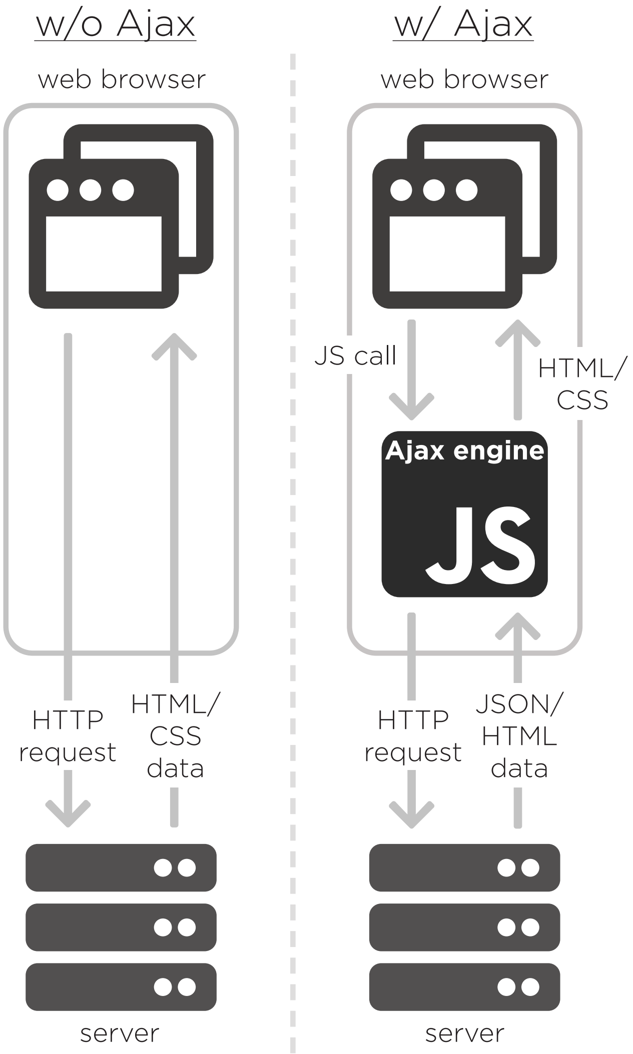
CS50 Ajax

Overview

Back in the 1990s, any time a website was slightly updated with new information, the entire webpage had to be refreshed and reloaded. Today, thanks to some techniques collectively known as Ajax, we can incrementally update parts of web pages without interrupting a user's experience. You've probably seen this on Google when it gives you search suggestions as you type. Any time you see a portion of a web page update without the entire web page refreshing, that's Ajax at work.

Key Terms

- Ajax
- JSON
- asynchronous programming
- XMLHttpRequest



Ajax

Ajax, short for asynchronous JavaScript and XML, is an approach to using existing technologies to make dynamic HTTP requests. Web applications usually work like this: a request is made from a browser to get information from a server, the server sends the appropriate information, and the browser updates the user's page. However, with Ajax, there is an intermediary between the browser and the server known as the Ajax engine; the visual presentation of a website (HTML/CSS) is separated from the handling of HTTP requests. By using an Ajax engine (written in JavaScript) to handle HTTP requests, the actual HTML and CSS page displayed on the browser is upheld while the request is being made. After the information from the request has reached the browser, the Ajax engine can directly update just a portion of the page's HTML or CSS, preventing the entire page from reloading.

Because the server is no longer needs to send entire HTML/ CSS data, Ajax uses simple notations to send small packets of information. XML, or Extended Markup Language, is a notation used to send information. However, most programmers today use **JSON**, or JavaScript Object Notation, instead because it is native to JavaScript.

All of these technologies work together to enable asynchronous programming. Synchronous programming is when you have to wait for previous parts of your program to finish running before moving onto another task. However, in web programming, the web application has to be listening for many possible events at once and then act accordingly. **Asynchronous programming** allows programs to handle multiple tasks at once. Using asynchronous programming with JavaScript means web applications can wait and respond to any number of events, reducing application response time and improving user experience.

XMLHttpRequest and jQuery

The programming techniques that make up Ajax are wrapped up in a native Javascript object known as the **XMLHttpRequest** (XHR) object. This object contains the functions to control the Ajax engine and perform asynchronous HTTP requests. The XHR object allows functions to be dependent upon what state a request is in, enabling seamless experiences for users of web applications. Many programmers use XHR indirectly through jQuery because the syntax is more convenient. Thanks to programmers before us, all you have to do to take advantage of Ajax in your projects is read through the jQuery and XHR object documentation.

CS50

Algorithms

Overview

Recall that computing involves taking some form of input, and then processing that input in order to produce some form of output. Processing input will often require the use of an **algorithm**, which are sequences of instructions that can be executed by a computer. In computer science, these algorithms are usually written in code. Computers depend upon such algorithms in order to perform tasks. In many cases, multiple different algorithms can be used to achieve the same result. However, in some cases, one algorithm will be faster than another at arriving at the correct result.

Key Terms

- algorithm
- correctness
- efficiency
- loops
- conditions

```
 1 pick up phone book
 2 open to first page of phone book
 3 look at names
 4 if “Smith” is among names
 5      call Mike
 6 else if not at end of book
 7      flip to next page
 8      go to line 3
 9 else
10      give up
```

```
 1 pick up phone book
 2 open to middle of phone book
 3 look at names
 4 if “Smith” is among names
 5     call Mike
 6 else if “Smith” is earlier in book
 7     open to middle of left half of book
 8     go to line 3
 9 else if “Smith” is later in book
10     open to middle of right half of book
11     go to line 3
12 else
13     give up
```

A Correct Algorithm

Algorithms are just sequences of steps that a computer can follow in order to translate input into output. Algorithms can be expressed in English as a detailed list of steps.

Take, for example, the task of finding a name (e.g. “Mike Smith”) in a phone book. One possible algorithm (represented to the left) involves picking up the phone book, opening to the first page, and checking to see if Mike Smith is on the page. If he’s not, flip to the next page, and check that page. Keep repeating this until you either find Mike Smith or get to the end of the book.

This algorithm is **correct** — if Mike Smith is in the phone book, then this algorithm will succesfully allow someone to find him. However, algorithms can be evaluated not only on their correctness but also on their **efficiency**: a measure of how well an algorithm minimizes the time and effort needed to complete it. The one-page-at-a-time algorithm is correct, but not the most efficient.

We could improve the algorithm by flipping two pages at a time instead of one — though we’d have to be careful of the case where we might skip over the page with Mike Smith’s name, at which point we’d have to go back a page. But even this algorithm is not the most efficient.

An Efficient Algorithm

Consider instead what might be a more intuitive, and efficient, algorithm. First, open to the middle of the phone book. If Mike Smith is on the page, then our algorithm is done. Otherwise, taking advantage of the fact that the phone book is sorted, we know which half of the book Mike Smith will be in, if he’s in the book at all. Thus, we can eliminate half of the book, and now repeat our algorithm using a book that’s effectively half the size. We can repeat this process until we arrive at a single page, which either does or does not have Mike Smith’s name on it. This algorithm is pictured above (below our original algorithm).

This algorithm is more efficient than the original algorithm. Consider what would happen if a 500 page phone book were to double in size the next year. Using the original algorithm, it might take 500 more steps to go through 500 more pages. However, using the second algorithm, it would only take 1 additional step when searching through a phone book that’s twice the size.

Note that our algorithms have made use of several programming constructs, including **loops**, which repeat steps multiple times, and **conditions**, which only executes certain steps if some statement is true.

CS50 Arrays and Strings

Overview

Recall that variables are used to store values. Quite frequently, we may want to use multiple variables to store a sequence of values: like a sequence of 10 test scores, or 50 addresses. For situations like these, C has a data structure called an **array**: which stores multiple values of the same type of data. For instance, an array of `int`s would store multiple `int` values back-to-back. The **string** type that you have been using is really just an `array` of `char`s.

Key Terms

- array
- string
- size
- index
- null-terminator

```
1   int ages[5];
```

0	1	2	3	4

```
2   ages[0] = 28;
```

0	1	2	3	4
28				

```
3   ages[1] = 15;
4   ages[2] = ages[1];
5   ages[3] = ages[1] - 1;
6   ages[4] = 17;
```

0	1	2	3	4
28	15	15	14	17

```
7   for (int i = 0; i < 5; i++)
8   {
9       ages[i] += 1;
10  }
```

0	1	2	3	4
29	16	16	15	18

Arrays

Like variables, arrays are declared by first stating the type of the data to be stored, followed by the name of the array. In brackets after the name of the array is the **size** of the array: which defines how many values the array will hold. For example, line 1 at left declares an array of 5 `int`s.

You can visualize an array as a sequence of boxes, each one holding a value, and each one with a numbered **index**, which is a number that can be used to access a specific value in an array. In C, arrays are zero-indexed, meaning that the first item in an array has index `0`, the second item has index `1`, etc.

To access a particular value in an array, use the name of the array, followed by the desired index in brackets. Line 2 at left sets the value of the first item in the `ages` array (the one at index `0`) to 28.

The value at each array index can be treated like a normal variable. For example, you can change its value, apply arithmetic or assignment operators to it.

Since each value in an array is referenced by its index number, it's easy to loop through an array. Lines 7 through 10 define up a `for` loop, which iterates through the entire array, and increases each age value by 1.

Strings

In C, a `string` is represented as an array of `char` values. Thus, when we write a line like `string s = "CS50";`, this information is stored as an array of `char`s, with one character at each index. The final index of a string in C is the **null-terminator**, represented by `'\0'`. The null-terminator is the character that tells a `string` that the `string` is over, and that there are no more characters in the `string`.

```
string s = "CS50";
```

0	1	2	3	4
'C'	'S'	'5'	'0'	'\0'

Since a `string` is just an array, you can index into the `string` just like you would index into any other array in order to access the value of a particular character. For instance, in the example above, indexing into `s[0]` would give you the character `'C'`, the first character in the string `"CS50"`.

This also makes it very easy to use a loop to interate through a string and perform computation on each individual character within a string, by first initializing the loop counter to 0, and repeating until the last index of the string. The function `strlen()` takes in a string as input, and returns the length of the string as an integer, which may help in determining how many times the loop should repeat.

CS50 AI and Machine Learning

Overview

From self-driving cars to virtual assistants like Siri and Alexa, artificial intelligence (AI) is becoming more and more common in our everyday lives. The field of AI is centered around the development of **machine intelligence**, or the ability of machines to think and react like humans. But why would we want to program computers and robots to think like us? Well, it turns out that sometimes they can make decisions and answer questions even better than we can. AI has already had a major impact on many parts of our world, including banking, technology, marketing, entertainment, and healthcare, and will likely soon change many more!

Key Terms

- machine intelligence
- Turing test
- machine learning
- training

Turing Test

Computer science pioneer and mathematician, Alan Turing, also played an important role in the field of AI. In his 1950 paper, "Computing Machinery and Intelligence," Turing asked the monumental and ever-relevant question, "Can machines think?" This sparked many conversations that influenced the field. Furthermore, he proposed an "imitation game," now known as the **Turing test**, to determine whether a machine exhibits human-like intelligent behavior to the point where the two are indistinguishable. It works by having a "judge" guess, based only on their answers to the same set of questions, which is the computer and which the human.

The test's efficacy has been widely debated. In recent years, many scientists have claimed to have passed the test with their programs, while still others hold that no programs have been able to do this so far. This is a pretty incredible feat since the first artificial intelligence program, the Logic Theorist, was presented at a conference in 1956!

Machine Learning

One method for achieving artificial intelligence is **machine learning**, a term which was coined in 1959 and is generally defined as the techniques we use to have programs learn from data. There are many applications for these programs, including computer vision and natural language processing. And with the rise of big data, this method is becoming ever more relevant.

Typically, the first step in machine learning is collecting and validating data. Then, a model is set up and trained with part of the data. This **training**, or learning, can be supervised, which means the data set is labeled, or unsupervised, where the data is unlabeled. After training, the model can be further tested and improved, and its accuracy can be measured. Finally, we can use our model to make judgements or predictions about other data.

Deep learning is a subset of machine learning which uses artificial neural networks (ANNs). This system of connected nodes is loosely based on the biological neural networks of animal brains. By using ANNs, some scientists think we can program machines to think more like us.

Future of AI

The AI that exists today is known as weak AI, or that which is designed to complete a specific task like classifying images or driving. The term strong AI refers to the point at which machines can think for themselves. We don't know when, if ever, we will reach the point where we have superintelligent machines. Still, the media is full of depictions of strong AI gone terrible.

In the meantime, we need to grapple with the issues emerging from the rise of AI. Consider, for instance, that we tell a self-driving car to get from point A to point B. It successfully completes this task but it may not be able to make ethical decisions like if it had to pick between hitting three pedestrians or one pedestrian (commonly known as the trolley problem). The machine did what we asked it to do, but not without serious consequences. Who should be held accountable?

Some scholars worry about superintelligence with goals that are not aligned with ours. But will artificially intelligent computers take over the world? We'll soon find out!

