointer of the current node.

Insertion

- Insertion into a linked list requires handling three different cases: the beginning, middle, and end of the list. In each case, we need to be careful in how we update the node pointers lest we end up orphaning part of the list.

- To visualize insertion, we'll bring 6 volunteers onstage. 5 of these volunteers will represent the numbers 9, 17, 22, 26, and 34 that are in our linked list and 1 volunteer will represent the `first` pointer.
- Now, we'll request memory for a new node, bringing one more volunteer onstage. We'll give him the number 5, which means that he belongs at the beginning of the list. If we begin by pointing `first` at this new node, then we forget where the rest of the list is. Instead, we should begin by pointing the new node's `next` pointer at the first node of the list. Then we update `first` to point to the new node.
- Again, we'll request memory for a new node, bringing another volunteer onstage and assigning her the number 55. She belongs at the end of the list. To confirm this, we traverse the list by updating `ptr` to the value of `next` for each node. In each case, we see that 55 is greater than `ptr->n`, so we advance to the next node. However, ultimately, we end up with `ptr` equal to `NULL` because 55 is greater than all of the numbers in the list. We don't have a pointer, then, to the last node in the list, which means we can't update it. To prevent this, we need to keep track of the node one to the left of `ptr`. We'll store this in a variable called `predptr` in our sample code. When we reach the end of the list, `predptr` will point to the last node in the list and we can update its `next` value to point to our new node.
- Another solution to this problem of keeping track of the previous node is to implement a *doubly linked list*. In a doubly linked list, each node has

a `next` pointer to point to the next node and a `prev` pointer to point to the previous node.

- Once more, we'll request memory for a new node, assigning the value 20 to our last volunteer. This time when we traverse the list, our `predptr` is pointing to the 17 node and our `ptr` is pointing to the 22 node when we find that `ptr->n` is greater than 20. To insert 20 into the list, we point the `next` pointer of `predptr` to our new node and the `next` pointer of our our new node to `ptr`.
- Linked lists are yet another example that design is very much subjective. They are not unilaterally better than arrays, but they may be more useful than arrays in certain contexts. Likewise, arrays may be more useful than linked lists in certain contexts.

Hash Tables

- The holy grail of running time is $O(1)$, i.e. constant time. We've already seen that arrays afford us constant-time lookup, so let's return to this data structure and use it to store a list of names. Let's assume that our array is of size 26, so we can store a name in the location corresponding to its first letter. In doing so, we also achieve constant time for insertion since we can access location `i` in the array in 1 step. If we want to insert the name Alice, we index to location 0 and write it there.
- This data structure is called a *hash table*. The process of getting the storage location of an element is called *hashing* and the function that does so is called a *hash function*. In this case, the hash function simply takes the first letter of the name and converts it to a number.

Linear Probing

- What problems might arise with this hash table? If we want to insert the name Aaron, we find that location 0 is already filled. We could take the approach of inserting Aaron into the next empty location, but then our running time deteriorates to linear because in the worst case, we may have to iterate through all n locations in the array to insert or search for a name. This approach is appropriately named *linear probing*.

Separate Chaining

- When two elements have the same hash, there is said to be a *collision* in the hash table. Linear probing was our first approach to handling collisions. Another approach is *separate chanining*. In separate chaining, each location in the hash table stores a pointer to the first node of a linked list. When a new element needs to be stored at a location, it is simply added to the beginning of the linked list.

The Birthday Problem

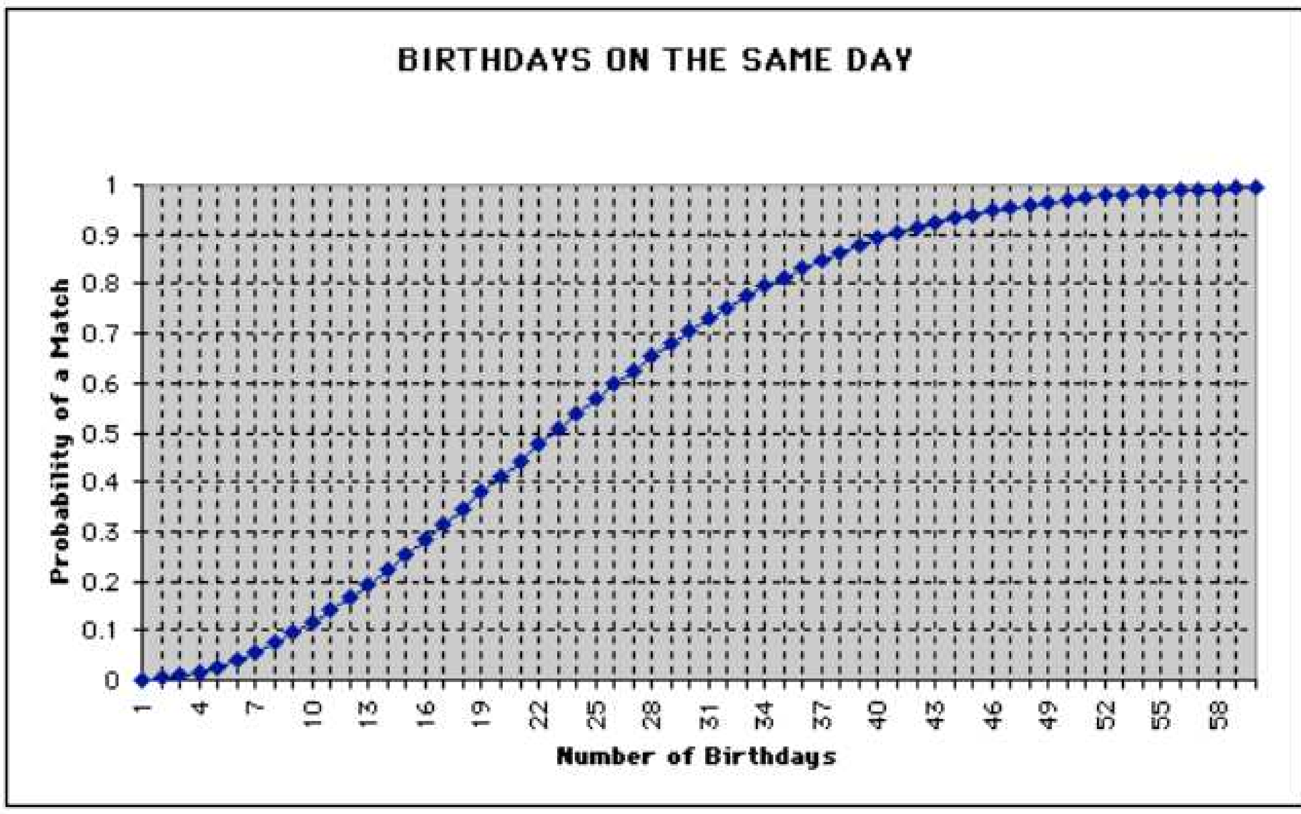
- Why worry at all about collisions? How likely is it really that they will happen? It turns out the probability of collisions is actually quite high. We can phrase this question in a slightly different way that we'll call the Birthday Problem:

  In a room of n CS50 students, what's the probability that at least 2 students have the same birthday?

- To answer this question, we'll consider the opposite: what's the probability that no 2 students have the same birthday. If there's only 1 student in the room, then the probability that no 2 students have the same birthday is 1. If there are 2 students in the room, then there are 364 possible birthdays out of 365 which the second student could have that would be different from the first student's. Thus, the probability that no 2 students have the same birthday in a room of 2 is 364 / 365. The probability that no 2 students have the same birthday in a room of 3 is 363 / 365. And so on. To get the total probability, we multiple all of these probabilities together. You can see this math here, courtesy of Wikipedia:

$$\begin{aligned}\bar{p}(n) &= 1 \times \left(1 - \frac{1}{365}\right) \times \left(1 - \frac{2}{365}\right) \times \cdots \times \left(1 - \frac{n-1}{365}\right) \\ &= \frac{365 \times 364 \times \cdots \times (365 - n + 1)}{365^n} \\ &= \frac{365!}{365^n (365 - n)!} = \frac{n! \cdot \binom{365}{n}}{365^n} = \frac{{}^{365}P_n}{365^n}\end{aligned}$$

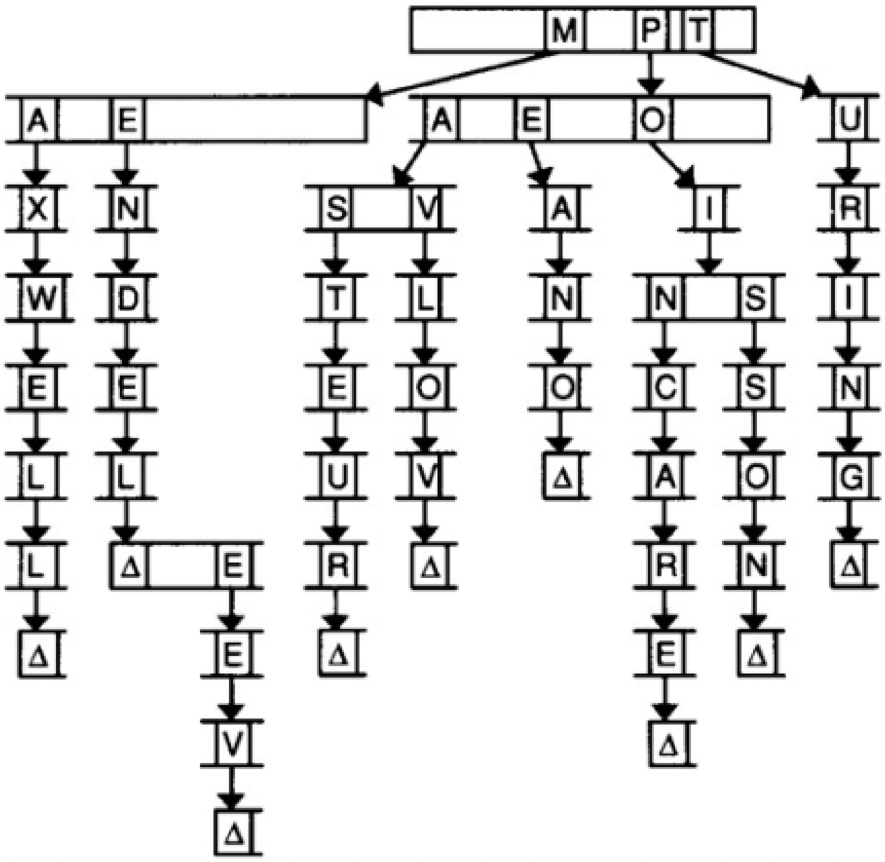
- This is much easier to interpret in the form of a graph, however:



- Notice that the probability is already 0.5 when there are only 22 students in the room. By the time we consider the case where there are 58 students in the room, the probability is almost 1. The implication for hash tables is that there are going to be collisions.
- What is the worst-case running time of search in a hash table that uses separate chaining? In the worst case, we're going to have to traverse the entire linked list at any hash table location. If we consider the number of locations in our hash table to be m, then the lookup time for a hash table that uses separate chaining is $O(n/m)$. m is a constant, though, so the lookup time is really just $O(n)$. In the real world, however, $O(n/m)$ can be much faster than $O(n)$.
- What is the worst-case running time of insertion in a hash table that uses separate chaining? It's actually $O(1)$ if we always insert to the beginning of the linked list at each hash table location.