CS50

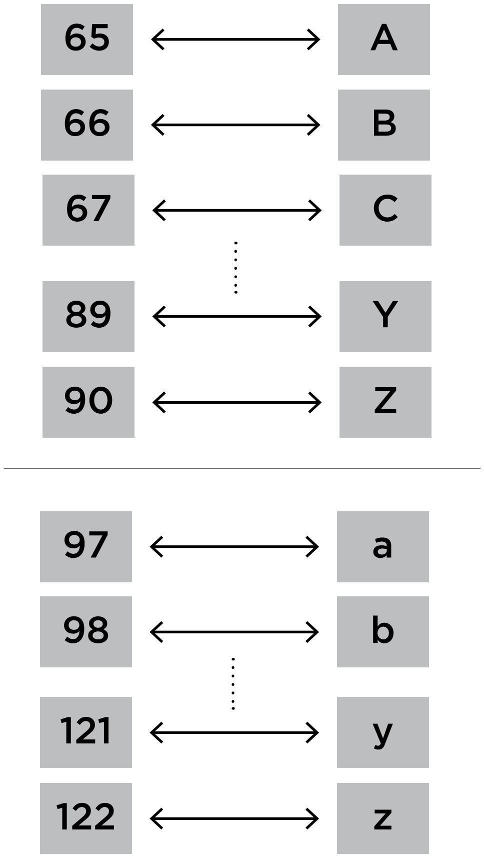
ASCII

Overview

Computers need a way of storing a variety of types of information, including text. However, since computers can only store data as 0s and 1s, computers need a way of using those 0s and 1s to represent characters in text. ASCII is a standard way of translating characters to and from sequences of binary digits that computers can understand.

Key Terms

- encoding
- ASCII
- ASCII table



ASCII Encoding Standard

In order to represent characters as numbers, a character **encoding** standard is used, which gives common characters a unique number to identify them. **ASCII** is a common encoding standard, which computers use in order to store text-based data. In the standard, the number 65 corresponds to the capital letter 'A'. Thus, if a computer wanted to store the capital letter 'A', it would need to store the number 65 in binary (which happens to be 1000001). The next 25 values in the ASCII encoding standard represent the other 25 letters, in order: so 66 represents 'B', 67 represents 'C', and so on.

Lowercase letters also have numerical representations in ASCII. The lowercase letter 'a' is represented by the number 97, 'b' is represented by 98, and so on. Thus, for a computer to store the lowercase letter 'a', it would need to store the number 97 in binary, which is 1100001. Note that this binary number differs from the binary representation of capital 'A' by just one bit: the value in the 32s place. This is because, in ASCII, lowercase letters are always represented as numbers 32 greater than their respective uppercase letters. As a result, letters can easily be switched from lowercase to uppercase or vice versa just by switching a single bit—the one in the 32s place—to 1 or 0 (1 for lowercase, 0 for uppercase).

There's no reason why ASCII has to use these exactly values: ultimately, the decision as to what number maps to which letter is arbitrary. What's important is that the standard is consistent: any computer can read and understand the numbers the same way.

ASCII's Limits

ASCII is frequently represented on an **ASCII table**: which is just a table that shows all possible ASCII characters, and which numbers correspond to them.

The original ASCII table represents all characters using just 7 bits: which means that there are 2^7, or 128, possible characters that can be represented in ASCII. Several extensions to ASCII exist which add an 8th bit, allowing for a total of 256 possible characters to be represented. Since there are only 52 letters, this means that ASCII has space to represent other types of characters: like punctuation, numbers, and some basic symbols (like the $ sign or the % sign).

However, event with 8-bit ASCII encoding, there are still a lot of characters that can't be represented, because there are more than 256 possible characters. For example, many mathematical symbols and characters in other languages do not fit into the standard ASCII table. As a result, other character encoding standards exist that have far more possible character options: Unicode, for example, is a character encoding standard that allows for more than 1 million possible characters to be represented. The first 128 characters in Unicode are identical to the 128 characters in ASCII, which makes them compatible with one another.

CS50

Binary Search

Overview

There are many different **algorithms** that can used to search through a given array. One option is **linear search**, but it can be a rather lengthy process. Luckily, there is a faster searching algorithm: **binary search**. You might recall that binary search is similar to the process of finding a name in a phonebook. This algorithm's speed can be leaps and bounds better than linear search, but not without a cost: binary search can only be used on data that is already sorted.

Key Terms

- algorithms
- linear search
- binary search
- pseudocode

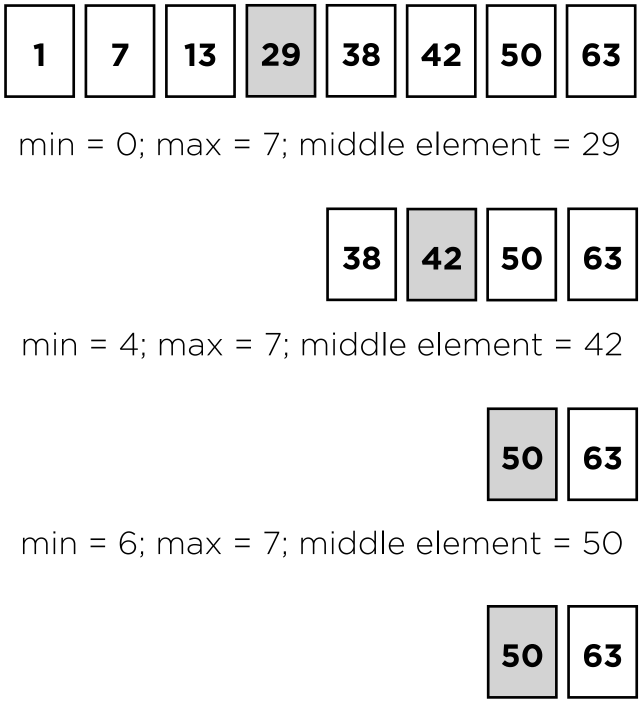
The Binary Search Algorithm

The basis of binary search relies on the fact that the data we're searching is already sorted. Let's take a look at what the binary search algorithm looks like in **pseudocode**. In this example, we'll be looking for an element *k* in a sorted array with *n* elements. Here, `min` and `max` have been defined to be the array indices that we are searching between, marking the upper and lower bounds of our search.

```
1  set min = 0 and max = n - 1
2  find middle of array
3  if k is less than array[middle]
4      set max to middle - 1
5      go to line 2
6  else if k is greater than array[middle]
7      set min to middle + 1
8      go to line 2
9  else if k is equal to array[middle]
10     you found k in the array!
11 else
12     k is not in the array
```

Using the algorithm described above, let's search for the number `50` in the array on the right. First we set the `min = 0` and `max = 7`. Next, calculate the `middle` of the array. Well, how do we decide whether to pick the array index `3` or `4` as the `middle`? It actually does not matter, as long as the algorithm is consistent. For our algorithm, let's choose `3`. We've now determined that the `middle` element is `array[3]`, or `29`. Since `50` is larger than `29` and our array is sorted, we know that `50` will not be on the left of `29`. Therefore, there is no need to check those elements. To continue the search, `min` is set to `4`, `max` remains at `7`, and we repeat the process!

Find the 50!



min = 0; max = 7; middle element = 29

min = 4; max = 7; middle element = 42

min = 6; max = 7; middle element = 50

50 is found!

Binary Search vs. Linear Search

In computer science, it is a common theme that whenever we make some improvement, it is at the cost of another factor. In this case, we trade the speed of a searching algorithm for the time it takes to sort the array. In some cases, it would be faster to just use linear search rather than to sort the data and then use binary search. Nevertheless, binary search is useful if you plan on searching the array multiple times. In case an element does not exist in the array, linear search would iterate through the entire array – however long it may be – to know that the given element does not exist in the array. In binary search, we can be more efficient. Using the algorithm described in our pseudocode, if the searched element is less than the initial value at `min` or larger than the initial value at `max` (the first and last elements of the array, respectively), the algorithm knows the element is not in the array.

CS50

Binary

Overview

Recall that computers represent data in the form of bits, which are just values that can be either 0 and 1. In order to perform mathematical calculations with bits, computers use a number system called **binary**, which is a number system which only uses two digits: 0 and 1.

Key Terms

- binary
- base
- decimal

Decimal System

3	2	8
100s	**10s**	**1s**

(3 x 100) + (2 x 10) + (8 x 1)

300 + 20 + 8
328

Binary System

1	1	0
4s	**2s**	**1s**

(1 x 4) + (1 x 2) + (0 x 1)

4 + 2 + 0
6

Number Systems

Every number system has a **base**, which refers to the number of possible values each digit can take. Most people are used to the **decimal** number system, also known as the base 10 system, where digits can be any value from 0 to 9. In the decimal system, each digit in a number represents a power of 10. The rightmost digit represents the 1s place (which is 10^0). The digit second from the right is the 10s place (or 10^1). The next digit over is the 100s place (or 10^2). To compute the value of a number, just multiply the digit in each place by the value of the place, and add the numbers together.

Binary is a number system with base 2, where digits can only be 0 to 1. In this system, each place value in a number represents a power of 2. The rightmost digit is still the 1s place (which is 2^0). The next digit over is the 2s place (equal to 2^1). The next digit over is the 4s place (equal to 2^2), and it would continue on: with the 8s place, the 16s place, the 32s place, etc. To compute the value of a binary number, just multiply the digit in each place (either 1 or 0) by the value of the place, and add the numbers together. So `110` becomes `1x4 + 1x2 + 0x1 = 4 + 2 + 0 = 6`.

Counting in Binary

Counting in binary is much like counting in decimal, with the restriction that we're only allowed to use two digits: 0 and 1. So `0` translated to binary is still `0`, and `1` translated to binary is still `1`. But since binary doesn't allow the digit 2, in order to represent `2` in binary we need another binary digit. Thus, the number `10` can be used to represent `2`. Since there is a 1 in the 2s place, and a 0 in the 1s place, the value of the number is `2x1 + 0x1 = 2`. If `2` in binary is `10`, then `3` in binary is `11`.

However, to represent the number `4`, we've once again run out of bits. In order to represent the number, a third bit is required, to create a value in the 4s place. `100` therefore is the binary representation of the number `4`.

Mathematics that can be performed in the decimal system can also be performed in binary. Binary numbers can be counted, added, subtracted, multiplied, and divided just like numbers in decimal, and thus can be used by computers in order to execute computations and make calculations.

Decimal to Binary

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

CS50 Boolean Expressions

Overview

Conditions are how programmers can make decisions in programs, by allowing some parts of the code to only run under certain circumstances. Conditions will generally work by evaluating a **boolean expression**, which is an expression that will have a value of either `true` or `false`. Programmers can set conditions such that different code will run depending on what the value of the boolean expression is.

Key Terms

- condition
- boolean expression
- if statement
- switch statement
- ternary operator

```
1   bool a = 3 < 5;

2   bool b = 2 >= 8;

3   bool c = a && b;

4   bool d = a || b;

5   bool e = !d;
```

a	b	c	d	e
true	false	false	true	false

Boolean Operators

Boolean operators are used to create boolean expressions that evaluate to `true` or `false`. Common boolean operators include the comparison operators: `<` (less than), `>` (greater than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to), and `!=` (not equal to). For instance, in line 1 to the left, `a` is set to `true` because the expression `3 < 5` is true (because 3 is in fact less than 5). In line 2, `b` is set to `false` because the expression `2 >= 8` is not true.

Logical operators can also be used to combine boolean expressions. `&&` is the logical AND operator: it will evaluate to `true` if both expressions on either side of it are true. `||` is the logical OR operator: it evaluates to `true` if at least one of the two expressions on either side is true. And `!`, the logical NOT operator, evaluates to the opposite of whatever the expression immediately after it is.

Conditions

Conditional branching refers to the idea that different parts of code should execute under different circumstances. The most common type of conditional is the **if statement**: where a certain block of code (enclosed in brackets) will only run if the condition (whatever is in the parentheses after the word `if`) evaluates to `true`.

Optionally, C also allows you to include an `else` block after an `if` statement, which defines which code should run if the `if` condition evaluates to `false`. C will also allow you to include one or multiple `else if` statement after an `if` statement, to add additional conditions that could run different blocks of code. The if statement to the right (lines 1-12) will print `"positive\n"` if the value of `x` is greater than `0`, `"negative\n"` if the value of `x` is less than `0`, and `"zero\n"` if the value of `x` is equal to `0`.

C also has other ways of expressing conditionals. The **switch statement**, shown to the right (lines 15-25), takes one variable, and defines what code should run based on which `case` the variable matches. In the example at right, if `x` is equal to `1`, `"A\n"` is printed; if x is equal to `2`, `"B\n"` is printed, and in all other cases (the `default` case), `"C\n"` is printed. Code within cases should end with `break` so that the program knows to stop executing code and go to the end of the `switch` statement.

The ternary operator is a third type of condition. The **ternary operator** takes an expression, and evaluates to one value if the expression is true, and another value if it is false. In the example on line 28, if `x > 3`, `y` is set to 2, and `1` otherwise.

```
 1  if (x > 0)
 2  {
 3      printf("positive\n");
 4  }
 5  else if (x < 0)
 6  {
 7      printf("negative\n");
 8  }
 9  else
10  {
11      printf("zero\n");
12  }
13
14
15  switch (x)
16  {
17      case 1:
18          printf("A\n");
19          break;
20      case 2:
21          printf("B\n");
22          break;
23      default:
24          printf("C\n");
25  }
26
27
28  int y = (x > 3) ? 2 : 1;
```

CS50

Bubble Sort

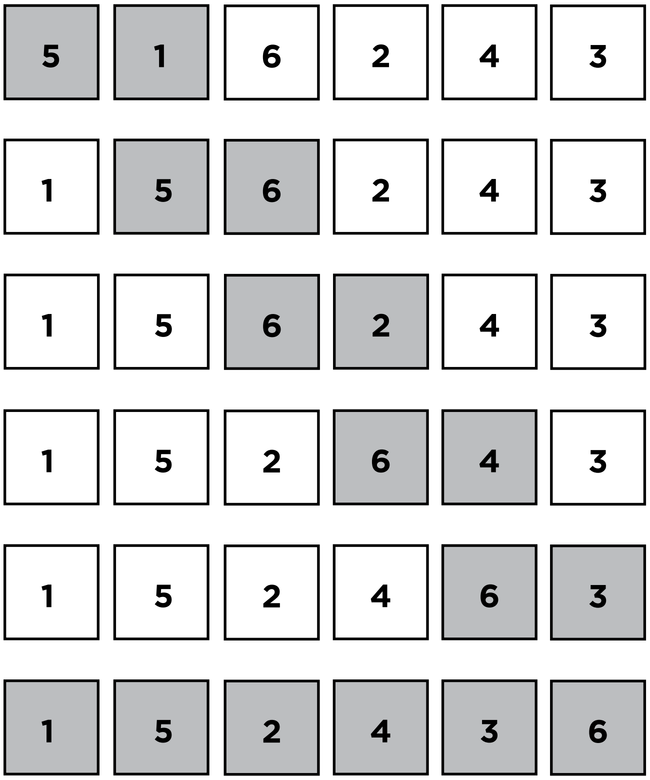
Overview

There are limited ways to search a list that is unsorted. It is often more efficient to sort a list and *then* search it. One of the most basic sorting algorithms is called **bubble sort**. This algorithm gets its name from the way values eventually "bubble" up to their proper position in the sorted array. This basic approach to sorting narrows the scope of our problem to focusing on ordering just two elements at a time, instead of an entire **array** at a time. This approach is very straightforward, but possibly at the expense of making an inordinate number of swaps just to put one single element into position.

Key Terms

- bubble sort
- array
- pseudocode

Step-by-step process of 1 pass through in bubble sort



5 1 6 2 4 3

1 5 6 2 4 3

1 5 6 2 4 3

1 5 2 6 4 3

1 5 2 4 6 3

1 5 2 4 3 6

Implementation

Bubble sort works by comparing two adjacent numbers in a list, and swapping them if they are out of order. Looking at the example on the left, if we are given an array of the numbers 5, 1, 6, 2, 4, and 3 and we wanted to sort it using bubble sort our **pseudocode** for one single pass might look something like this:

```
for every element in the array
        check if element to the right is smaller
                if so swap the two elements
        else move on to the next element in the list
```

When this is implemented on the example array, the program would start at 5 and compare it with 1. Since 1 is smaller than 5 it would swap them. It would then move on to compare 5 and 6 since those are in the correct order, we just move on to the next element. Next 6 and 2 are compared, and so on.

Finally after doing this for all the elements in the array we are left with the array 1, 5, 2, 4, 3, and 6. It's not completely sorted, but notice that after the first passthrough, the 6 is already in its correct location. After n passthroughs the last n elements are in their correct position. This fact can be used to optimize this algorithm since it is not necessary to look at those correctly sorted elements. It is this effect of the larger elements "bubbling" to the right side that gives this algorithm its name!

Sorted Arrays

If bubble sort was implemented only as above, we would only go through one passthrough, but as the example shows, it is not guaranteed that the array will be sorted after one pass. So how many times should this algorithm be run? Well in the worst case scenario, a reverse sorted list (6, 5, 4, 3, 2, 1), it might need to run 5 times. Indeed the same would hold true for *n* elements, the algorithm might need to run *n-1* times. That seems wasteful though, since it would only need to run that maximum number of times if the array is a "worst case scenario" (more on that in the time complexity module).

How can you ensure you only run this algorithm the necessary amount of times, maybe saving a few steps? Well, if this algorithm is run and no swaps are made, it must be true that the array is sorted (think about it)! Maybe then it would make sense to amend our implementation to include a counter for the amount of swaps made. If `counter == 0`, then the array is sorted, however if `counter > 0`, then more passthroughs are needed to sort the array. Now we only decide at the end of every passthrough whether more passthroughs are necessary!

CS50 Bugs and Debugging

Overview

A **bug** is an error in code which results in a program either failing, or exhibiting a behavior that is different from what the programmer expects. Bugs can be frustrating to deal with, but every programmer encounters them. **Debugging** is the process of trying to identify and fix bugs that exist in code. Programmers will often do this by making use of a program called a **debugger**, which assists in the debugging process.

Key Terms

- bugs
- debugging
- debugger
- breakpoint
- `debug50`

Debugging Basics

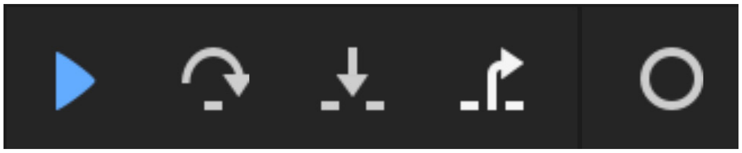
Programs generally perform computations much faster than a human possibly could, which makes it difficult to see what's wrong with a program just by running it all the way through. Debuggers are valuable because they allow a programmer to freeze a program at a particular line, known as a **breakpoint**, so that the programmer can see what's happening at that point in time. It also allows the programmer to execute the program one line at a time, so that the programmer can follow along with every decision that a program makes.

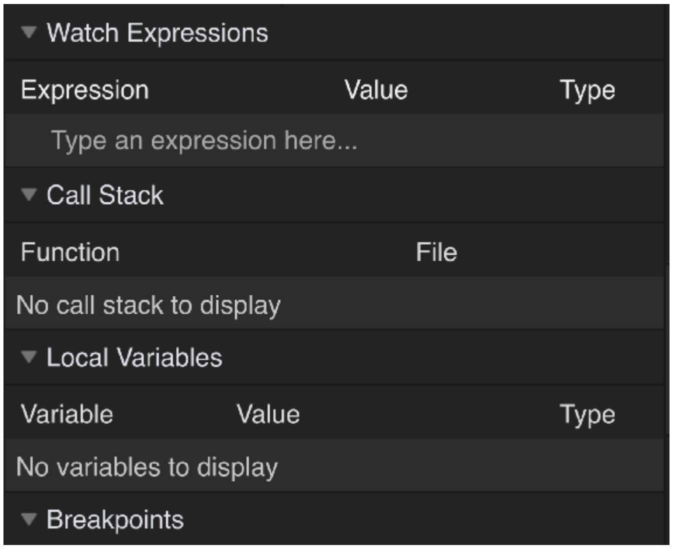
Using `debug50`

`debug50` is a program that we've created that runs a built-in graphical debugger in the CS50 IDE. Before you run `debug50`, you must set at least one breakpoint. If you have a general sense for where your program seems to be going wrong, it may be wise to set a breakpoint a few lines before there, so that you can see what's happening in your program as it moves into the section that you believe is causing the problem. If you're not sure at all, then it's totally fine to set a breakpoint at the first line of your `main` function so that you can step through the entire code from the beginning. To set a breakpoint, click the space to the left of the line number of your program. You will see a red dot appear in that space and it will be added to the list of all your breakpoints at the bottom of the graphical debugger window. You can remove a breakpoint by clicking on the red dot next to the line number.

Once you're satisfied with your breakpoints, run your program with as usual with `debug50 ./program_name` and any command-line arguments. Your program will run, and automatically stop at any breakpoints. In the debugger window, you'll see several tabs you can interact with.

At the top of the window, you'll see five buttons. The first, the blue triangle, will run your code until it hits the next breakpoint. The next button, the curved arrow, allows you to skip over a block of code. Next to that is the down arrow which allows to move through your code slowly one line at a time. The last arrow allows you to step out of a function (other than `main`). The last icon, the outline of a circle, clears all of the breakpoints that you set in your program.





▾ Watch Expressions

Expression Value Type

Type an expression here...

▾ Call Stack

Function File

No call stack to display

▾ Local Variables

Variable Value Type

No variables to display

▾ Breakpoints

The window at left summarizes some some features of `debug50`. In the "Watch Expressions" tab, you can type in a variable or function you want to watch while your code runs.

The "Call Stack" tab displays what function the line of code you are on is in and the file path for the programming that is running.

The "Local Variables" and "Breakpoints" are pretty intuitive. For each variable, you can see it's name, value and type. You can even override variable values by clicking on the value and typing in a new value. For each breakpoint you've marked, you'll see the file name, line number, and what appears on that line of code. You can turn off breakpoints from this tab by clicking on the checkbox next to the breakpoint, or delete it by clicking on the 'x' in the top right corner of the breakpoint when you hover over it.

CS50 Command-Line Interaction

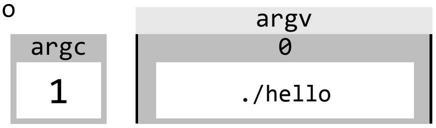
Overview

When running a program from the command line, you've generally executed a command like `./program_name` at the command line. C also allows you to specify a program's **command-line arguments**, which allows the person running the program to pass arguments into the `main` function of the program by specifying the arguments at the command line. This offers an alternative means of providing input to a program beyond just requesting input while a program is running, such as with `get_string()`.

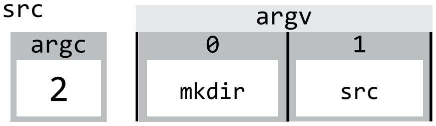
Key Terms

- command-line arguments
- argument count
- argument vector

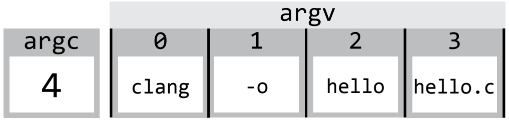
`./hello`



`mkdir src`



`clang -o hello hello.c`



argc, argv

Many of the command-line programs that you have likely called before (`make`, `cd`, `clang`, `mkdir`) all take command-line arguments. In C, command-line arguments are passed into the main function as inputs. However, we've previously written our `main` functions to take no arguments (`void`).

To accept command-line arguments, we can revise the `main` function to take two arguments: `argc`, an integer, and `argv`, an array of `string`s.

`argc`, which stands for "**argument count**", represents the number of arguments passed into through the command line. Each word (separated by spaces) counts as its own argument, and the calling of the program itself (e.g. `./hello`) counts as an argument.

`argv`, which stands for "**argument vector**", is the actual array representing the arguments themselves. Each value in the array is a `string`.

If you were to look at `argc` and `argv` when calling a program with no arguments, like calling `./hello`, `argc` would be 1 (because the calling of the program is the only argument). `argv`, on the other hand, would be an array consisting of just one element: the string `"./hello"` stored at index `0`.

If you were to look at `argc` and `argv` when calling a program that does have arguments, like calling `mkdir src`, `argc` would be 2, since two arguments are passed in via the command line, and `argv` would be an array with two elements: the string `"mkdir"` stored at index `0`, and the string `"src"` stored at index `1`.

Using Command Line Arguments

Shown to the right is an example of a program which accepts command-line arguments. Notice on line 4 that the definition of the `main` function has changed to include the arguments `argc` and `argv`. No size of `argv` is specified on line 4, so that any array, regardless of its size, can be passed into the `main` function.

Inside of `main` function, the program loops through the array, starting at index `0`, and incrementing so long as `i < argc`. It's important to stop there, because the largest index of `argv` that you can access is `argc - 1` (since arrays are zero-indexed). During each iteration, the program prints out the value of `argv` at index `i`.

The result of the program is that each of the program's command line arguments is printed on a new line.

```
 1  #include <cs50.h>
 2  #include <stdio.h>
 3
 4  int main(int argc, string argv[])
 5  {
 6      for (int i = 0; i < argc; i++)
 7      {
 8          printf("%s\n", argv[i]);
 9      }
10  }
```

CS50

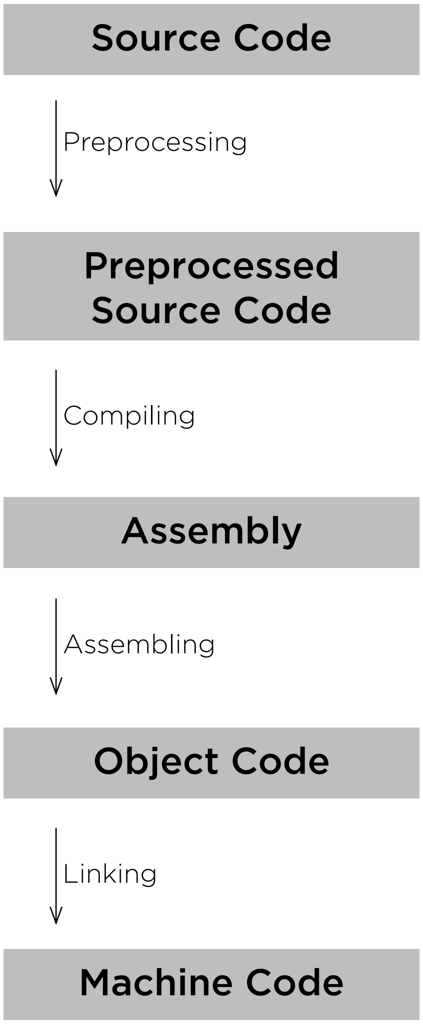
Compiling

Overview

Compiling is the process of translating source code, which is the code that you write in a programming language like C, and translating it into **machine code**: the sequence of 0s and 1s that a computer's central processing unit (CPU) can understand as instructions for how to execute the program. Although the command `make` is used to compile code, `make` itself is not a compiler. Instead, `make` calls upon the underlying compiler `clang` in order to compile C source code into object code.