Tries

- One last data structure we'll discuss is a *trie*. The word "trie" comes from the word "retrieval," but is usually pronounced like "try." For our purposes, the nodes in a trie are arrays. We might use a trie to store a dictionary of names of famous scientists, as this diagram suggests:



- In this trie, each index in the array stands for a letter of the alphabet. Each of those indices also points to another array of letters. The Δ symbol denotes the end of a name. We have to keep track of where words end so that if one word actually contains another word (e.g. Mendeleev and Mendel), we know that both words exist. In code, the Δ symbol could be a Boolean flag in each node:

```c
typedef struct node
{
    bool word;
    struct node* children[27];
}
node;
```

- One advantage of a trie is that insertion and search times are unaffected by the number of elements already stored. If there are n elements stored in the trie and you want to insert the value Alice, it still takes just 5 steps, one for each letter. This runtime we might express as $O(k)$, where k is the length of the longest possible word. But k is a constant, so we're actually just talking about $O(1)$, or constant-time insertion and lookup.

- Although it may seem like a trie is the holy grail of data structures, it may not perform better than a hash table in certain contexts. Choosing between a hash table and a trie is one of many design decisions you'll have to make for Problem Set 6.

Teaser

- Before long, we'll transition to talking about web development, including HTML, PHP, and JavaScript. As a brief teaser, enjoy this trailer to Warriors of the Net.

Last updated 2013-10-26 21:50:16 PDT

Week 8

Andrew Sellergren

Table of Contents

Announcements and Demos

- Problem Set 5 is now over, but the photo contest is not! Submit the link to your photos by noon on Monday 11/4 for a chance to win a Leap Motion!
- We offer a number of CS50 Seminars on a wide variety of topics to help you gear up for your Final Project. Register here and check out past seminars here. On the roster so far for this year are:
  - Amazing Web Apps with Ruby on Rails
  - Computational Linguistics
  - Introduction to iOS
  - JavaScript for Web Apps
  - Leap Motion SDK
  - meteor.js: JavaScript on the back end
  - Node.js

    - Sleek Android Design
    - Web Security: Active Defense
- Thanks to the folks at Leap Motion, we have ~30 devices available for those who would like to develop their Final Project with one!
- Short on Final Project ideas? Check out projects.cs50.net.

From Last Time

Linked Lists

- We introduced linked lists as a data structure with dynamic size. One tradeoff for this is that a linked list requires more memory than an array of equal size. The extra memory is necessary to store the pointers within each node. Another tradeoff is that search in a linked list runs in $O(n)$ even if it's sorted; in the worse case, we have to traverse the whole list because we don't have random access as we do in an array.
- Recall our implementation of a linked list node:

```
typedef struct node
{
    int n;
    struct node* next;
}
node;
```

- We must declare `next` as a `struct node*` because `node` is not yet fully defined.

Hash Tables

- We also discussed hash tables, which associate keys with values. The keys are determined by taking a deterministic hash of the values using a hash function. In our first example, this hash function simply took the first letter of the name that we wanted to store, 0 for Alice, 1 for Bob, and so on. In code, this might look like:

```
int hash(char* s)
{
    return s[0] - 'A';
}
```

- This function is a little buggy, of course, because we're not checking if `s` is `NULL` and we're not accounting for lowercase strings. But you get the idea.
- Hash tables inevitably have collisions in which two values have the same hash. Linear probing was the first approach we looked at for handling

collisions. More compelling, however, was the second approach in which our hash table consisted of pointers to linked lists. If a second value needed to be inserted at a key, we simply add it to the beginning of the linked list. By inserting at the beginning of the list, we maintain $O(1)$ insertion time.

Tries

- The final data structure we examined was a trie. Both insertion time and search time for a trie are $O(k)$, where k is the length of the word being inserted or searched for. But k is really a constant since words have a finite length, so insertion time and search time are actually $O(1)$.
- A trie is a tree structure consisting of arrays of arrays. Each index in each array stores a pointer to another array. Considering how many arrays we're actually storing, then, the tradeoff for a trie implementation is clearly the memory it requires. Yet again, we see that there is a tradeoff between memory and running time, between space and speed.

Stacks

- We've already seen that a program's memory is called the stack because of the way in which function frames are layered on top of each other. More generally, a stack is a data structure that has its own advantages and disadvantages compared to arrays, linked lists, hash tables, and tries.
- We interact with stacks using only two basic operations: push and pop. To add data to the stack, we push it onto the top of the stack. To retrieve data from the stack, we pop it off the top of the stack. As a result, a stack exhibits last in first out (LIFO) storage. The only data that we can retrieve from the stack is the last data we added to it.
- In what contexts might LIFO storage be useful? Clearly it's useful for organizing a program's memory. As we'll see soon, it's also useful for validating the tree structure of a web page's HTML.
- We might implement a stack like so:

```
typedef struct
{
    int trays[CAPACITY];
    int size;
}
stack;
```

- It's convenient to think of a stack like the stack of trays in the dining halls. In the code above, `CAPACITY` is a constant defining the maximum number of such trays that a stack can contain. Another integer named `size` stores the number of trays currently in the stack.

- Let's say `CAPACITY` is 3. Initially, `trays` contains nothing but garbage values and `size` is 0. Let's say we push the number 9 onto the stack. Now `size` becomes 1 and the 0th index of `trays` is set to 9. Next, we push 17 onto the stack, `size` becomes 2 and the 1st index of `trays` is set to 17. Finally, we push 22 onto the stack,`size` becomes 3 and the 2nd index of `trays` is set to 22. What happens when we try to push 27 onto the stack? We can't add it to the stack because we have filled all available indices in `trays`. If our push function returned a Boolean, we would want it to return false in this case.
- One way to implement a stack with dynamic size would be to declare `trays` as a pointer and `malloc` it at runtime. Another way would be to declare `trays` as a pointer to a linked list.

Queues

- If you're familiar with the lines that form outside the Apple store when a new iPhone is released, then you're familiar with queues. We also interact with queues using two basic operations: enqueue and dequeue. Whereas stacks exhibit LIFO storage, however, queues exhibit FIFO, i.e. first in first out, storage. Imagine how upset the people outside the Apple store would be if the line were implemented as a stack instead of a queue!
- We can implement a queue using a struct:

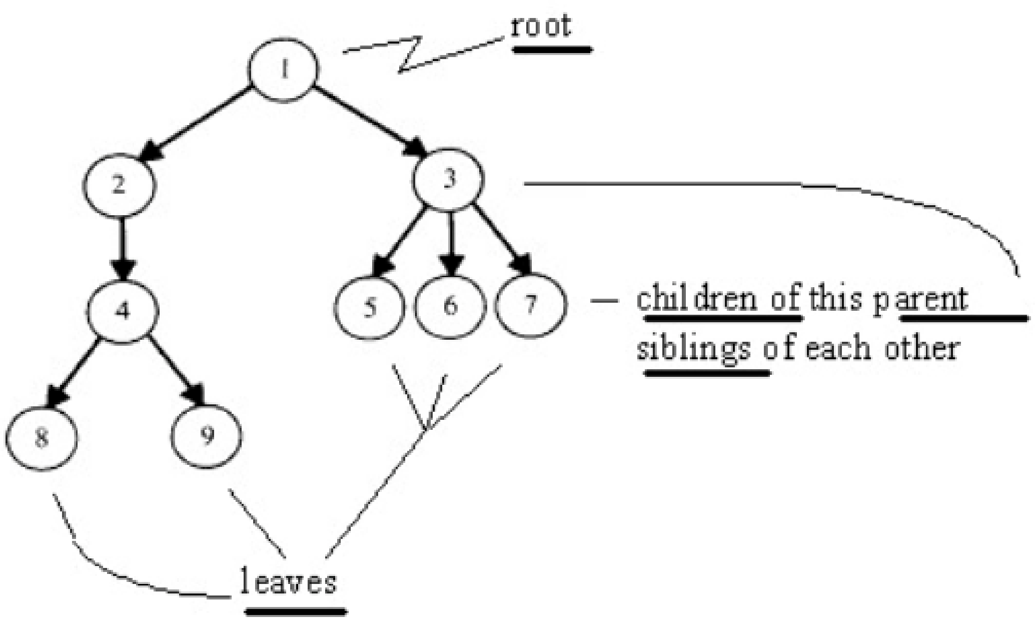
```
typedef struct
{
    int numbers[CAPACITY];
    int front;
    int size;
}
queue;
```

- Note that our `queue` type is very similar to our `stack` type. Why do we need the extra `int` for `queue`? `front` keeps track of the index of the next value to be dequeued. If we add 9, 17, and 22 as we did to the stack and then remove 9, we need to know that 17 should be the next value dequeued. `front` is incremented whenever a value is dequeued, at least until we reach `CAPACITY`.
- What happens when we have one value in the last index of `numbers` and we want to enqueue another number? We can use an operator called modulus (`%`) to wrap around to index 0 of `numbers`. Our one value is at

index 2, so if we insert at 3 modulo `CAPACITY`, we'll be inserting at index 0.
- Using this approach, insertion into a queue runs in constant time.

Trees

- A trie is actually a specific type of a data structure called a *tree*:



- A tree consists of 0 or more nodes. If there is at least one node, it is called the *root*. Each node can have 0 or more *children*. A node with 0 children is called a *leaf*.
- The diagram above shows a tree that has varying numbers of children for each node. But if we are a little more strict in how many children a node can have, we create a structure that's easily searchable. Consider a binary tree, in which a node can have at most two children. We could define this in code like so:

```
typedef struct node
{
    int n;
    struct node* left;
    struct node* right;
}
node;
```

Binary Search Trees

- A *binary search tree* is a special type of binary tree. What's interesting about a binary search tree is that a node's left child is less than it and a node's right child is greater than it. Searching a tree like this is trivial then: if the current node is less than the value we're looking for, then we move to the right child; if the current node is greater than the value we're looking for, then we move to the left child. An implementation of search might look like this:

```
bool search(int n, node* tree)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (n < tree->n)
    {
        return search(n, tree->left);
    }
    else if (n > tree->n)
    {
        return search(n, tree->right);
    }
    else
    {
        return true;
    }
}
```

- Remember that the `->` operator means to access and dereference a field within a struct.
- Note that both the first if condition and the final else condition in the above are base cases. It's not necessary to place them both at the top of our logic.

Teaser

- Soon we'll start working in HTML, a markup language that allows you to specify what a web page should look like, and JavaScript, a programming language that allows you to execute logic within a browser. Our first web page will be implemented like so:

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello, world</title>
    </head>
    <body>
        hello, world
    </body>
</html>
```

- If you get really clever, you may be able to implement a page like Rob’s or even Hamster Dance.

Last updated 2013-10-30 20:49:09 PDT

Week 8, continued

Andrew Sellergren

Table of Contents

Announcements and Demos

- Problem Set 6 is one of the more challenging problem sets we will complete this semester, so be sure to leave yourself plenty of time to work through it! If you’re up for it, see how efficient your implementation is compared to your classmates' on The Big Board.
- Pre-Proposal for the Final Project is due Monday! It’s really just a casual, thought-provoking conversation you should have with your TF.
- Check out The CS50 Store to memorialize your time in the course! If you’re interested in submitting your own design, be sure to follow the specs outlined atcs50.net/design and listed here:
    - PNG
    - 200+ DPI
    - ⇐ 4000 x 4000 pixels
    - ⇐ 10 MB

The Internet

HTTP

- How does the internet actually work? When you type facebook.com into your browser (Chrome, Internet Explorer, Firefox, etc.), the browser makes an HTTP request. HTTP, which stands for *hypertext transfer protocol*, defines the language that the browser and web server speak to each other. Think of a web server exactly like a server at a restaurant: when you make a request of him, he brings it to you. In the context of the internet, the server is bringing you a web page written in HTML. More on HTML later.
- HTTP is a protocol for browsers and servers to talk to each other. Humans, too, have protocols for talking to each other. Consider that when you meet someone, you often greet him or her with a handshake. Browsers and servers also greet and acknowledge each other according to HTTP.
- Servers do a lot more than just serve web pages. To accommodate different types of requests, servers use different ports. The default port number for HTTP requests is 80. Navigating to facebook.com is identical to navigating to facebook.com:80 because the 80 is implied.
- To see HTTP in action, we can fire up a command-line program named `telnet`. We open up a terminal window and type `telnet www.facebook.com 80`. This presents us with a prompt like so:

```
Trying 31.13.69.32...
Connected to star.c10r.facebook.com.
Escape character is '^]'.
```

- 31.13.69.32 is an IP address. IP stands for *internet protocol*. An IP address is a unique (more or less) identifier for a computer on the internet. An IP address is to a computer what a mailing address is to a house. In this case, the IP address corresponds to one of Facebook's servers.
- Now we type the following:

```
GET / HTTP/1.1
Host: www.facebook.com
```

- In turn, we'll get a response from the server like this:

```
HTTP/1.1 302 Found
Location: http://www.facebook.com/unsupportedbrowser
Content-Type: text/html; charset-utf-8
X-FB-Debug: OigNZFku4U2xO68YDYkoMQs95BMNbmwwMqYVgo0yGx8=
Date: Wed, 30 Oct 2013 17:20:59 GMT
Connection: keep-alive
```

```
Content-Length: 0
```

- Facebook doesn’t like the fact that we’re pretending to be a browser, so it’s redirecting us to a site to tell us that. We can actually trick Facebook into thinking that we’re coming from a normal browser like Chrome by adding a line to our HTTP request:

```
GET / HTTP/1.1
Host: www.facebook.com
User-Agent: Mozilla/5.0 (Macintosh; Indel Mac OS X 10_8_5)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/30.0.1599.101
Safari/537.36
```

- Note that these user agent strings tell websites a lot about your computer. In this case, it tells Facebook that we’re using an Intel-based Mac running OS X 10.8.5 and version 30.0.1599.101 of Chrome.
- With this user agent string added to our HTTP request, we get a normal response back from the server. The server actually still redirects us, this time to the more secure HTTPS version of the site.
- Why do we write `GET /`? We’re requesting the root of the website. The root of the site is denoted with a slash just as the root of a hard drive is.
- If we switch gears and make an HTTP request to www.mit.edu, we get back an HTTP response that starts with `HTTP/1.1 200 OK` and actually contains the HTML that makes up their homepage. 200 is the "all is well" HTTP status code. You can see this same HTML if you go to View Source within your browser.

DNS

- How does your browser know the IP address of MIT’s web server? There are special servers called DNS, or domain name system, servers whose job it is translate hostnames like www.mit.edu into IP addresses.

TCP/IP

- TCP/IP is the protocol that defines how information travels through the internet. Information travels from source to destination via several *routers* in between. Routers are other servers that simply take in bytes and direct them elsewhere. We can see which routers our information passes through using a command-line program named `traceroute`. Each of the lines in the output represents a router that our request went through. Lines that are just three asterisks represent routers that ignore this type of request, so we don’t know where they are. On the right side, there are three time values which represent three measurements of the number of milliseconds it took to reach this router.

- Typically, information requires fewer than 30 hops between routers to get to its destination. If we run `traceroute www.mit.edu`, we see that the first few hops are Harvard's routers, but by step 6, we're in New York City. After that, the hops are obscured.
- If we run `traceroute www.stanford.edu`, we see that we can get from Boston to Washington DC in ~7 steps and less than 15 milliseconds! After that, we jump to Houston and LAX and finally to Stanford, all in under 90 milliseconds or so.
- There are other machines besides routers that sit between your computer and the information you're requesting. For example, that information may live behind a machine that restricts access known as a *firewall*. You can think of an HTTP request as an envelope with the return address being your IP and the to address being the IP of the web server. A very simple firewall might reject requests solely based on the IPs they came from, i.e. the return addresses written on the envelopes. A more advanced firewall might reject e-mails but not web requests, discriminating using the port number of the request.
- For the sake of efficiency and reliability, HTTP requests are broken up into chunks of information called *packets*. These packets don't have to follow the same path to reach their destination and some of them never will because they are dropped by routers in between, whether intentionally or unintentionally.
- For a more in-depth look at TCP/IP and the internet, check out Warriors of the Net.

Last updated 2013-11-01 18:55:47 PDT

Week 9

Andrew Sellergren

Table of Contents

Announcements and Demos

- This week we start our foray into web programming! The fundamentals and concepts from the first 9 weeks of the course will still play a role in how you program, but you'll find that tasks are an order of magnitude easier to accomplish now.
- After taking a course like CS50, you may start noticing mistakes in a show like [Numb3rs](#). First, IP addresses (at least v4 addresses), are of the

form w.x.y.z, where w, x, y, and z are all numbers between 0 and 255. Second, the language of the code you see in Charlie's browser window is Objective C and appears to be manipulating a variable named `crayon`, which has nothing to do with what Amita was ostensibly programming.

From Last Time

- Every computer on the internet has a (more or less) unique IP address. An IP address can be used for good in order to send and receive information. An IP address can also be used for evil if an adversary wants to track a particular computer.
- When you log onto a website, your browser (the client) is communicating with another computer on the internet (the server) via HTTP. HTTP is just one of many services like FTP and SMTP which sit on top of the internet. A single computer can support multiple different services on different ports.
- HTTP responses are tagged with a status code like one of the following:
  - 200 - OK
  - 301 - Moved Permanently
  - 302 - Found
  - 401 - Unauthorized
  - 403 - Forbidden
  - 404 - Not Found
  - 500 - Internal Server Error
- You're probably familiar with 404, which you'll see if a file has been deleted from a web server or if you've mistyped a URL. You can think of 500 as the analog of a segmentation fault: it means something really bad happened!

Web Debugging Tools

- Browsers these days have very powerful tools for debugging. In Chrome, there's Developer Tools, in IE, there's Developer Toolbar, and in Firefox, there's Firebug. Chrome is installed by default on the Appliance, so we'll take a peek at Developer Tools.
- If you right click on any part of a web page and select Inspect Element, the Developer Tools pane will be opened at the bottom. The Elements tab provides a much more readable and organized version of the HTML

of the web page. The Network tab allows you to poke around the HTTP requests that your browser executes. If we navigate to facebook.com, we'll see that the first such HTTP request is met with a 301 response code. We can also see the `GET / HTTP/1.1` request that we sent if we click on view source next to Request Headers.

- Why is Facebook responding with a "Moved Permanently" status? We didn't type www before facebook.com, so Facebook is adding it for us. They might be doing this for technical or even branding reasons.
- But the second HTTP request is also met with a 301 response code! This time we're being redirected to the HTTPS version of the site.
- When our request is finally answered with some content, it's a lot more than just one page of HTML. There are a number of scripts, stylesheets, and images that are received as well. This makes sense since web pages consist of more than just HTML.

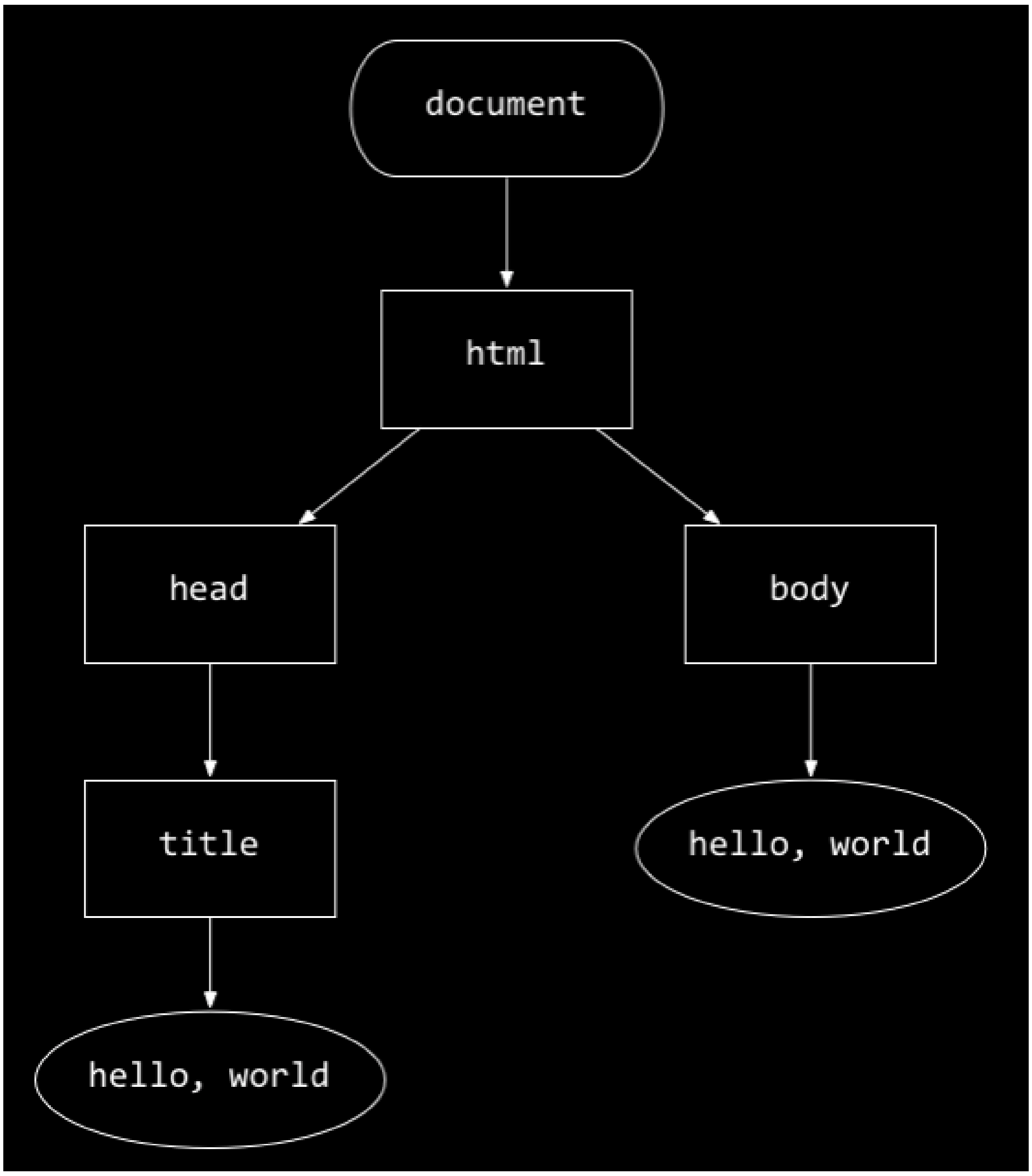
Intro to HTML and PHP

HTML

- HTML stands for hypertext markup language. The word "markup" implies that it's not a programming language, per se. For the most part you can't (and shouldn't) express logic in HTML. Rather, HTML allows you to tell the browser how to render a web page using start and end tags.
- A very simple web page we looked at last time was as follows:

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello, world</title>
    </head>
    <body>
        hello, world
    </body>
</html>
```