Key Terms

- compiling
- machine code
- preprocessing
- assembly
- object code
- linking



Preprocessing

The entire compilation process can be broken down into four steps. The first step is **preprocessing**, performed by a program called the preprocessor. Any source code in C that begins with a `#` is a signal to the preprocessor to perform some action.

For example, `#include` tells the preprocessor to literally include the contents of a different file in the preprocessed file. When a program includes a line like `#include <stdio.h>` in the source code, the preprocessor generates a new file (still in C, and still considered source code), but with the `#include` line replaced by the entire contents of `stdio.h`.

Compiling

After the preprocessor produces preprocessed source code, the next step is to compile (using a program called a compiler) C code into a lower-level programming language known as **assembly**.

Assembly has far fewer different types of operations than C does, but by using them in conjunction, can still perform the same tasks that C can. By translating C code into assembly code, the compiler takes a program and brings it much closer to a language that a computer can actually understand. The term "compiling" can refer to the entire process of translating source code to object code, but it can also be used to refer to this specific step of the compilation process.

Assembling

Once source code has been translated into assembly code, the next step is to turn the assembly code into object code. This translation is done with a program called the assembler.

Object code is essentially machine code with some non-machine code symbols. If there's only one file that needs to be compiled from source code to machine code, the compilation process is over now. However, if there are multiple files to be compiled, a file's object code only represents part of the program and an additional step is required. The object code file's non-machine code symbols denote how the file fits with the other parts of the program. The entire program is put together in a process called linking.

Linking

If a program has multiple files that need to be combined into a single machine code file (such as if a program includes multiple files or libraries like `math.h` or `cs50.h`), then one final step is required in the compilation process: **linking**. The linker takes multiple different object code files, and combines them into a single machine code file that can be executed. For example, linking the CS50 Library during compilation is how the resulting object code knows how to execute functions like `get_int()` or `get_string()`. It is important to note that only one file can have a `main` function so that the program knows where to begin.

CS50 Computational Complexity

Overview

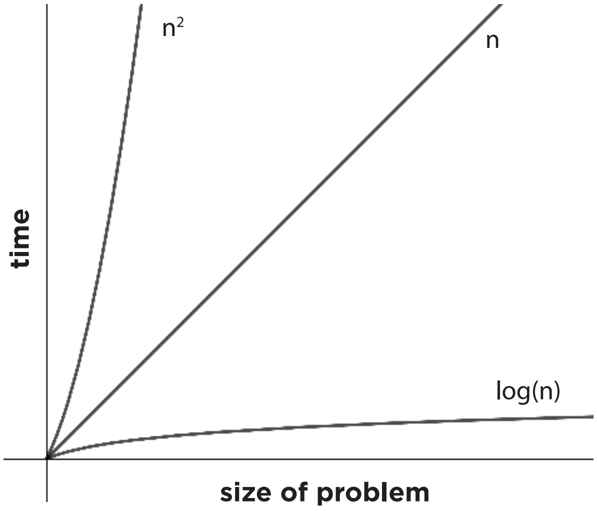
The subject of computational complexity (also known as time complexity and/or space complexity) is one of the most math-heavy topics, but also perhaps one of the most fundamentally important in the real-world. As we begin to write programs that process larger and larger sets of data, analyzing those data sets systematically, it becomes increasingly important to understand exactly what effect those algorithms have in terms of taxing our computers. How much time do they take to process? How much RAM do they consume? One aspect of **computational complexity** is the amount of time an algorithm takes to run, in particular considering the theoretical worst case and best case scenarios when running programs.

Key Terms

- computational complexity
- Big O
- Big Ω

Computational Complexity Notation

Big O notation, shorthand for "on the order of", is used to denote the worst case efficiency of algorithms. Big O notation takes the leading term of an algorithm's expression for a worst case scenario (in terms of *n*) without the coefficient. For example, for linear search of an array of size *n*, the worst case is that the desired element is at the end of the list, taking *n* steps to get there. Using Big O notation, we'd say linear search runs in *O(n)* time. We can calculate the computational complexity of bubble sort in the same way, albeit with a little more math. Remember that bubble sort involved comparing things by pairs. In a list of length *n*, *n - 1* pairs were compared. For example, if we have an array of size 6, we would have to compare `array[0]` and `array[1]`, then `array[1]` and `array[2]`, and so on until `array[4]` and `array[5]`. That's 5 pairs for an array of size 6. Bubble sort ensures that after *k* passthroughs of the array, the last *k* elements will be in the correct location. So in the first passthrough there are *n-1* pairs to compare, then on the next passthrough only *n-2* comparisons and so forth until there is only 1 pair to be compared. In math *(n-1) + (n-2) + ... + 1* can be simplified to *n(n-1)/2* which can be simplified even further to $n^2/2 - n/2$. Looking at the expression $n^2/2 - n/2$, the leading term would be $n^2/2$, which is the same as $(1/2)\ n^2$. Getting rid of the coefficient, we are left with n^2. Therefore, in the worst case scenario, bubble sort is on the order of n^2, which can be expressed as $O(n^2)$. Similar to big O, we have **big** Ω (omega) notation. Big Ω refers to the best case scenario. In linear search, the best case would be that the desired element is the first in the array. Because the time needed to find the element does not depend on the size of the array, we can say the operation happens in constant time. In other words, linear search is *Ω(1)*. In bubble sort, the best case scenario is an already sorted array. Since bubble sort only knows that a list is sorted if no swaps are made, this would still require *n-1* comparisons. Again, since we only use the leading term without the coefficients, we would say bubble sort is *Ω(n)*.



Comparing Algorithms

Big O and big Ω can be thought of as upper and lower bounds, respectively, on the run time of any given algorithm. It is now clear to see which algorithms might be better to use given a certain situation. For instance, if a list is sorted or nearly sorted, it would not make sense to implement a selection sort algorithm since in the best case, it is still on the order of n^2, which is same as it's worst case run time. Binary search may seem to be the fastest search but it is clear to see that searching a list once with linear search is more efficient for a one time search, since binary search run requires a sort algorithm first, so it could take $O(log(n)) + O(n^2)$ to search a list using binary if the list is not already sorted.

Algorithm	Big O	Big Ω
linear search	O(n)	Ω(1)
binary search	O(log(n))	Ω(1)
bubble sort	$O(n^2)$	Ω(n)
insertion sort	$O(n^2)$	Ω(n)
selection sort	$O(n^2)$	$Ω(n^2)$

CS50 Computers and Computing

Overview

A **computer**, in the most general sense, is just a device that accepts data or input, and processes it in some way to automatically produce a result. When a computer is doing any kind of work, whether it's opening an application, editing an image, or playing a song, it is computing. **Computing**, in the most general sense, means calculating. In order for a computer to operate correctly, many different parts of the computer have to communicate and interact with one another in just the right way.

Key Terms

- computer
- computing
- input
- output
- algorithm
- programming
- computational process
- hardware
- software
- operating system
- CPU

Inputs and Outputs

A computer starts by taking in some sort of data or information, called input. **Input** can take a variety of forms—mouse clicks, keyboard presses, taps on a touch screen, or button presses, for instance. Input can also take less traditional forms: such as the way smoke detectors take in information from the environment, or the way cars take input from a steering wheel in order to determine which way to turn.

Computers use the inputs provided to them in order to generate a result. In computer science, this result is called the **output**. In the case of a traditional desktop computer, output might take the form of whatever is displayed on the user's screen. But output can take many other forms, such as producing sound or causing motion.

Somehow, computers need to translate inputs into outputs, by processing the information in the input in order to generate the necessary output. This processing takes the form of an **algorithm**, which is just a set of rules that a computer must follow in order to translate inputs into the desired outputs. **Programming** is the process of providing a computer with a set of instructions, or an algorithm, in order to perform a particular task.

The Computational Process

The process of translating inputs into outputs is known as the **computational process**, and will likely involve performing a series of calculations in the form of an algorithm.

The computational process can range in its complexity and in the number of steps required to complete a task. Sometimes, the computational process is relatively simple, like the process of calculating 5+3. In other cases, many computational tasks are much more complicated. All tasks computers perform, like a GPS calculating a route from home to work, or an alarm going off at a certain time, require computation.

How Computers Work

Each part of the computer serves a specific function, and together they allow for computers to perform tasks. Computers require a combination of **hardware**, the physical parts that make up the computer, and **software**, the programs and instructions that run on the computer. Much of a computer's hardware is attached to the computer's motherboard (or logic board), which contains the hardware that helps different parts of the computer communicate with each other.

Computers require electricity to function, so they must have a power supply— desktop computers plug into an electrical outlet, and laptop computers can use a battery. When the power button on a computer is pressed, the power supply begins providing electricity to the computer, which begins the process of starting up the computer's hardware.

After the computer's hardware has started up, the next step is getting the computer's software ready, beginning with the **operating system**, which is the software that manages the execution of other programs on the machine (common operating systems include Windows, macOS, and Linux). The operating system, as well as other software and computer files, are stored on the computer's hard drive, which is the computer's primary form of storage. Each computer also has a Central Processing Unit (**CPU**), often referred to as the "processor," which is responsible for running the computer's software and executing computations.

CS50

CPU and SoC

Overview

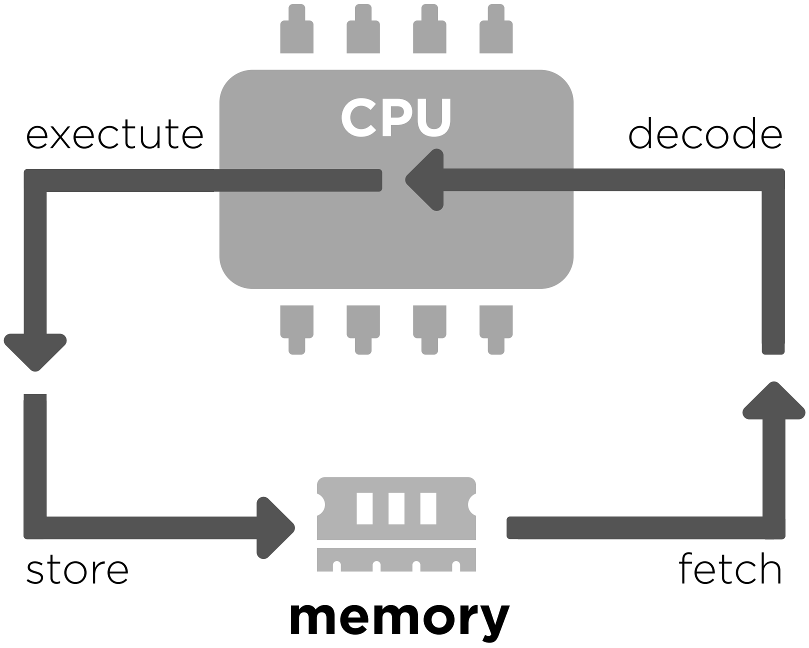
One key component of any computer is a CPU. A **CPU** (or processor) is the brains of your computer; it runs applications from calculators, web browsers, to video games. However, a computer is more than just a CPU. CPUs handle the calculations that allow us to run code, but they must work without other hardware components such as memory and graphic processing units (GPUs) to function as the computers we know today.

Key Terms

- CPU
- core
- hyperthreading
- motherboard
- SoC

CPUs

A CPU consists of billions of microscopic transistors that together handle instructions in the form of 0s and 1s. These instructions are prescribed by humans in the form of code, and so by writing code, we can tell a computer to do whatever we want it to do. A CPU's tasks can be broken into four main steps: fetch, decode, execute, and store. Fetching is the process of getting instructions from some location in memory, decoding is translating those instructions into something the CPU can directly understand, executing is actually following the given instructions, and storing is saving the result of the execution somewhere for later access. CPUs go through this overall process many, many times. CPUs are organized into **cores**, or parts of the CPU that can independently process instructions. Early CPUs only had one core, however, today's computers often have multiple; you'll often find computers that advertise dual and quad-core processors. Each core can only run one program at a time. For example, if you only have one washing machine, you can only run one load of laundry at a time. If you have multiple loads of laundry, you would have to wait until the previous load is finished before washing the next. Having multiple cores is like having multiple washing machines. By leveraging multiple cores within a CPU, computers are able to multitask and run multiple things at once. In addition to having multiple physical cores, CPUs can utilize a technology called hyperthreading. **Hyperthreading** allows a single physical core to act as two individual cores. These new cores are known as virtual cores because they aren't actually cores, but computer software treats them as if they are. Virtual cores can be used to speed up programs, however they are not as advantageous as physical cores.



SoC

CPUs do not work alone. In order for CPUs to work with other things, CPUs are usually connected to a **motherboard**, the main circuit board that connects all of a computer's hardware components. However, motherboards with its hardware components can be quite large. While modern desktops and laptops still have CPUs connected to a larger motherboard, smaller devices like smartphones and tablets take advantage of something called system on a chip (**SoC**). A SoC is exactly what it sounds like: it is an entire system on a single chip. In a SoC, the CPU is fully integrated with memory, GPUs, and more on a single chip. You can think of it as fully functional computer in a tiny, tiny box. SoCs are significantly smaller and require less power than traditional CPUs, but there is a tradeoff. By nature of having a tightly integrated unit, SoCs are inflexible and cannot be customized or upgraded. In other words, if part of the SoC breaks, the whole things breaks, and there is no way to improve or replace any of its components.