- The first line is the doctype declaration which tells the browser "here comes some HTML." Everything after that is enclosed in the `<html>` tag. This tag has two children, `<head>` and `<body>`. Each of these children, which begin with an open tag and end with a close tag, we'll call HTML *elements*. Given that certain elements are children of each other, we can use a tree structure to represent the HTML of a web page:



PHP

- As with other languages, let’s start by writing a "hello, world" program in PHP:

```
printf("hello, world\n");
```

- PHP is an *interpreted* language, so there’s no compiler. To run this program, we pass it to the PHP interpreter, which is conveniently named `php`. When we do so, we just get our line of code printed out to the screen. What went wrong? PHP has tags (`<?php` and `?>`) which tell the interpreter where code begins and ends. Without them, our lines of code are interpreted as raw HTML that should be passed over. Let's add them in:

```
<?php

    printf("hello, world\n");

?>
```

- If you run the command `ls` in your home directory on the Appliance, you'll see a directory called `vhosts`. You can actually run multiple websites on a single web server using this concept of *virtual hosts*. When a browser sends a request to that web server, it will include in that request the domain name that it wants a response from. Within the `vhosts` folder, there's a single folder named `localhost`. On another web server, there might be multiple folders here, one for each website that the server hosts. Within the `localhost` folder is a `public` folder where we can store our HTML source code. The Appliance has been configured such thathttp://localhost maps to this directory.
- If we navigate to http://localhost/index.html on Chrome in our Appliance, we get a 403 Forbidden error. Sad trombone. We can diagnose the problem by typing `ls -l` in our `public` directory:

```
-rw------- 1 jharvard students 135 Nov  4 13:29 index.html
```

- The `rw` on the far left stands for "read or write." The fact that there is only one such `rw` means that only the owner of this file can read it. We want everyone to be able to read it, so we run the command `chmod a+r index.html`. Now when we reload the page in our browser, we see "hello, world."

More HTML

- What's our favorite thing to do on a web page? Click on links of course! To add a link, we use the `<a>` tag:

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello, world</title>
    </head>
    <body>
        hello, <a href="https://www.cs50.net/">CS50</a>.
    </body>
</html>
```

- `href` is an *attribute* of the `<a>` tag. In this case, it contains the URL that we want to link to. What goes inside the element is the text we want the link to show: "CS50."

- Consider the security implications of this. If we change the `href` attribute to be https://www.fakecs50.net/, the text will still show CS50. This is misleading and potentially harmful to the user if the site he or she ends up on is malicious. We call this a *phishing attack*. Don't fall for it!
- Another useful tag is the `<div>` tag, which doesn't have any aesthetics, but allows us to organize the structure of our HTML. If we want to add aesthetics as well, we can use the `style` attribute like so:

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello, world</title>
    </head>
    <body>
        <div style="background-color: red;">
            Top of Page
        </div>
        <div>
            Bottom of Page
        </div>
    </body>
</html>
```

- The value of this `style` attribute is a language known as CSS, or cascading stylesheets. It specifies properties and their values to control what the page looks like. Instead of the English word "red," we can even specify a hexadecimal RGB value like `ff0000` for `background-color` and achieve the same effect.
- As you can see, it's very easy to make very hideous web pages. To see all the attributes of a given tag and all the different CSS properties, consult an online reference like W3Schools.

- Better than embedding CSS within our tags is factoring it out into a separate `<style>` tag at the top of our page:

```
<!DOCTYPE html>

<html>
    <head>
        <style>
            #top
            {
                background-color: #ff0000;
            }
            #bottom
            {
                background-color: #abcdef;
                font-size: 24pt;
            }
        </style>
        <title>hello, world</title>
    </head>
    <body>
        <div id="top">
            Top of Page
        </div>
        <div id="bottom">
            Bottom of Page
        </div>
    </body>
</html>
```

- Note that we can specify CSS properties that apply only to a single `<div>` by assigning it an `id` attribute that we reference within the `style` tag.
- Separating out CSS into a single `style` element is good practice because it is easier to reuse. In fact, we can (and should) even separate out CSS into a separate file altogether:

```
<!DOCTYPE html>

<html>
    <head>
        <link href="styles.css" rel="stylesheet"/>
        <title>hello, world</title>
    </head>
    <body>
        <div id="top">
            Top of Page
        </div>
        <div id="bottom">
            Bottom of Page
        </div>
    </body>
</html>
```

- The `<link>` tag is an example of an empty tag in that there is no need for a separate close tag. `styles.css` must be in the same folder as `index.html`, unless we specify the full or relative path, and it must be world-readable.

Implementing Google

- When you search for something on Google, the URL changes from google.com to something with a lot of information after the `/`. This information is actually a series of parameters that Google uses to create its search results for you. One such parameter is your search query. If you navigate to http://www.google.com/search?q=cats, you'll notice that the search term "cats" is already filled in for you. `q` is a key meaning "query" and its value, "cats," is specified after the `=`.

- Let's implement Google! First, we need an input box for a user's query:

```
<!DOCTYPE html>

<html>
    <head>
        <title>CS50 Search</title>
    </head>
    <body>
        <h1>CS50 Search</h1>
        <form>
            <input type="text"/>
            <input type="submit"/>
        </form>
    </body>
</html>
```

- With this `<form>` tag and a few `<input>` tags, we have a very simple search engine...that doesn't do anything. We need to specify an `action` for the `<form>` tag:

```
<!DOCTYPE html>

<html>
    <head>
        <title>CS50 Search</title>
    </head>
    <body>
        <h1>CS50 Search</h1>
        <form action="https://www.google.com/search" method="get">
            <input name="q" type="text"/>
            <br/>
            <input type="submit" value="CS50 Search"/>
        </form>
    </body>
</html>
```

- Now we're telling the form to submit its information directly to Google using the GET method. There are two methods for submitting form information, GET and POST. For now, just know that GET means the information is appended to the URL.

- We've also added a `name` attribute to the text input to match the URL parameter we saw that Google was using. We changed the text that the submit button displays to "CS50 Search" using its `value` attribute. Finally, we added a line break using the `<br/>` tag between the two inputs.
- When we type "cats" and click "CS50 Search," we end up on http://www.google.com/search?q=cats! We've implemented Google!

Frosh IMs

`froshims0.php`

- Back at the turn of the 19th century when David was a freshman at Harvard, the process of registering for intramural sports was painfully manual. You had to fill out a paper form and actually drop it off at the dorm room of the proctor in charge. David decided to change all that by implementing an online registration form. Although his original implementation was in Perl, we can recreate it in HTML and PHP:

```
<?php

    /**
     * froshims-0.php
     *
     * David J. Malan
     * malan@harvard.edu
     *
     * Implements a registration form for Frosh IMs.
     * Submits to register-0.php.
     */

?>

<!DOCTYPE html>

<html>
    <head>
        <title>Frosh IMs</title>
    </head>
    <body style="text-align: center;">
        <h1>Register for Frosh IMs</h1>
        <form action="register-0.php" method="post">
            Name: <input name="name" type="text"/>
            <br/>
            <input name="captain" type="checkbox"/> Captain?
            <br/>
            <input name="gender" type="radio" value="F"/> Female
            <input name="gender" type="radio" value="M"/> Male
            <br/>
            Dorm:
            <select name="dorm">
                <option value=""></option>
                <option value="Apley Court">Apley Court</option>
                <option value="Canaday">Canaday</option>
                <option value="Grays">Grays</option>
                <option value="Greenough">Greenough</option>
                <option value="Hollis">Hollis</option>
```

```
•                 <option value="Holworthy">Holworthy</option>
•                 <option value="Hurlbut">Hurlbut</option>
•                 <option value="Lionel">Lionel</option>
•                 <option value="Matthews">Matthews</option>
•                 <option value="Mower">Mower</option>
•                 <option value="Pennypacker">Pennypacker</option>
•                 <option value="Stoughton">Stoughton</option>
•                 <option value="Straus">Straus</option>
•                 <option value="Thayer">Thayer</option>
•                 <option value="Weld">Weld</option>
•                 <option value="Wigglesworth">Wigglesworth</option>
•             </select>
•             <br/>
•             <input type="submit" value="Register"/>
•         </form>
•     </body>
• </html>
```

- As before, we have `<head>` and `<body>` tags. Within the `<body>`, there's a `<form>` whose `action` attribute is `register0.php`. We see `<input>` tags with `type`set to "text," "checkbox," and "radio." Text and checkbox should be self-explanatory, but radio refers to the bulleted buttons for which the user can only choose 1 option. To create a dropdown menu, we use the `<select>` tag with `<option>` tags within it. Finally we have our submit button which displays "Register" as its`value` attribute.
- When we enter in values into this form and click "Register," we're taken to the `register0.php` URL that was specified in the `action` attribute of the form. Unlike with our CS50 Search example, this URL doesn't have any of our inputs embedded in it. That's because we used the POST method of sending data rather than the GET method.
- `register0.php` does nothing more than print out our inputs as an *associative array*. Whereas we only worked with numerically indexed arrays in C, PHP supports arrays that can use strings and other objects as keys. An associative array is really just a hash table! Because POST sends data via the headers rather than the URL, it is useful for submitting passwords, credit card numbers, and anything that's sensitive. It's also useful for sending data that's too large to embed in the URL, for example an uploaded photo.

conditions-1.php

- To get a feel for this new language, let's take a look at how we would implement `conditions-1.c` in PHP:

```
• <?php
•
•     /**
•      * conditions-1.php
•      *
```

```
         * David J. Malan
         * malan@harvard.edu
         *
         * Tells user if his or her input is positive, zero, or negative.
         *
         * Demonstrates use of if-else construct.
         */

        // ask user for an integer
        $n = readline("I'd like an integer please: ");

        // analyze user's input
        if ($n > 0)
        {
            printf("You picked a positive number!\n");
        }
        else if ($n == 0)
        {
            printf("You picked zero!\n");
        }
        else
        {
            printf("You picked a negative number!\n");
        }

    ?>
```

- The syntax for PHP is actually quite similar to that of C. Variable names in PHP are prefixed with a `$`. Variables also do not need to be declared with explicit types because PHP is a loosely typed language. In different contexts, PHP will implicitly cast variables from one type to another. `readline` is a new function, but the if-else construct is identical to C.

register-0.php

- `register0.php` is a quick example of commingling PHP and HTML:

```
<!DOCTYPE html>

<html>
    <head>
        <title>Frosh IMs</title>
    </head>
    <body>
        <pre>
            <?php print_r($_POST); ?>
        </pre>
    </body>
</html>
```

- Within the `<pre>` HTML tags, we enter PHP mode by inserting the `<?php` and `?>`. Once we're in PHP mode, we access a variable named `$_POST`. This is an associative array which PHP constructs for you whenever you pass in data via the POST method. If we had used the GET method, the data would be available in the`$_GET` variable. `print_r` is a function which prints recursively, meaning it prints everything that's nested within a variable. When we pass the `$_POST` variable to`print_r`, we see the four inputs that the user provided, each with a key that corresponds to the name attribute of the input. `$_POST` and `$_GET` are known as*superglobal* variables because they're available everywhere.

register-3.php

- In `register3.php`, we take the extra step of actually e-mailing the user's information:

```
<?php

    /**
     * register-3.php
     *
     * Computer Science 50
     * David J. Malan
     *
     * Implements a registration form for Frosh IMs.  Reports registration
     * via email.  Redirects user to froshims-3.php upon error.
     */

    // require PHPMailer
    require("PHPMailer/class.phpmailer.php");

    // validate submission
    if (!empty($_POST["name"]) && !empty($_POST["gender"]) && !empty($_POST["dorm"]))
    {
        // instantiate mailer
        $mail = new PHPMailer();

        // use SMTP
        $mail->IsSMTP();
        $mail->Host = "smtp.fas.harvard.edu";

        // set From:
        $mail->SetFrom("jharvard@cs50.net");

        // set To:
        $mail->AddAddress("jharvard@cs50.net");

        // set Subject:
```

```
        $mail->Subject = "registration";

        // set body
        $mail->Body =
            "This person just registered:\n\n" .
            "Name: " . $_POST["name"] . "\n" .
            "Captain: " . $_POST["captain"] . "\n" .
            "Gender: " . $_POST["gender"] . "\n" .
            "Dorm: " . $_POST["dorm"];

        // send mail
        if ($mail->Send() == false)
        {
            die($mail->ErrInfo);
        }
    }
    else
    {
        header("Location: http://localhost/src9m/froshims/froshims-3.php");
        exit;
    }
?>

<!DOCTYPE html>

<html>
    <head>
        <title>Frosh IMs</title>
    </head>
    <body>
        You are registered!  (Really.)
    </body>
</html>
```

- The `require` function in PHP is similar to the `#include` directive in C. First, we check that the user's inputs are not empty using the appropriately named function`empty`. If they aren't, we begin using a library called PHPMailer to create and send an e-mail. To use this library, we create a new object of type `PHPMailer` named`$mail` and we call the `IsSMTP` method of that object by writing `$mail->IsSMTP()`. We set the mail server to be `smtp.fas.harvard.edu` and call a few more methods to set the to and from addresses as well as the subject and body. We know the names of these methods simply by reading the documentation for PHPMailer. To construct a body for our message, we use the dot operator (`.`) to concatenate the user's inputs into one long string. Finally, we call the `Send` method and voila, we have just registered a user for freshman intramurals!

- One interesting implication of this is that it's pretty easy to send e-mails from any e-mail address to any e-mail address. Be wary!

Crash Course in PHP

- PHP is a programming language which thankfully is quite accessible since its syntax is very similar to that of C.
- There's no `main` function in PHP.
- Conditions, Boolean expressions, switches, and loops all have the same syntax in PHP as they do in C. Switches in PHP have the additional capability to use strings as the variables that define cases.
- In PHP, variable names begin with `$`. To declare an array of numbers, you can write the following:

```
$numbers = [4, 8, 15, 16, 23, 42];
```

- Interestingly, there's no explicit size or type associated with the variable `$numbers`. C is a so-called strongly typed language in that it requires you to declare an explicit type for every variable. PHP is loosely typed. You don't need to inform PHP in advance as to what data type you're going to fill a variable with. Consequently, functions can also return multiple types.
- One extra loop type available in PHP is the foreach loop:

```
foreach ($numbers as $number)
{
    // do this with $number
}
```

- In PHP, there's another type of array called an associative array whose indices can be strings, objects, etc.:

```
$quote = ["symbol" => "FB", "price" => "49.26"];
```

- Under the hood, associative arrays are implemented as hash tables. Performance-wise, they're slower to access and require more memory than the numerically indexed arrays we worked with in C, but they're very convenient to use!
- So far, we've used the superglobal variables `$_GET` and `$_POST` which store input from forms. In addition to these superglobal variables, there are also the following:
    - `$_SERVER`
    - `$_COOKIE`
    - `$_SESSION`

Model-view-controller

- As you begin to design web applications, you'll want to think about how to organize your code. One paradigm for organizing code is called Model-view-controller (MVC). The View encapsulates the aesthetics of the website. The Model handles interactions with the database. The Controller handles user requests, passing data to and from the Model and View as needed.
- Let's try to create a course website for CS50 using the MVC framework. In version 0, the pages are well organized into separate directories, but there are a lot of files with very similar code.
- To see how we might abstract away some of the logic, let's jump ahead to version 5:

```
<?php require("../includes/helpers.php"); ?>

<?php render("header", ["title" => "CS50"]); ?>

<ul>
    <li><a href="lectures.php">Lectures</a></li>
    <li><a href="http://cdn.cs50.net/2013/fall/lectures/0/w/syllabus/syllabus.html">Syllabus</a></li>
</ul>

<?php render("footer"); ?>
```

- Now the header and footer are being automatically generated by the function `render`. This is better design because we can change the header and footer of all the pages within our site just by changing a few lines of code.

Teaser

- Soon we'll dive into the world of databases and even implement our own e-trading website!

Last updated 2013-11-07 00:27:04 PST

Week 9, continued

Andrew Sellergren

Table of Contents

From Last Time

- We talked about how web servers send web pages to your browser when they request it. Those web pages are written in a markup language called HTML. CSS is a language that controls aesthetics like font size and color. Ideally, for the sake of reusability and organization, CSS lives in a separate file that gets linked into the HTML. One downside of this approach is that it requires a separate HTTP request to be fetched and thus it might increase latency. Smart browsers will often save a copy of CSS files and other static content locally so that they don't have to be refetched. This local repository is called a *cache* and the act of saving to it is called *caching*.
- Unlike HTML, PHP is a true programming language in that it can express logic. When we need our web pages to change their content dynamically, we use PHP to output HTML that the browser then renders. Note that style-wise, we don't really care about indentation in the HTML, only in the PHP.