CS50

CSS

Overview

Cascading Style Sheets (**CSS**) is a language used on the Internet to style web pages. While HTML describes the structure of a web page, CSS determines text alignment, the size of various elements, the color of various elements, and how HTML elements appear as a window is resized, amongst other things. There are also several different ways of incorporating CSS into a web page.

Key Terms

- CSS
- attribute-value
- ID
- class

The Style Attribute

CSS can be included directly into HTML using the style HTML attribute in any HTML tag. CSS takes the format of **attribute-value** pairs, where each CSS attribute is followed by a colon, followed by its value (multiple attribute-value pairs can be separated by semicolons). In the example here, we've included CSS directly in this `<p>` tag. The CSS attribute font-size is set to `28px`. As a result, when the HTML is displayed in a web browser, the paragraph will appear in 28-point font.

```
<p style="font-size:28px;">
    This is a paragraph.
</p>
```

Note that there is a distinction between HTML attributes and CSS attributes. **`style`** is the name of an HTML attribute, while **`font-size`** is an example of a CSS attribute. There are many different CSS tags. Common ones for styling text include: **`color`**, which sets the color of the text; **`text-align`**, which sets the text alignment (centered or left-aligned, for example); and **`font`**, which sets the font for the text.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page</title>
    <style>
        p
        {
           font-size: 28px;
        }
    </style>
  </head>
  <body>
    <p>This is a paragraph.</p>
  </body>
</html>
```

The Style Tag

CSS can also be located inside of a **`style`** element, usually located in the head section of an HTML document. Within the **`style`** tags, we first need to specify what the styling should apply to. This could be the name of a type of element (e.g. **`p`**), or it could be an ID or class of an HTML element. When determining whether to use an ID or a class, you can think of how we use the words in an everyday context. An **ID** is typically unique to a user so it should be used to style only one element. A **class** meanwhile is a group of students so it should be used to style multiple elements that have something in common. To apply styling to an ID, the ID should be referenced with a **#** symbol followed by its name. To apply styling to a class, the ID should be referenced with a **.** followed by the name of that class.

After specifying what their styling should apply to, CSS attribute-value pairs can be included within curly braces, separated by semicolons. In the example at left, the CSS specifies that all **`p`** elements should have font size **`28`**. Styling CSS in this manner can be advantageous if the styling applies to multiple different HTML elements, since then we don't repeat the same CSS.

Factored CSS

A third way to style with CSS is to store all the CSS in an entirely separate file. If the all of the CSS that would be inside of **`<style>`** tags is stored in a document (typically one that ends with **`.css`**), then that file can just be imported into an HTML document. We can do this by including a **`<link>`** tag in the head element of the HTML document. For instance, by including a line like **`<link href="style.css" rel="stylesheet" />`**, we can add an external CSS document to the HTML document and know that the appropriate styling will appear.

By putting all of the overlapping CSS code into one file, we remove unneeded repetition and redundancy from our HTML files. We also make changing the styling of multiple HTML documents more efficient since we would only need to update the one CSS document and then those HTML documents would be automatically re-styled. In these ways, factoring out CSS into a separate document can be particularly advantageous when dealing with multiple different HTML documents that use the same styling.

CS50 Cybersecurity

Overview

The Internet is full of security threats. One significant threat is the threat of **cyberattacks**, where hackers attempt to target computer systems and networks for malicious purposes. **Cybersecurity** refers to systems and practices that web sites and users can employ in order to better protect themselves against cyber threats. Users can help to protect themselves against cyber threats through a variety of means, including choosing more secure passwords and being mindful of spam email.

Key Terms

- cyberattacks
- cybersecurity
- phishing
- two-factor authentication
- SSL
- TLS

Passwords

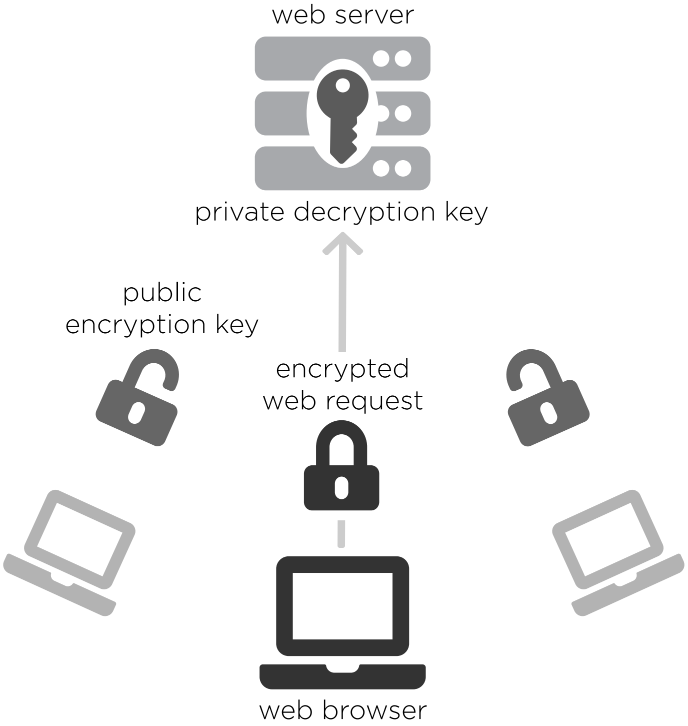
Hackers can attempt to obtain passwords in various ways. One way is to try submitting millions of possible username and password combinations until one is successful. This is why choosing a longer and harder to predict password can improve security. Hackers may also attempt **phishing** attacks, where they send emails to users pretending to be a legitimate company and ask users to click on a link that asks for a password or other sensitive information.

Some services (including Google and Facebook) offer **two-factor authentication** as a means of combating possible password theft. Two-factor authentication requires two types of authentication that are inherently different, one factor would be a username and password, while the other factor can take the form a verification code sent via text to your phone or a security question or SecurID, which was a physical device that would generate a random 6-digit integer. Thus a user needs both their password and a secondary device in order to be able to login successfully. However, there is a tradeoff in convenience: if your phone is lost, or do not have access to your phone then you may be unable to access your account.

Website Security

HTTPS (with the "S" standing for "Secure") is a protocol for communication across the internet that combines HTTP and a technology called Secure Sockets Layer (**SSL**). Websites that use SSL each have a certificate, which is distributed to users who are trying to access the website. The certificate secures the connection between the server and the individual and also contains a public encryption key, which tells web browsers how to encrypt requests that are sent to the web server. The web server has another key, the private key, which can decrypt the encrypted requests. As a result, when a user sends an encrypted request to a web server using SSL, the information is more secure. You can generally tell which websites are using this technology by noting whether or not their URLs begin with `https://`

Today, a technology called Transport Layer Security (**TLS**) is more commonly used, but it is just an updated and improved version of SSL.



Other Cyberattacks

Hackers use several other techniques to perform cyberattacks. In a man-in-the-middle attack, a malicious piece of equipment (like a router or DNS server) in between a user and a web server can replace any occurence of `https://` with `http://` in links and redirects. The result is that an adversary can return pages to a user that look like the correct website, but actually are not.

Session hijacking is another cyberattack technique, wherein an adversary monitors network traffic for cookies, and uses the cookie in the adversary's own HTTP headers, tricking a web server into thinking that the adversary is someone else.

CS50

Data Types

Overview

Unlike many modern programming languages, C is a **statically-typed** language; it requires that every time you declare a variable, that you specify the data type of that variable. Many modern languages are **dynamically-typed**: at runtime, the program figures out the type of all the variables in the program. There are several different primitive (or basic) data types that are built in to C, and several more that are offered by the CS50 Library.

Key Terms

- statically-typed
- dynamically-typed
- native
- qualifiers

Native Types

C's **native** data types are the data types built into the programming language. An `int` is a data type which represents an integer: its value could be a positive or negative whole number, or zero. Numbers like 5, 28, -3, and 0 can be represented as `int`s, but numbers like 2.8, 5.124, and -8.6 cannot. When an `int` is declared, the computer allocates 4 bytes worth of space for it. Since 4 bytes is 32 bits, this means that there are 2^{32} (more than 4 billion) possible integers that can be represented as an `int`: in the range from -2^{31} to (2^{31} - 1).

What if you need to store an integer outside of this range? C also includes **qualifiers**, which are keywords that can be added in front of type names to cause changes to the type. One such qualifier is the `unsigned` qualifier, which designates a type to be not negative. As a result, an `unsigned int`, while still 4 bytes in size, doesn't need to include the negative numbers in its range of possible values. An `unsigned int` can therefore take on a value in the range 0 to 2^{32} - 1.

Another qualifier is the `long` qualifier, which allocates more bytes to the variable, allowing it to store more values. The long long integer (denoted by the type `long long`) is an integer which uses 8 bytes of storage instead of 4, allowing numbers in the range from -2^{63} to (2^{63} - 1).

In addition to `int`s, C also has several other native data types. A `char` is a data type which represents a character of text. A `char` in C is surrounded by single quotation marks. Examples of possible `char` values include lowercase letters like 'a', uppercase letters like 'Z', symbols like the exclamation point '!', or even the newline character `'\n'`, which counts as a single character.

To store numbers that isn't a whole number, C has a type called `float` (short for floating-point), which uses 4 bytes to store a decimal value like 2.8 or 3.14. C also has a native type called `double`, which also stores decimal values but does so using 8 bytes instead of 4.

CS50 Library Types

The CS50 Library makes other types available to you, so long as you remember to type `#include <cs50.h>` at the start of your program. The `bool` type (short for Boolean) stores one of only two values: `true` or `false`.

The CS50 Library also defines a type called `string`, which stores text.

C doesn't limit users to only using the data types built into the programming language. It also offers additional features which allow the programmer to define their own custom types to use in programs.

Data Type	Native?	Sample Values	Size
int	Yes	5, 28, -3, 0	4 bytes
char	Yes	'a', 'Z', '?', '\n'	1 byte
float	Yes	3.14, 0.0, -28.56	4 bytes
double	Yes	3.14, 0.0, -28.56	8 bytes
long long	Yes	5, 28, -3, 0	8 bytes
bool	No	true, false	1 byte
string	No	"Hi", "This is CS50"	4 or 8 bytes

CS50 DNS and DHCP

Overview

There are two important systems in place to make sure that devices on the Internet use IP addresses effectively. The first is the Domain Name System, or **DNS**, which is responsible for converting the words that are typed into an address bar in a web browser like Google Chrome or Safari into the corresponding IP address. The second is the Dynamic Host Configuration Protocol, or **DHCP**, which helps assign each device an IP address.

Key Terms

- DNS
- DHCP
- URL
- domain

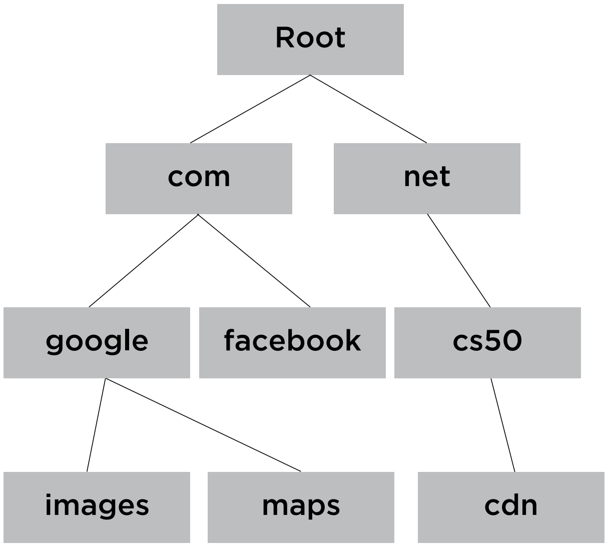
DNS

Most people browsing the Internet don't type an IP address in when they want to access a web page. Instead, they type in a **URL**, a Uniform Resource Locator, which acts as a more human-readable and memorable web address than an IP address.

However, IP still requires the computer to know which IP address it is trying to access. This is where DNS comes in. DNS is responsible for taking the **domain**, which is just an identifier like "google.com" or "facebook.com", and translating it into its respective IP address(es).

When a user types a URL into a web browser, the computer contacts a DNS server, which stores information about which domain names map to which IP addresses. There are many DNS servers, and not all of them will updated at the same time when the mappings between domain names and their IP addresses are changed. As a result, and because it takes time for these changes in the DNS system to propagate throughout all of the DNS servers on the Internet, DNS servers must always communicate with one another about these updates.

DNS Hierarchy



Domains in DNS are organized in a tree-like hierarchy. There are a set of basic "top-level domains" (TLDs), which appear at the ends of many familiar websites. Two hierarchies exist at this level: organizational and geographic. Amongst the top-level organizational domains are com, edu, gov, net, org, among others. The geographic domains are two-letter country codes (such as uk, es, fr, ar).

Website URLs must branch off from one of these top-level domains. For instance, "google.com" branches off of the "com" top-level domain. And “google.co.uk” branches off both the “com” organizational domain and the “uk” geographic domain. Some websites, like "images.google.com" and "maps.google.com", branch even further and are known as subdomains.

DHCP

Computers, and the humans that use them, need a system for allocating these IP addresses. At one point in the Internet's history, a human network administrator was responsible for this, assigning IP addresses to computers manually. Nowadays, the Dynamic Host Configuration Protocol, or DHCP, can do this automatically. When computers connect to a network, they will connect to a DHCP server. The DHCP server then accesses a pool of available IP addresses and assigns each computer on the network a unique one.

So, using DHCP and DNS, devices on the Internet are able to receive their own IP address and determine which IP address corresponds to the website that a user is trying to visit. These systems are crucial, allowing the Internet Protocol to effectively facilitate communication across the Internet.

Exit Codes

Overview

You may have noticed that the `main` function definition returns an `int`, but in the past we haven't been returning any value at the end of the main function. By default, if no return value is specified in the `main` function, the compiler will automatically assume that the `main` function returns `0`. The value that the `main` function returns is referred to as the program's **exit code**. As your programs become longer and more complicated, exit codes can be a valuable tool.

Key Terms

- exit code
- input validation

Using Exit Codes

By convention, if a program completed successfully without any problems, then it should return with an exit code of `0`. That's why the compiler assumes that if no return statement is provided at the end of `main`, the program should return `0`. You could, however, explicitly specify `return 0` at the end of a program.

Any non-zero exit code (commonly `1` or `-1`) conventionally means that there was some sort of error during the program's executing that prevented the program from completing successfully.

One common use of exit codes is during **input validation**: when the program checks to make sure that the inputs provided by the user are valid. For instance, if a program expects two command line arguments, but only receives one, it might return a non-zero exit code to signal an error.

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

int main(int argc, string argv[])
{
    if (argc == 2)
    {
        printf("hello, %s\n", argv[1]);
        return 0;
    }
    else
    {
        return 1;
    }
}
```

Take the above program, which takes (in addition to the program's name) a command line argument specifying the user's name. The program then says hello to the user.

Upon starting the `main` function, the program checks to see whether `argc` is 2. If it is, then the user's input is valid: the program can say hello, and successfully return with exit code `0`.

On the other hand, if the user didn't provide the correct input values (say, by not providing enough command line arguments, or by providing too many), then `argc` would not be equal to `2`.

The program would then execute the `else` block, which returns the number `1` as an exit code, indicating that there was an error in the program's execution.

Debugging

If you run programs normally from the command line, you won't actually see the return values that the `main` function returns. However, many debugging tools, which are programs designed for helping programmers find sources of problems in their code, will allow you to see the exit code with which the `main` function exited.

Knowing the exit code can be a valuable tool for determining why a program failed during the debugging process. In larger programs, which may include many instances of error checking and input validation, the program may return a different exit code for each error.

If the program fails, knowing which status code the program exited with can help determine what went wrong.

CS50

File I/O

Overview

Interacting with data stored on a computer's disk is integral to many programs. Data storage allows information to be used after the lifetime of a program without the hassle of inputting it again. Data is stored on a disk in the form of **files**, collections of data organized in such a way that the computer can understand. Files store a variety of media, including audio, text, movies, pictures, and more.

Key Terms

- file
- file pointer
- file extension
- I/O

Remember that all files are fundamentally just bits – 0s and 1s – that have been organized in a specific way. File types (a "video file," for instance) are merely abstractions of these bits. Likewise, **file extensions**, such as .txt, .c, or .mp3, merely act as references for computer programs that tell them what they should expect to find inside the file and what they should use the file for. So when a computer sees a certain extension, it knows that the corresponding file (i.e., its 0s and 1s) is formatted in a specific way and should be opened with an appropriate program.

Interacting With Files

The "**I/O**" in "file I/O" stands for **input/output**. Thus, file I/O refers to the process of retrieving information from and storing information in files. In general, the file I/O process, broadly, consists of a few steps. First, you must open the file, specifying the ways in which the file can be manipulated. After that, the file's data is free to be manipulated in any of the specified ways. Finally, the file must be closed!

Diving Into Code

Files are interacted with in C via "file pointers," i.e. references to files, which in code are the `FILE *` data type. New file references are usually initialized with the library function `fopen`, which takes two arguments (both strings): the filename and the mode for which the file should be opened.

There are many different ways to manipulate files. The most important of these are reading ("r"), writing ("w"), and appending ("a"), which is just like writing but done directly at the end of the file. To write to a file, we can use `fprintf`, specifying the file pointer to which we want to write and also what it is we want to write. Other functions, like `fgetc`, let us read from a file, getting characters, strings, and the like from the file pointer. Since there are many methods for writing and reading various types of data, we need to make sure to use the one which best suits our needs!

```
1  | // Write to file
2  | FILE *fp = fopen(“document.txt”, “w”);
3  | fprintf(fp, “Hello, world!\n”);
4  | fclose(fp);
5  |
6  | // Read from file
7  | fp = fopen(“document.txt”, “r”);
8  | char c = fgetc(fp);
9  | while (c != EOF)
10 | {
11 |     printf(“%c”, c);
12 |     c = fgetc(fp);
13 | }
14 | fclose(fp);
```

Error Checking

It's also very important to understand and check for potential errors that might occur in the file I/O process. For instance, unlike the previous example, we should always make sure fopen was successful.

Take a look at the code at right and note the error checking to ensure `fopen` was indeed successful. Here, the mode specified was writing, or "w," so failures could occur if the file exists and it is corrupted. Had the mode been reading, or "r" (as in line 7 of the former code), errors could have resulted from trying to read from a nonexistent file. Also, not having the appropriate permission to open a file (to read from or to write to) will also return an error.

```
// Write to file
FILE *fp = fopen(“document.txt”, “w”);
// ensure the file was successfully opened
if (fp == NULL)
{
    fprintf(stderr, “Error opening file.\n”);
    return 1;
}
// continue...
```

CS50 Functions

Overview

Functions are reusable sections of code that serve a particular purpose. Functions can take inputs and outputs, and can be reused across programs. Organizing programs into functions helps to organize and simplify code. This is an example of **abstraction**; after you've written a function, you can use the function without having to worry about the details of how the function is implemented. Because of abstraction, others can use (or "call") the function without knowing its lower-level details as well.

Key Terms

- functions
- abstraction
- return type
- side-effect
- return value
- scope

Function Syntax

```
#include <stdio.h>

void say_hi(void)
{
    printf("Hi!\n");
}

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

All programs you've written in C already have one function: `main`. However, programs in C can have more functions as well. The program on the left defines a new function named `say_hi()`.

The first line of a function requires three parts: first, the function's **return type**, which is the data type of the function's output that is "returned" to where the function was called. If the function does not return a value, the return type is `void`. Second, the function's name; this cannot include spaces and cannot be one of C's existing keywords. Third, in parentheses, the function's parameters, also known as arguments. These are the function's inputs (if there are none, use `void`). After this first line (known as the declaration line), the code defining the function itself is enclosed in curly braces.

In the example above, the `say_hi()` function causes `"Hi\n"` to be printed to the screen. This is called a **side-effect**, which is something a function does outside of its scope that is not returning a value. The `say_hi()` function is then called twice in the `main` function. Functions are called by writing the function's name, followed by any arguments in parentheses, followed by a semicolon. When the program is run, `"Hi\n"` prints to the screen twice.

Inputs and Outputs

The example on the right shows a function, `square`, which takes input and output. `square` takes one input: an integer called `x`. It also returns an `int` back to the where the function is called. Line 5 of the function specifies the function's **return value**, denoted by the word `return`. In this case, the `square` function returns the input value `x` multiplied by itself. When the `return` line is reached, the function is exited and the return value is returned to where the function was initially called.
Now that we've written this function, we can use `square` elsewhere in our program anytime we want to square a number. In the `main` function on the right, the `square` function is called three times: each time, the function is evaluated and returns the appropriate return value in the place of the function. So `printf("%i\n, square(2))` has the equivalent effect of writing `printf("%i\n", 2 * 2)` or `printf("%i\n", 4)`.

```
1  #include <stdio.h>
2
3  int square(int x)
4  {
5      return x * x;
6  }
7
8  int main(void)
9  {
10     printf("%i\n", square(2));
11     printf("%i\n", square(4));
12     printf("%i\n", square(8));
13 }
```

Scope

Variables that are defined inside of functions or in the list of function parameters have local **scope**, meaning those variables only exist inside of the function itself and have no meaning elsewhere. In the example above, if you were to try to reference the variable `x` inside of the `main` function, the compiler would give you an error; the `main` function doesn't know what `x` means, only `square` does. Likewise, any variables defined inside of `main` can't be accessed from inside of `square`.

If variables are defined outside of any functions, they have global scope instead of local scope. This means they can be accessed from any of the functions in the file. However, global variables are more difficult to keep track of and can be changed from any location in the program. Because global variables have global scope, that variable name cannot be reused in other parts of your program.

CS50

Hexadecimal

Overview

Recall that our computers break everything from ASCII symbols to source code down into combinations of 0s and 1s (**binary**). Those 0s and 1s are not that efficient when it comes to expressing large numbers. To express the decimal number 15, for instance, we need four place values in binary: 1 1 1 1. Because four digits of binary can represent 16 values, computer scientists settled on hexadecimal, a number system of base 16, to represent those larger numbers.

Key Terms

- binary
- hexadecimal
- RGB values

Hexadecimal

In the decimal system (base 10), we have ten digits, 0-9, and each place value represents the next power of 10. So the nth place value can be calculated by taking 10_{n-1}, like in binary (base 2), where we could calculate the nth place value by taking 2_{n-1}.

Similarly, in **hexadecimal** (base 16), we use 0-9 for the first ten digits and the letters A-F for the remaining six. We can think of A as 10, B as 11, and so forth. As you might guess, hexadecimal's place values are based on powers of 16. Note that all the hexadecimal place values are found in binary, albeit more spread out. This makes sense when we remember that 2^4 = 16 and that what takes 4 digits to express in binary can be expressed in 1 digit in hexadecimal.

To convert numbers directly from binary to hexadecimal, simply block off the binary number into chunks of four digits and express what they represent as a single hexadecimal digit. For example, 0 0 0 0 in binary would be a 0 in hexadecimal, and a 1 1 1 1 in binary would be converted into an F (which represents 15) in hexadecimal. This optimization allows us to represent much larger numbers using fewer characters.

Decimal System

3	1	9
100s	**10s**	**1s**
(3 x 100) +	(1 x 10) +	(9 x 1)

300 + 10 + 9
319

Hexadecimal System

1	3	F
256s	**16s**	**1s**
(1 x 256) +	(3 x 16) +	(15 x 1)

256 + 48 + 15
319

Binary System

1	0	0	1	1	1	1	1	1
256s	**128s**	**64s**	**32s**	**16s**	**8s**	**4s**	**2s**	**1s**
(1 x 256) +	(0 x 128) +	(0 x 64) +	(1 x 32) +	(1 x 16) +	(1 x 8) +	(1 x 4) +	(1 x 2) +	(1 x 1)

256+ 0 + 0 + 32 + 16 + 8 + 4 + 2 + 1
319

Hex Colors

One application of the hexadecimal system is the representation of colors. As you may know, all colors are made up of varying levels of red, green, and blue. We refer to these as the **RGB values**. Each of the three colors can have a value between 0 and 255 (16^2-1), which means we need to be able to represent 16,777,216 different colors. And using the hexadecimal number system, we are able to do this in only 6 digits! Imagine using the binary system to express that many colors. It would take 4 times as many digits.

CS50 How Computers Work

Overview

Computers were invented by many teams of people, all working on particular parts. Like most machines, a computer is made of separate pieces with specific functions that all work together. The physical pieces of a computer are called **hardware**, and the virtual pieces of a computer are called **software**. Within both hardware and software, there are individual components, and within those components, there are even more components. This pattern of getting smaller and going lower into the innerworkings of a computer continues until the level of transistors and binary. However, thanks to the work of computer scientists, we can leverage these devices without worrying about low level details.

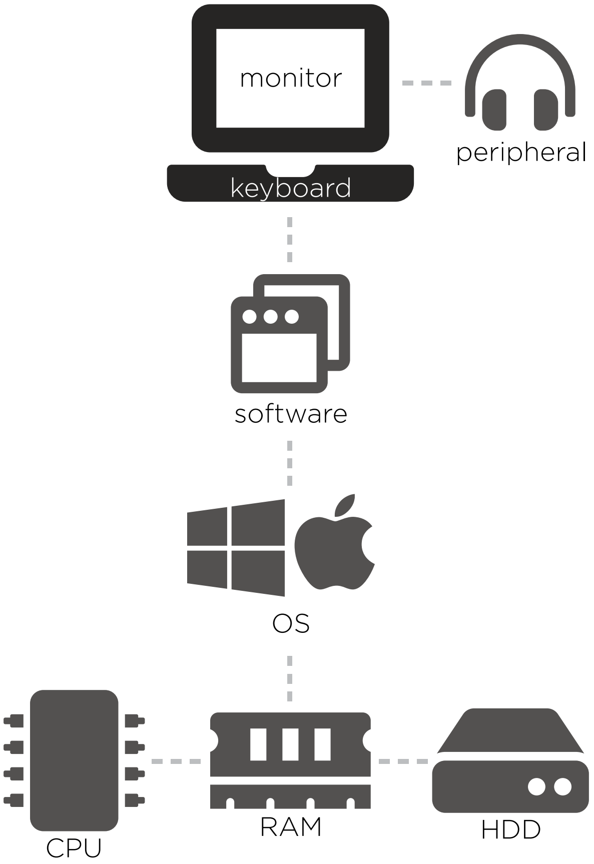
Key Terms

- hardware
- software
- CPU
- RAM
- HDD
- SSD
- peripherals
- OS

Hardware

Hardware consists of all of the physical components of a computer. On the outside of a computer, most computers have a keyboard, a mouse, and/or a trackpad, which are used to interact with a computer by inputting information. Computers also have a monitor, or a screen, to output information through computer-generated images. On the inside of a computer, the main components are the CPU and memory. The **CPU**, or processor, is responsible for performing calculations and following instructions. The CPU works tightly with memory to fetch instructions and store calculated results. To fulfill different needs, there are a few types of memory within a computer: RAM and hard disk drives. **RAM**, or Random Access Memory, is the short-term memory that software can use to store data quickly and temporarily. In contrast, hard disk drives (**HDD**) store memory more permanently, but are much slower than RAM. Other drives, known as solid state drives (**SSD**) also store data like hard disk drives, but not without their own trade offs: although they are significantly faster at reading and writing data than HDDs, they are much more expensive.

Not all hardware that's used in computing is inside the computer. Often, devices that are not a part of the computer itself will connect to and work with computers. These devices are known as **peripherals**. Some peripherals include external speakers, flash drives, and drawing tablets. To connect these peripherals to the computer, computers use physical ports, such as HDMI and Universal Serial Bus (USB).



Software

Hardware components communicate with software components through a piece of low level software called the Operating System, or OS. The **OS** is the computer's manager; it's in charge of translating input from your keyboard and mouse, displaying information on the screen, and moving things around in memory. It provides you with the user interface that you're familiar with and decides how to delegate hardware resources for different software applications. It is through the OS that software applications are able to work with the various hardware components. Because the OS is loaded into RAM when you turn on your computer, you are able to use a computer and even program a computer without directly interacting with the internal hardware yourself.

Engineers can build software applications like word processors and web browsers by writing programs that interacts with the OS. By building on top of what others have done, software can be created directly on a computer using high level tools like code, enabling engineers to develop new and improved applications very quickly.

CS50

HTML

Overview

HyperText Markup Language, or **HTML**, is the language that describes the contents of web pages. It's not a programming language (it doesn't contain loops, if statements, etc.), but it does allow a web designer to decide how a web page is laid out. When a user requests a web page, the web server will send the contents of that page in HTML. The web browser interprets this HTML and displays the web page to the user.

Key Terms

- HTML
- element
- tag
- DOM
- attribute

HTML Basics

To the right is a basic, sample HTML web page. The first line clarifies that the document is an HTML document (specifically, an HTML5 document). After that, HTML code is organized as a series of nested **elements**. Each element begins and ends with a **tag** that indicates what type of element it is. Anything between those tags is the contents of the element. In our example, the outermost element is the html element, since it is enclosed with `<html>` at the start, and `</html>` at the very end. Everything in between those two tags is part of the html element.