Reimplementing `speller`

- Unless you really need to your program to be highly performant, perhaps because your data sets are huge, you'll probably reach for a language like PHP rather than a language like C. Your C implementation of `speller` is no doubt very fast, but think how long it took you to program. In contrast, let's see how easy it is to re-implement the logic in PHP.

```
<?php
    $size = 0;

    $table = [];

    function load($dictionary)
    {
        global $size, $table;
        foreach (file($dictionary) as $word)
        {
            $table[chop($word)] = true;
            $size++;
        }
        return true;
    }

?>
```

- A hash table in PHP is as easy to implement as declaring an associative array. We can use the words themselves as indices in the array. `file` is a function that reads in a file and hands it back to you as an array of lines.
- A few other details include declaring our variables as globals, stripping the whitespace from the word, and incrementing the `$size` variable.
- Let's take a stab at `check`:

```
<?php
    $size = 0;

    $table = [];

    function check($word)
    {
        if (isset($table[strtolower($word)]))
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    function load($dictionary)
    {
        global $size, $table;
        foreach (file($dictionary) as $word)
        {
            $table[chop($word)] = true;
            $size++;
        }
        return true;
    }
```

```

?>
```

All we need to check is if the index for a particular word is set to know whether it's in the dictionary. `size` and `unload` are similarly trivial to implement:

```
<?php
    $size = 0;

    $table = [];

    function check($word)
    {
        global $table;
        if (isset($table[strtolower($word)]))
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    function load($dictionary)
    {
        global $size, $table;
        foreach (file($dictionary) as $word)
        {
            $table[chop($word)] = true;
            $size++;
        }
        return true;
    }

    function size()
    {
        global $size;
        return $size;
    }

    function unload()
    {
        return true;
    }

?>
```

If it's this easy to implement this in PHP, why bother implementing it in C? When you run `speller` in C over the King James Bible, you get a runtime of about 0.5 seconds. When you run `speller` in PHP over the King James Bible, you get a runtime of about 3 seconds. That's an order of magnitude difference! One reason is that PHP is an interpreted language, so it has to be read and executed line by line. C on the other hand is already compiled into 0s and 1s by the time it is executed. There are ways to cache the results of the PHP interpreter so that content can be served up faster on the web, but it will likely never compete with C in terms of performance.

Sessions and Cookies

- There are a number of variables called superglobals that are available everywhere in PHP programs:
    - `$_COOKIE`
    - `$_GET`
    - `$_POST`
    - `$_SERVER`
    - `$_SESSION`
- A *cookie* is a long unique identifier that is planted on your computer when you visit a web page, typically via the "Set-cookie" HTTP header. This identifier gets passed back to the web server via the "Cookie" HTTP header when you revisit that web page so that it knows who you are and can serve you content that is specific to you. Think of it like a hand stamp at an amusement park.

counter.php

- `counter.php` demonstrates the use of sessions to implement a simple counter for a user's visits:

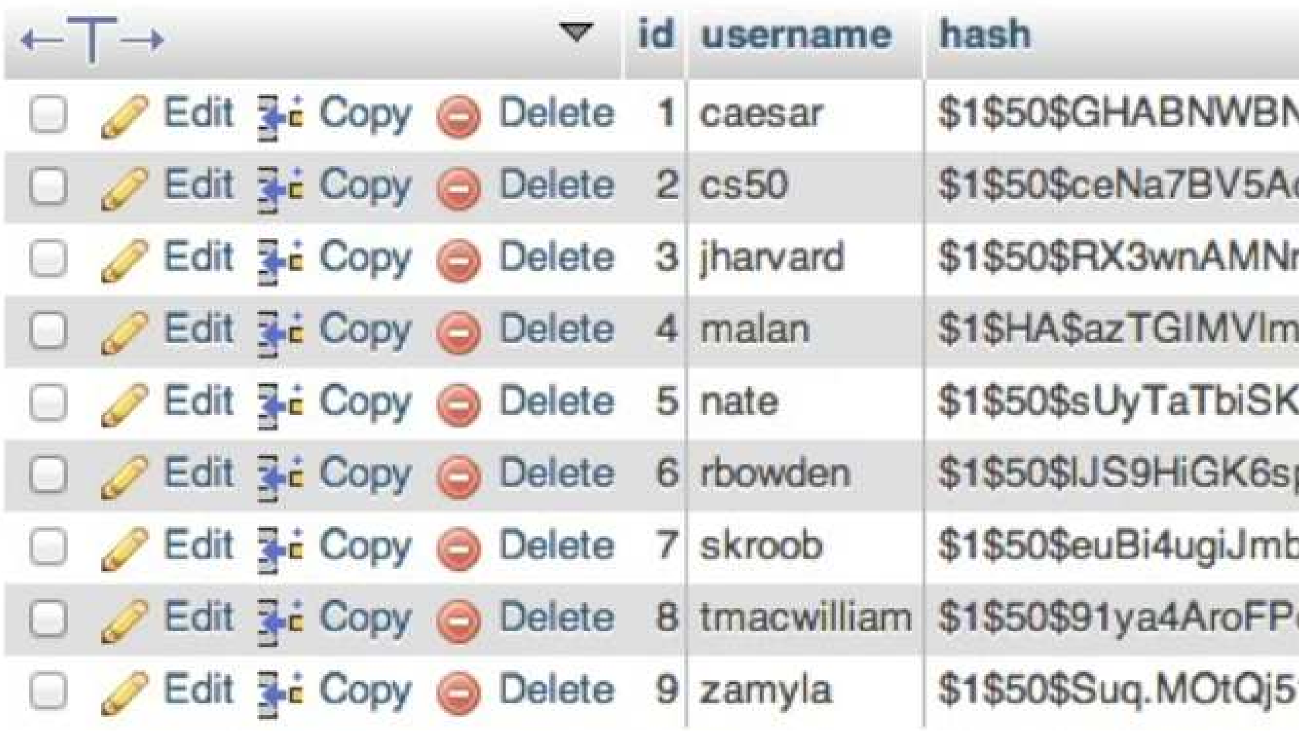
```php
<?php
    // enable sessions
    session_start();

    // check counter
    if (isset($_SESSION["counter"]))
    {
        $counter = $_SESSION["counter"];
    }
    else
    {
        $counter = 0;
    }

    // increment counter
    $_SESSION["counter"] = $counter + 1;

?>
```

```
<!DOCTYPE html>
<html>
    <head>
        <title>counter</title>
    </head>
    <body>
        You have visited this site <?= $counter ?> time(s).
    </body>
</html>
```

- Note that `$counter` exists in scope even outside the curly braces of the if-else blocks.
- `session_start()` needs to be called in order to use HTTP sessions, which allow a web server to keep track of a user across visits. This function call tells the server to send the "Set-cookie" header with a long alphanumeric string after the keyword `PHPSESSID`.
- Question: are PHP variables always global? Yes, unless they are declared within a function.
- The `<?= $counter ?>` syntax is shorthand for switching into PHP mode and printing out a variable.
- If cookies are used to identify users, then impersonating a user is as easy as stealing a cookie. The defense against this is to encrypt HTTP headers using SSL. This might be familiar to you as HTTPS. Even SSL encryption can be broken though!

SQL

- We introduced SQL last time as a language to interact with databases. You can think of a database as an Excel spreadsheet: it stores data in tables and rows.
- There are four basic SQL commands:
    - `SELECT`
    - `INSERT`
    - `UPDATE`
    - `DELETE`
- For Problem Set 7, we've set you up with a database and an application named phpMyAdmin (not affiliated with PHP) to interact with it. In that database, there is a users table with id, username, and hash columns:



←T→				id	username	hash
☐	Edit	Copy	Delete	1	caesar	$1$50$GHABNWBN
☐	Edit	Copy	Delete	2	cs50	$1$50$ceNa7BV5A
☐	Edit	Copy	Delete	3	jharvard	$1$50$RX3wnAMN
☐	Edit	Copy	Delete	4	malan	$1$HA$azTGIMVIm
☐	Edit	Copy	Delete	5	nate	$1$50$sUyTaTbiSK
☐	Edit	Copy	Delete	6	rbowden	$1$50$IJS9HiGK6s
☐	Edit	Copy	Delete	7	skroob	$1$50$euBi4ugiJmb
☐	Edit	Copy	Delete	8	tmacwilliam	$1$50$91ya4AroFP
☐	Edit	Copy	Delete	9	zamyla	$1$50$Suq.MOtQj5

- Presumably, username is unique, so why bother with an id column? An id is only 32 bits because it's an integer, whereas username is a variable-length string. Comparing strings is not as fast as comparing integers, so lookups by id will be faster than lookups by username.
- SQL supports the following types:
    - `CHAR`
    - `VARCHAR`
    - `INT`
    - `BIGINT`
    - `DECIMAL`
    - `DATETIME`
- A `CHAR` is a fixed-length string whereas a `VARCHAR` is a variable-length string. It's slightly faster to search on a `CHAR` than a `VARCHAR`.
- SQL tables also have indexes:
    - `PRIMARY`
    - `INDEX`
    - `UNIQUE`
    - `FULLTEXT`
- Adding indexes makes searching a table faster. Because we know that we'll be using id to look up rows and we know that id will be unique, we can specify it as a primary key by choosing `PRIMARY` from the Index dropdown menu. Although we decided that id should be the lookup field rather than email, we still want email to be unique, so we

choose `UNIQUE` from the Index dropdown menu. This tells MySQL that the same e-mail address should not be inserted more than once. Choosing `INDEX` tells MySQL to build a data structure to make searching this column more efficient, even though it's not unique. Similarly, `FULLTEXT` allows for wildcard searching on a column.

- Databases can be powered by one of several different engines:
    - InnoDB
    - MyISAM
    - Archive
    - Memory
- This is one more design decision that we have to make, but we won't trouble ourselves with the particulars for right now.