```
<!DOCTYPE html>

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

In general, opening tags will take the form `<tagname>` and closing tags will take the form `</tagname>`. Inside of our `html` element are two other elements: `head` and `body`. The `head` element contains information about the webpage that isn't in the body of the web page itself. For instance, the head element here contains a `title` element, which sets the title of the webpage to `hello, world`. Inside of the body element, which defines what's actually in the main content area of the webpage, is just the word `Hello!`

You can think of the structure of an HTML document as a tree-like hierarchy. The `html` element is at the root of the tree. Branching off of it are the head and body tags. Branching off of the `head` tag is the `title` tag, and so forth. This model of viewing an HTML document as a tree is known as the Document Object Model, or **DOM**.

Common Tags

HTML offers many tags that can be used to format a web page. The ones in our example – `html`, `head`, `title`, and `body` – are very common and will likely appear in every web page we write. Other tags may appear in particular situations. For example, headings in web pages are denoted by the tags `<h1>` through `<h6>`, where `<h1>` is the largest, main heading, and each subsequent one is smaller. The `<p>` tag denotes a paragraph.

Element Attributes

In addition to having a name, HTML elements can also have **attributes**, which provide additional information about them. For instance, the `<img>` (image) tag takes an attribute called `src`, which specifies the address for, or where to find, an image. So a tag like `<img src="cat.jpg">` would place the image cat.jpg on the webpage, so long as `cat.jpg` is in the same directory as the HTML document.

To create links to other pages in HTML, we use the `<a>` tag. The `<a>` tag takes an attribute called `href`, which specifies what web page the link should link to. As a result, HTML like `<a href="http://google.com">Click Here</a>` would create a link labeled "Click Here" which, when clicked, would take the user to Google.

All HTML elements can also have an `id` attribute, which must be unique. The `id` can help to identify particular elements in the webpage. HTML elements can also have a `class` attribute, which does not have to be unique and can also be used to identify HTML elements. We'll see the utility of the `id` and `class` attributes when we begin dealing more with CSS and JavaScript.

CS50

HTTP

Overview

The Hypertext Transfer Protocol, or **HTTP**, is a protocol for how web browsers communicate with web servers. When a user wishes to visit a webpage, their web browser (which may be referred to as the **client**) must request the contents of the web page from a web **server**. In response, the web server must interpret the request and send the requested page back to the client. HTTP facilitates this process and sets a standard way for these requests to be sent and received.

Key Terms

- HTTP
- client
- server
- GET
- POST
- Status Code

GET and POST Requests

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

When a user requests a web page by typing a URL into their web browser, the web browser sends a particular type of HTTP request called a **GET** Request. The text of a GET request begins with the word `GET`, to indicate the request type. Following the word `GET` is a path indicating which web page the user is requesting, called the "Request URI," where URI stands for Uniform Resource Identifier.

Following `GET` is `/`, which indicates the root of the web page, such as when you type a URL like `google.com/` or `facebook.com/` without specifying anything after the `/`. Finally, the first line of the GET request will end with the version of the HTTP protocol that the request is using, generally `1.1`. The next line specifies the "Host," which is the domain name which the user is requesting a page from.

Web browsers can also submit a different type of HTTP request, called a **POST** request, which is meant for transmitting data from the client to the web server, such as when a user fills out an online form. In this case, a client would have asked the server for the blank form via a GET request. Once submitted, the filled out form would be sent back to the server via a POST request.

Status Codes

When a web server receives an HTTP request from a client, the server must send a response back to the client. Servers indicate the results of requests with **status codes**, which they send back to the client.

For instance, if a client requests a web page, the server should send back the contents of that web page. If the server has the page that the user is looking for and is able to respond with it successfully, then the server sends the status code `200`, which means that the request was handled successfully. But if the user requests a page that doesn't exist on the web server, then the server responds with status code `404`, which stands for "Not Found."

Other types of errors are also represented by status codes. For example, if the user tries to access a web page that the user does not have permission to access, then a web server will respond with status code `403`, which means "Forbidden." If an error occurs in the web server while trying to process a user's request, then the server will frequently respond with status code `500`, which stands for an "Internal Server Error."

```
HTTP/1.1 200 OK
Content-Type: text/html
```

Status Code	Meaning
200	OK
301	Moved Permanently
302	Found
403	Forbidden
404	Not Found
500	Internal Server Error

CS50 Images

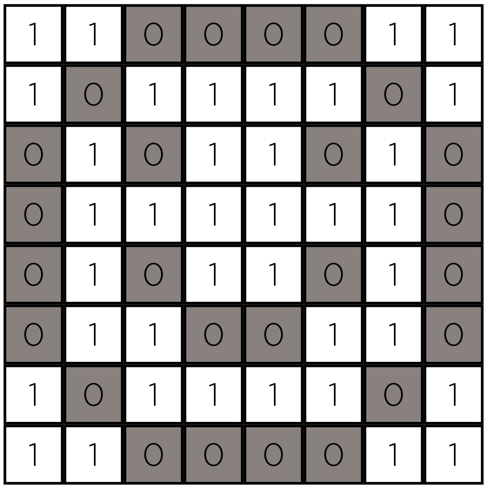
Overview

From social media to cancer screenings, newspapers to comic books, images are important in our lives. Images are stored as files, a series of bytes, just like the programs, word docs, and text files you're used to writing. Common image file formats include bitmaps (.bmp), JPGs (.jpg), PNGs (.png), TIFFs (.tiff), and GIFs (.gif).

Key Terms

- pixel
- hexadecimal
- header
- lossy
- lossless

Bitmaps



1	1	0	0	0	0	1	1
1	0	1	1	1	1	0	1
0	1	0	1	1	0	1	0
0	1	1	1	1	1	1	0
0	1	0	1	1	0	1	0
0	1	1	0	0	1	1	0
1	0	1	1	1	1	0	1
1	1	0	0	0	0	1	1

Our entire screen is composed of **pixels**, little dots with programmable color and brightness values. A bitmap describes a pattern of all the pixel values that make up an image: when our screen's pixels become those values, the image will appear. A bitmap takes some number of bits per pixel (bpp). More bits can be used to create a more detailed color palette. Back when screens were black and white, just a single bit was used per pixel, with `0 = black`, `1 = white`, as shown at left.

RGB Triples

The bitmaps we've worked with contain three bytes for each pixel in a color image. Each byte specifies a number between `0` and `255`, or, in **hexadecimal**, `0x00` to `0xff`. These three numbers detail how much red, green, and blue to put in that pixel. We can make sense of how this information comes together by thinking about painting, where mixing different combinations of just a few colors can produce many, many different shades. In this case, red, green, and blue can be combined to make the entire rainbow with varying levels of brightness.

Bitmap Headers

Bitmaps also have a **header** – a few bytes at the beginning of the file that tell the display program how to interpret the bits in that file. For the common .bmp Microsoft file extension, we use the struct at right to specify its header. These fields specify the size of the image in bytes, its width and height in pixels, and more. Having this information bundled together ensures that our display program knows exactly how to format the image.

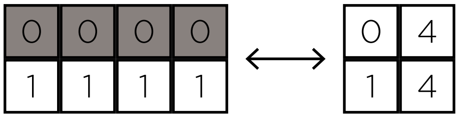
```
typedef struct
{
    DWORD biSize;
    LONG biWidth;
    LONG biHeight;

    (...)

} __attribute__((__packed__))
BITMAPINFOHEADER;
```

Other Image File Formats

Do we need to store all of the information corresponding to every pixel in our image file? After all, many images feature a lot of repetition and redundancy (see what we did there?). Is there a way to encode this repetition to create smaller file sizes?

The answer is, typically, yes. Notice the four horizontal repetitions of 0 and 1, respectively, in the example at right. In the file below it, we encoded `(pixel value, repeat number)` pairs. What we've just done here is compressed the original file, resulting in a new file of smaller size.

GIF compression



0	0	0	0
1	1	1	1

0	4
1	4

There are two main types of file compression: lossy and lossless. In **lossy** file compression, files, such as JPGs, are compressed in such a way that data is lost, meaning that the original file cannot be completely recovered. On the other hand, in **lossless** compression, which is used with PNGs and GIFs, no data is lost and the original file can be exactly reconstructed. The compression in the example above falls into this category.

CS50 Insertion Sort

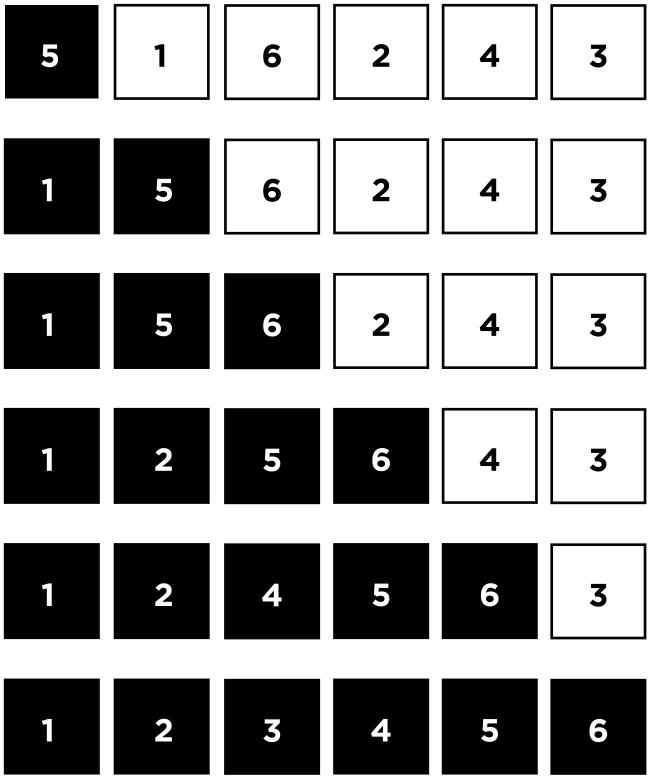
Overview

Insertion sort is yet another algorithm to sort arrays, but this time it does not require multiple iterations over the **array**. Like usual, optimizations usually force the programmer to sacrifice something else. However these sacrifices are nearly negligible, in the case of insertion sort, when the array is small or the array is nearly sorted. Similar to selection sort, the array of elements will be split into two parts: a sorted portion and an unsorted portion.