Race Conditions

- Let's motivate our discussion of race conditions with a short story. Imagine that you come home from class and open the refrigerator to find that there's no milk. You close the door, walk to the store, buy some milk, and come home. In the meantime, your roommate has also come home and noticed that there's no milk. He went out to buy some more, so when he comes back, you now have two cartons of milk, one of which will certainly spoil before you can drink it.
- How could you have avoided this? You could have left a note on the refrigerator or you could have even padlocked the door. In the real world, we might run into this same situation when withdrawing from an account from two different ATMs. If both ATMs query for the balance of the account at the same time, they will return the same number. Then when you withdraw $100 from each ATM, 100 will be subtracted from that number on each ATM instead of the cumulative 200 that should be subtracted. Free money!
- For Problem Set 7, you'll need to keep track of users' cash balances and shares of stock. If you issue a `SELECT` statement to get these numbers and then issue an `UPDATE` statement to update them, you may run into the same problem as the milk or ATM situations we just discussed. Better to run one *atomic* operation like so:

```
INSERT INTO table (id, symbol, shares) VALUES(7, 'DVN.V', 10)
    ON DUPLICATE KEY UPDATE shares = shares + VALUES(shares);
```

- Atomicity means that multiple operations happen at the same time or don't happen at all. The above statement is both an `INSERT` and

an UPDATE statement, where the UPDATE statement is executed only if the INSERT fails because there's already an entry for that stock.
- Some database engines also support *transactions*:

```
START TRANSACTION;
UPDATE account SET balance = balance - 1000 WHERE number = 2;
UPDATE account SET balance = balance + 1000 WHERE number = 1;
COMMIT;
```

- Because these two UPDATE statements are part of the same transaction, they will only succeed if *both* succeed.

JavaScript

- JavaScript is an interpreted programming language that executes clientside. Whereas PHP is executed on the server, JavaScript is downloaded by the browser and executed there.
- If you've ever been on page that updates its content without reloading, you've seen JavaScript in action. Behind the scenes, it has actually made another HTTP request.
- JavaScript is also used to manipulate the DOM, the document object model, the tree of HTML that we saw earlier.
- The syntax for conditions, Boolean expressions, loops, switch statements is much the same in JavaScript as it is in PHP and C. One new type of loop exists, however:

```
for (var i in array)
{
    // do this with array[i]
}
```

- Functionally, this loop is equivalent to the foreach loop in PHP.
- The syntax for arrays is slightly different:

```
var numbers = [4, 8, 15, 16, 23, 42];
```

- Note that we don't have the $ prefix for variables anymore. We still don't specify a type for the variable, though.
- Another built-in data structure in JavaScript are objects:

```
var quote = {symbol: "FB", price: 49.26}
```

- Objects are similar in functionality to associative arrays in PHP or structs in C.
- JavaScript Object Notation, or JSON, is a very popular format these days. If you work with APIs like Facebook's for your Final Project, you'll be passed data in JSON. JSON is quite useful because it is self-describing: each of the fields in an object is named.

dom-0.html

- Let’s start with a "hello, world" for JavaScript:

```
<!DOCTYPE html>

<html>
    <head>
        <script>

            function greet()
            {
                alert('hello, ' + document.getElementById('name').value
+ '!');
            }

        </script>
        <title>dom-0</title>
    </head>
    <body>
        <form id="demo" onsubmit="greet(); return false;">
            <input id="name" placeholder="Name" type="text"/>
            <input type="submit"/>
        </form>
    </body>
</html>
```

- What’s interesting here is that we can embed JavaScript directly in HTML using the `<script>` tag. The `alert` function is a quick and dirty way of displaying output via a pop-up window. Convention in JavaScript is to use single quotes for strings, by the way.
- In JavaScript, there exists a special global variable named `document` that contains the entire tree structure of the HTML. The `document` object also has functions associated with it known as methods. One such method is `getElementById` which retrieves an HTML element with the specified `id` attribute.
- `greet` is called when the form is submitted because it is passed to the `onsubmit` attribute of the form. After that, we have to `return false` because otherwise the form will actually submit and redirect to whatever its `action` attribute is.

dom-2.html

- jQuery is a library that adds a great deal of convenience to writing JavaScript. Take a look at it in `dom-2.html`:

```
<!DOCTYPE html>

<html>
    <head>
        <script                          src="http://code.jquery.com/jquery-
latest.min.js"></script>
```

```
        <script>

            $(document).ready(function() {
                $('#demo').submit(function(event) {
                    alert('hello, ' + $('#name').val() + '!');
                    event.preventDefault();
                });
            });

        </script>
        <title>dom-2</title>
    </head>
    <body>
        <form id="demo">
            <input id="name" placeholder="Name" type="text"/>
            <input type="submit"/>
        </form>
  </body>
</html>
```

- For now we'll wave our hands at the first line of JavaScript above. Basically, it just waits till the document has loaded before executing anything. `$('#demo')` is equivalent to `document.getElementById('demo')`.
- One feature of JavaScript that we're leveraging here is the ability to pass functions as objects to other functions. The only argument to the `submit` method is an*anonymous function* that takes in its own argument `event`.

Ajax

`ajax-2.html`

- Ajax is the technology that allows us to request external data without a page refresh. Take a look at `ajax-2.html`:

```
<!--

ajax-2.html

Gets stock quote from quote.php via Ajax with jQuery, embedding result
in page itself.

David J. Malan
malan@harvard.edu

-->

<!DOCTYPE html>

<html>
    <head>
```

```
        <script src="http://code.jquery.com/jquery-latest.min.js"></script>
        <script>

            /**
             * Gets a quote via JSON.
             */
            function quote()
            {
                var url = 'quote.php?symbol=' + $('#symbol').val();
                $.getJSON(url, function(data) {
                    $('#price').html(data.price);
                });
            }

        </script>
        <title>ajax-2</title>
    </head>
    <body>
        <form onsubmit="quote(); return false;">
            Symbol: <input autocomplete="off" id="symbol" type="text"/>
            <br/>
            Price: <span id="price">to be determined</span>
            <br/><br/>
            <input type="submit" value="Get Quote"/>
        </form>
    </body>
</html>
```

- The `quote` function appears to be constructing a URL with a stock symbol as a parameter. If we visit `quote.php?symbol=GOOG` directly, we'll get some JSON spit out to the screen that includes a stock price. In the JavaScript above, we're asking for that JSON programmatically, then passing it to a function that inserts it into the DOM.

Teaser

- Check out `geolocation-0.html` which seems to know where you are!

Last updated 2013-11-10 06:44:05 PST

Week 10

Andrew Sellergren

Table of Contents

Announcements and Demos

- Please welcome Jonathan Zittrain, Professor at Harvard Law School, Harvard Kennedy School, and Harvard School of Engineering and Applied Sciences! He just finished teaching CS 42: Controlling Cyberspace.

Technology as Owned vs. Unowned

- Most technologies are owned, but the Internet is unowned.
- Some examples of owned technology:
  - IBM System 360, used in an actuary, bank, insurance company, programmed by the vendor (IBM); part of its business model was to keep it apart from the consumer
  - Friden Flexowriter, like a standard typewriter but also made indentations in a tape that could be fed back into the machine to retype automatically; could cut and paste to create mail merges
  - Brother Smart Word Processor, only dealt with data, not programmable, no room for surprise

    - CAT scan, operates exactly as designed except for one unowned piece, the personal computer PC
- In 1977, at the West Coast Computer Fair, Steve Jobs presented the first reprogrammable machine, the Apple II, to 10,000 “computer freaks”
- Within two years, the first digital spreadsheet (VisiCalc) was developed by Bob Frankston and Dan Bricklin and the Apple II started flying off the shelves
- The Apple II was unowned in the sense that it accepted contributions from others and if something went wrong with it, it wasn’t clear that it was Apple’s fault; this model became the model for all that followed
- It’s not up to you, for example, to put the 7th blade in your razor!

Unowned Technologies

- Technologies that now are unowned didn’t necessarily have to end up that way, but they did.
- Some key features of unowned technologies:
    - origins in a backwater
    - ambitious but not fully planned
    - contribution welcomed from all corners
    - success beyond expectation
    - influx of usage
    - success is threatened
- In the early 20th century, AT&T owned the telephone networks. Technologies like the Hush-A-Phone, a telephone silencer, were initially frozen by licensing battles with AT&T, who feared that it would damage their network. Neither could you buy a telephone; you had to get it from the telephone service provider.
- Soon CompuServe emerged as a leader in the network space. Their model was very much after owned technology; if you wanted a new category of actions for users, you had to petition them to put it on the home screen.
- Enter ARPANET, the predecessor to the Internet as we know it. Its founders didn’t expect to make any money off it and didn’t have the resources to roll it out everywhere. IBM even said in 1992 that it would be impossible to build a corporate network using TCP/IP.

- Think of packets on the Internet like beer in Fenway Park. It can get almost all the way from the vendor to the buyer, but the last distance it travels is through a number of other spectators. They have no real contract with the buyer, but they pass it along anyway. Similarly, there are entities on the Internet that handle packets properly despite having no relationship with the sender or the receiver.

Hourglass Architecture

- The Internet has been said to have an Hourglass Architecture in which Internet Protocol is the bridge between software applications and hardware devices.
- We haven't messed around with the Internet all that much since its inception. We didn't implement a lot of features for it, but instead left that up to the end devices and applications to implement them.

Clarke's Third Law

- Consider these words of wisdom:

  Any sufficiently advanced technology is indistinguishable from magic.

  — Arthur C. Clarke

  Witchcraft to the ignorant…Simple science to the learned.

  — Leigh Brackett

- Part of CS50 is to help you to understand what others consider magic! Try it out, take it for a spin, see if you can change the world.
- One such person who changed the world was Tim Berners-Lee, who developed the first protocol for the World Wide Web. So too did Peter Tattam, creator of Trumpet Winsock, the first implementation of Windows network software.
- Unowned technologies like the World Wide Web are what enabled other unowned technologies like Wikipedia to exist.

The Cycle

- Although certain technologies have begun as unowned, we see that they often become owned as their success is threatened.
- As success is threatened, companies will go to great lengths to protect it. When the music industry first started producing compact discs (CDs),

they didn't imagine that the data could ever be copied from those discs. They went so far as to add borderline malicious rootkits to the CDs so that any attempt to copy the data would be thwarted.

Security

- This is also a lesson in trust and security. Malicious software has only gotten more sophisticated, beginning with the likes of the Storm Worm and progressing to the hardly detectable Stuxnet.
- An amusing anecdote: the Cap'n Crunch Bosun whistle emitted a ton at the exact frequency that AT&T recognized as an idle line. If you blew it into the telephone receiver, you could get free long distance! Because theirs was an owned technology, AT&T could quickly fix this vulnerability. Vulnerabilities in unowned technologies, for example viruses, malicious links, or even remote access tools (RATs), cannot be so easily fixed.

Technology as Hierarchy vs. Polyarchy

- In addition to unowned vs. owned, there is another axis by which we might measure technology: hierarchy vs. polyarchy. Hierarchy means there's only one choice whereas polyarchy means everyone is on his own.
- In the quadrant of polyarchy and unowned, we have trouble protecting ourselves because not everyone will always be aware of what's going on. Consider the malicious software warnings you've gotten and probably ignored.
- In the quadrant of hierarchy and owned, we have entities like the government who, despite their best efforts, aren't really able to improve security because they really don't have any control over it. Consider the Information Sharing and Analysis Centers (ISACs) that don't really have any power or the Focus Group on Next Generation Networks (FGNGN) that proposed a much more complicated version of the Internet that will probably never come to fruition.
- In the quadrant of polyarchy and owned, we have the corporate sector. Companies such as RSA Security offer solutions like two-factor authentication. This quadrant is perilous because it has some inefficiencies and inequalities built in. Consider the App Store by which iPhone and Mac software is curated: they decide not only what is "safe," but what is not offensive, not resource-intensive, and more. Consider

the Nook and the Kindle, both of which control the content that you read.

- In the final quadrant, that of hierarchy and unowned, there is an opportunity for renewal. Consider Wikipedia, which polices itself because of a distributed population of unpaid, caring administrators. Consider mesh networking, which could prevent a single entity from being able to pull the plug on the Internet. Consider the work of the Berkman Center to pre-cache linked pages so that their content can still be viewed even if the original host goes down.

A Final Question

- As a CS50 grad, who are you in this riddle? You have a tool with which you can change everything. Use it to forge systems that distribute power rather than focus it.

Last updated 2013-11-14 01:12:38 PST

Week 10, continued

Andrew Sellergren

Table of Contents

Announcements and Demos

- In case you haven't seen it yet, check out the story of Saroo, the boy who overslept on a train and found his family 25 years later using Google Maps.
- We know from collecting statistics that about 50% of you won't continue on to take any more CS courses. That's okay! One of our overarching goals is to empower you to understand the world of technology as you pursue other knowledge. Bring your ideas and your Final Projects to other departments!
- Quiz 1 will take place on Wednesday 11/20! Details are available on the course website.
- Problem Set 8 is due Friday 11/15. One minor point of clarification: if you're running into issues with Google Earth crashing and you're positive it's not your fault, try disabling buildings with the following line of code:
- `earth.getLayerRoot().enableLayerById(earth.LAYER_BUILDINGS, false);`

  We don't really recommend this because there will be no buildings, but it's a failsafe nonetheless. * The CS50 Hackathon will be on 12/4! This is your opportunity to bang out some code for your Final Project among friends and food. * The CS50 Fair will be on 12/9! This is the climax of the course in which you display your Final Project to friends and family. There will also be recruiters from various companies so you can chat about opportunities.

From Last Time

- We used the superglobal `$_SESSION` to track the user's ID so that she doesn't have to sign in every time she visits the site. HTTP is a stateless protocol. Once you connect to a server and download some HTML, JavaScript, and CSS, your browser's loading icon stops spinning because the connection has been terminated.
- Sessions are enabled via cookies, usually a big random number planted on your computer's hard drive or in its RAM. You can think of cookies as virtual handstamps for the server to remember who you are.
- There's a threat here, as well. If you steal someone's cookie, you can hijack his session and impersonate him.
- We looked briefly at Ajax, a technique that allows for fetching data from a server without refreshing the page. Google Maps doesn't load the

entire world when you browse it; rather, it loads one square at a time and fetches more as needed using Ajax.

Final Project

- Just to plant one seed in your mind, check out the list of e-mail addresses for the various cell providers here. With these, you can send text messages programmatically! Be careful, lest you send some 20,000 text messages mistakenly, as David did during lecture last year.
- Receiving text messages is a little more difficult. You can use the service provided by textmarks.com. For example, if you send a text message to 41411 like "SBOY mather quad," you'll get a response from the CS50 Shuttleboy app.
- Consider using Parse as your backend database instead of MySQL!
- CS50 has its own authentication service called CS50 ID. Check out the manual to see how to verify that a user is someone from Harvard.

Web Hosting

- Check out your options for web hosting if you want your Final Project to live outside of the Appliance. Namecheap is just one!
- To see who owns a particular domain, you can look it up using `whois` from the command line. Under the "Name Servers" heading, you'll see a list of servers that are the canonical sources for returning the IP address of the domain you looked up. When you type in this domain into your browser, the browser will eventually query these name servers to find the final IP address. When you register for web hosting, you'll need to tell the registrar what your name servers are. Since CS50 uses DreamHost, you'll enter in NS1.DREAMHOST.COM, NS2.DREAMHOST.COM, and NS3.DREAMHOST.COM if you use CS50's hosting account.
- SSL stands for *secure sockets layer* and is indicated by a URL that begins with https. To use SSL for your own website, you need a unique IP address for which you'll have to pay a web hosting company a few more dollars per month.

Security

- As a random segue into security, check out the first volume of CS50 Flights.

- As we talked about on Monday, it’s important to be careful when installing software. Often you’ll be prompted to give permission to an installer as a security measure because it needs to run as an administrator. This has very serious security implications because you’re giving this installer the ability to execute almost any command on your computer.
- The trust you implicitly or explicitly give to the software you run can easily be abused. Sony got a lot of flak a few years ago for including rootkits on the CDs they sold. These rootkits would actually hide themselves so that you couldn’t see they were running if you opened Task Manager.
- What does the padlock icon on a website mean in terms of security? Virtually nothing. But we’ve been conditioned to think that a website is secure when we see that padlock. That means it’s just as easy for an adversary to put a padlock on his malicious website and trick you into trusting him.
- Some browsers like Chrome go one step further in showing the owner of the SSL certificate. When you navigate to Bank of America’s website, Chrome shows "Bank of America Corporation [US]" in green in the address bar.
- But how many of you have actually noticed or changed your behavior because of these security measures?

Session Hijacking

- You can see the actual value of the cookie that Facebook plants on your computer by using Developer Tools in Chrome. Usually this cookie is planted when you first visit Facebook. But how did you get to Facebook? You probably didn’t type "https" to begin with, so you must have be redirected to the SSL-enabled version of the website. During that redirection, your cookie was forwarded along. If a bad guy is on the same network on you, he may be able to intercept this cookie while you’re being redirected. This attack is called *session hijacking*.

Man in the Middle

- A bad guy could even intercept your HTTP request and respond with his own fake version of Facebook in order to steal your credentials. This attack is called *man in the middle*.

SQL Injection Attack

- Let's focus on server-side security now. Imagine you accept a username and password from a user and execute a query against your database like so:

```
$username = $_POST["username"];
$password = $_POST["password"];
query("SELECT * FROM users WHERE username='$username' AND password='$password'");
```

  This looks reasonable and correct, but it's vulnerable to a *SQL injection attack*. What if the user enters `skroob` as his username and `12345' OR '1' = '1` as his password. Now the query you execute looks like the following:

```
SELECT * FROM users WHERE username='skroob' AND password='12345' OR '1' = '1'
```

- Because 1 = 1, this query will always return a row even if 12345 is not the correct password for username skroob.
- In Problem Set 7, we asked you to use question marks as placeholders for user input when executing the `query` function. One thing the `query` function does with these question marks is guarantee that user input will be properly escaped.

Week 12

Andrew Sellergren

Table of Contents

Announcements and Demos

- This was CS50.
- Recall what we said at the beginning of the course:

  what ultimately matters in this course is not so much where you end up relative to your classmates but where you, in Week 12, end up relative to yourself in Week 0
- 73% of you had no prior CS experience before joining this course! Congratulations on no longer being part of "those less comfortable" if you once were!
- Can you guess what the significance of the following numbers from Quiz 1 is?

  ```
  34 59 20 106 36 52
  ```

  E-mail heads@cs50.net if you think you know!
- We've gone from Scratch to CS50 Shuttle! As you dive into Final Projects, keep in mind how far you've come. Even Milo has his own website.
- If you want to continue developing software after CS50, there exist a number of different tools to help you do so:
  - Mac OS
    - CS50 Appliance

        - Code::Blocks
        - Eclipse
        - NetBeans
        - XAMPP
        - Xcode
    - Windows
        - CS50 Appliance
        - Code::Blocks
        - cygwin
        - Eclipse
        - NetBeans
        - Visual Studio
        - XAMPP
    - Linux
        - CS50 Appliance
        - clang
        - Code::Blocks
        - Eclipse
        - NetBeans
        - XAMPP
    - Congrats to Richard for topping the Big Board from Problem Set 6! Also happy birthday to Mike! Also congrats to Chris, Layla, Raul, Daniel, and Daniel for identifying and taking pictures with so many computer scientists from Problem Set 5!
- A special thank you to Joseph, R.J., and Lucas, our head teaching fellows; to Zamyla, our fearless walkthrough leader; to Rob, our preceptor; and to Lauren, our course head for making CS50 possible! And, of course, thank you to all 102 members of the staff! Want to be a part of the fun next year? Apply now!
- We'd also like to applaud Gabriel for his efforts in bringing CS50 to his high school in Brazil, there known as CC50. He started teaching CC50 as a 17-year-old back in 2011 and now he's a teaching fellow of our own!

- The Hackathon will be held on 12/4! It's an opportunity for you to work independently or with your partner(s) to finish up your Final Project. Don't wait till next Wednesday to start coding!
- The CS50 Fair is on 12/9! There you'll have a chance to show off your Final Project to friends and family as well as mingle with friends from industry including:
    - Amazon Web Services
    - APT
    - Box
    - Bridgewater
    - eBay
    - Facebook
    - Google
    - Kayak
    - Microsoft
    - Quora

This is CS50 Jeopardy!

- Those review questions you were asked to submit as part of the last problem set have now become part of a game show!
- Question: how is C typed? Answer: strongly!
- Question: what is the answer to life, the universe, and everything (in binary)? Answer: 101010.
- Question: what does the fox say? Answer: ring ding ding ding ding ding ding.
- Question: what used to allow you to make free long distance phone calls? Answer: the Captain Crunch whistle!
- Question: what does the status code 403 mean? Answer: Forbidden.

Farewell

- It has been a pleasure teaching you this semester! We leave you with a reel of outtakes.

Last updated 2013-11-27 20:07:40 PST