Key Terms

- insertion sort
- array
- pseudocode

Step-by-step process for insertion sort



5 1 6 2 4 3

1 5 6 2 4 3

1 5 6 2 4 3

1 2 5 6 4 3

1 2 4 5 6 3

1 2 3 4 5 6

Implementation

In the case of having two elements in the array, the implementation is relatively simple. Consider the first element to be automatically in the sorted portion of the list. Look at the next element in the array, and determine where it fits in the sorted list. This can be applied to a larger array with the following **pseudocode**:

```
for each unsorted element, n, in the array

        determine where in the sorted portion of the
        array to insert n

        shift sorted elements rightwards as necessary
        to make room for n

        insert n into sorted portion of the list
```

When this is implemented on the example array, the program would start at `array[1]`, which is 1, since an array of size one (`array[0]`) is already sorted. 5 would get shifted over to the right, and 1 would be moved to `array[0]`. Next, the program looks at 6. 6 is greater than 5 so no elements need to be shifted to make room for it. And so on and so forth. Eventually with this procedure, the entire array will be sorted.

Sorted Arrays

There is no guarantee that after any step in the implementation, any elements are in the correct location in the array. Even if an element did happen to be in its correct location in the initial array, it is likely that it would be moved to a different location within the sorted array before it would be shuffled back into its final location. Also note that in insertion sort, all the elements to the right of the newest element in the sorted list have to be shifted over one space. So while it may seem like insertion sort involves *n* steps, we are increasing the amount of times an element is being moved because elements are being shifted over to accomodate new elements rather than just being swapped. This is, however, dependent on the order of the initial array. It is also worth considering that sorting algorithms need to address cases in which two elements are equal. When dealing with few unique elements the key is just to be consistent. If, in the implementation of insertion sort, there was a line that checked if the current element was equal to an element in the sorted array, it should always have the same outcome, whether it is to the right or the left of that element is irrelevant.

CS50 Internet Basics

Overview

Programming isn't just limited to writing programs that run on the command line. Code can also be written for web browsers and and shared on the Internet. To be able to share information anywhere, a standard set of rules for sending, receiving, and interpreting the information must be set. Because the Internet connects people and computers from all over the world, there are many different systems and protocols in place that work together in order to allow people to use the Internet effectively. Understanding what these are and how they work will enhance your understanding of the Internet and computer science overall.

Key Terms

- IP address
- access point
- DHCP
- DNS
- URL
- TCP
- HTTP

IP Addresses

Devices on the Internet are assigned an **IP address** (which stands for Internet Protocol) to help identify them and allow them to be found by other devices also on the Internet. IP addresses take the form of `#.#.#.#`, where each `#` is a number in the range of 0 to 255. This allows for about 4 billion possible IP addresses. Although this may sound like a lot, there are far more devices on the Internet than that. This has led to some workarounds, including assigning some devices private IP addresses that together share a single public IP address.

In the long term, however, the 32-bit IP address scheme (IPv4) we mostly use today will be replaced by a 128-bit address scheme called IPv6. While IPv4 addresses take the form of 4 numbers, each representing an 8-bit value, IPv6 addresses have 8 numbers – each representing a 16-bit value – in the form `#:#:#:#:#:#:#:#`.

When information is being sent across the Internet, IP addresses are used so that the Internet knows where the information is being sent to and from. In this sense, it's very much like sending physical mail: information has a both a mailing (to) and a return (from) address.

Connecting to the Internet

Several steps are involved in connecting a device to the Internet. For a wireless device (like a laptop or cell phone), it must first connect wirelessly to an **access point** (known as an AP). For many consumers, this access point takes the form of a home router. This access point is connected to a switch, which is connected to another router, which can then connect to the rest of the Internet.

Two other servers are particularly important for connecting to the Internet: DHCP and DNS. **DHCP**, which stands for Dynamic Host Configuration Protocol, is responsible for assigning computers IP addresses. Early in the Internet Age, network administrators had to manually assign IP addresses to all computers, but thankfully, DHCP now automates this process.

It would be very difficult if everyone using the Internet had to remember each IP address for every website they wanted to visit. Instead, most people type a text-based address (such as "google.com") into their web browsers to access a page. This text-based address is called a **URL**, or Uniform Resource Locator. But how do our computers know what IP addresses to take us to based on our text input?

This is where **DNS**, which stands for Domain Name System, comes in. The term DNS refers to servers that take URLs and converts them to and from IP addresses. When a user types a URL into their web browser, DNS servers look up the URL and then determine which IP address that name refers to, relating this important information back to the computer.

Other Protocols

Several other protocols are involved in ensuring that communication on the Internet works effectively. **TCP**, the Transmission Control Protocol, is responsible for guaranteeing the delivery of all data packets that are submitted via the Internet. It also makes sure that these packets of information sent via the Internet know what service they are meant for (web browsing, email, etc.). **HTTP**, the Hypertext Transfer Protocol, is another protocol which helps web browsers communicate with server.

CS50

IP Addresses

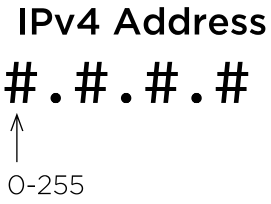
Overview

The **Internet Protocol** is a protocol, or set of rules, that helps define how information on the Internet is transmitted. Part of this protocol involves assigning each device on the Internet an **IP Address**, which helps to identify that device on the Internet. IP has gone through several different versions, the most recent of which is **IPv6**, which is intended to replace the existing protocol, **IPv4**.

Key Terms

- Internet Protocol
- IP Address
- IPv6
- IPv4

IPv4 and IPv6 Addresses



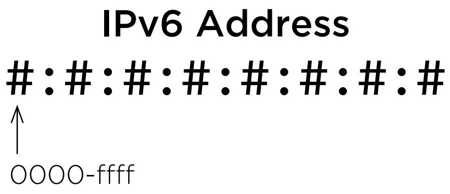
IPv4 Address

`#.#.#.#`

0-255

Under the IPv4 system, each IP address is composed of four numbers separated by decimal points. Each number is a decimal number in the range of 0 to 255 inclusive (8 bits of space). As a result, each IPv4 address is 32 bits, which means there can be at most 232 addresses under IPv4. This amounts to about 4.3 billion addresses total.

However, as the Internet has grown, 4.3 billion addresses is no longer sustainable to support all of the devices that are trying to connect to the Internet. As a result, the IPv6 standard was developed in order to add more possible IP addresses.



IPv6 Address

`#:#:#:#:#:#:#:#`

0000-ffff

Under IPv6, each IP address consists of eight numbers, separated by semicolons. Each number is a 16-bit number (compared to the 8-bit numbers used in IPv4). Instead of representing each number as a decimal, IPv6 uses hexadecimal (16-bit) instead, in the range of 0000 to fffff. Since each IPv6 address stores 128 bits (8 numbers * 16-bits), that means that there are more than 340 billion billion billion billion possible IP addresses. This is significantly more addresses than are currently used, so many IPv6 addresses currently include several 0s among their 8 component numbers.

As a shorthand method, IPv6 addresses can be abbreviated by cutting off any leading 0s in front of hexadecimal numbers and replacing multiple consecutive component 0s with a double colon (`::`).
For instance, the IP address `28aa:0000:0000:0000:0000:0000:0018:a5b2` could be abbreviated to just `28aa::18:a5b2` by removing leading 0s and replacing multiple consecutive 0s with double colons. Note that there can only be one double colon per address in this abbreviated format.

Private IP Addresses

Not all IP addresses are accessible on the Internet at large. Some addresses, known as private IP addresses, are set aside to be used within a particular local network. Other computers on the local network can communicate with one another via their private IP addresses, but computers outside of the network don't have access to them.

Often, devices with private IP addresses will share a single public IP address. This helps reduce the number of public IP addresses that are needed under the IPv4 standard. Certain ranges of IPv4 addresses, such as those which take the form `10.#.#.#`, `172.16.#.#` - `172.31.#.#`, or `192.168.#.#`, are set aside to be used specifically for private IP addresses.

The IP address `127.0.0.1` is an IP address that connects to the same machine that the user is currently using, rather than connecting to a different one. It is known as the loopback address, or the “localhost.” In computers that use IPv6, this address is `0:0:0:0:0:0:0:1`, or `::1`. You know what they say, there’s no place like `127.0.0.1`.

CS50

JavaScript

Overview

If you've seen websites with cool animations when you interact with them, it's likely that those features were written in JavaScript. Each programming language plays a certain role in computer science, and JavaScript's job is in the web browser. JavaScript is a language used in web development to program the behavior of web pages. Because JavaScript was created for this use, JavaScript has many features that other languages such as Python, C, and Java do not have.

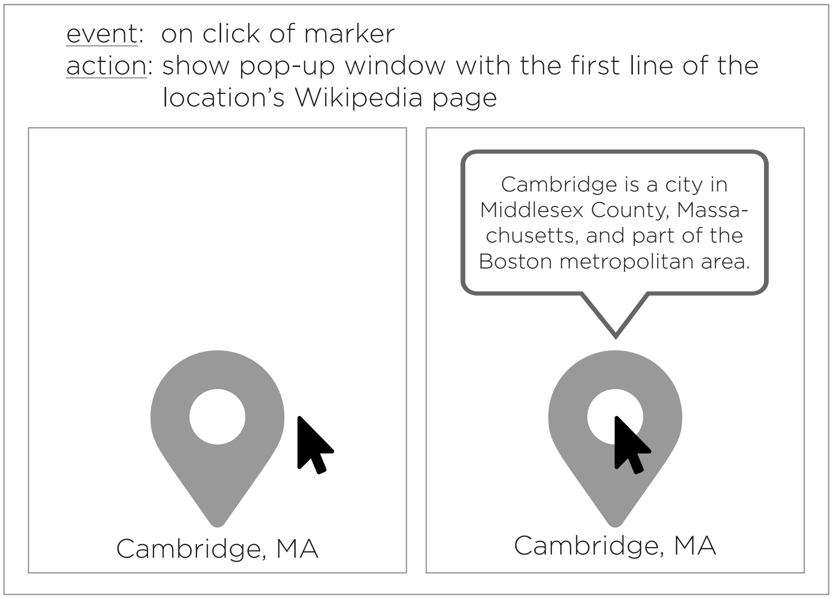
Key Terms

- JavaScript
- event
- anonymous function
- jQuery

JavaScript

JavaScript is a programming language used in web browsers to create dynamic web pages. Because JavaScript runs in all major web browsers, it is universally supported and used in the majority of modern websites. JavaScript is primarily used on the client-side of web applications, meaning the code is run on a user's browser rather than somewhere on a external server. This means that without additional HTML requests, JavaScript does not have access to information on the server.

JavaScript works closely with HTML and CSS to create user interfaces for web applications. Javascript is written within script tags of an HTML page and has many functions that allow for direct manipulation of HTML using the DOM. Part of what makes JavaScript so powerful and dynamic is its focus on responding to **events**, or actions from a user or other devices. An event-driven language is one that enables a program to act according to these events. Some examples of events include mouse clicks, key presses, scrolling, or outputs from sensors. The job of many JavaScript programs is to do something depending upon what the user does, and JavaScript provides built-in features to support this kind of functionality.



event: on click of marker
action: show pop-up window with the first line of the location's Wikipedia page

Cambridge, MA
Cambridge is a city in Middlesex County, Massachusetts, and part of the Boston metropolitan area.
Cambridge, MA

One of these features is unnamed functions, or **anonymous functions**. Functions are named so we can call them in various parts of our program, but since JavaScript is event-driven, many actions are only triggered by particular events. In other words, it is common to have functions that are only used once in a program. Therefore, JavaScript allows us to create anonymous functions. Unlike declared functions, anonymous functions run immediately without storing them in memory first. This simplifies code, lets us assign functions as variables, and allows us to pass functions around as arguments in other functions. All of these features simply make creating interactive and user-friendly web pages more convenient.

Additional Resources

JavaScript is widely used among web developers, so there are many JavaScript resources available to make coding more convenient and to expand the range of possibilities. One of these resources is a library called jQuery. **jQuery** is a cross-platform JavaScript library that simplifies some of JavaScript's syntax and includes some additional helpful functions. It uses the $ symbol as a global variable to access jQuery. In addition to jQuery, JavaScript has a plethora of resources available. Some of these resources include libraries of pre-built JavaScript components and tools for data visualization. Because web development brings together so many technologies and has such a big impact on how information is presented, there are constantly new resources being developed around JavaScript. Part of web development is keeping up with these technologies and being able to adapt quickly. Be sure to check out which resources could be useful for your next project!

CS50

Libraries

Overview

Libraries are shared collections of code that programmers can use to work with one another. Libraries usually include functions that may be commonly used among programmers. For example, the C library string.h includes many useful premade functions to manipulate strings (see below). By allowing us to use functions that others have already written, libraries enable us to build off of the work of others and use their functions in our own programs, instead of reinventing those functions ourselves.

Key Terms

- library
- header file

Using Libraries

To use a functions from a library in C, remember to `#include` the **header file** (such as with `#include <math.h>`), which defines the library's functions, at the top of your source code file. When compiling your code, you'll also need to link the library so that the resulting object code knows how to execute the functions.

Some Common C Library Functions

In `ctype.h`:

`isalnum()` takes a `char` as input, and returns `true` if the character is alphanumeric and `false` otherwise
`isalpha()` takes a `char` as input, and returns `true` if the character is alphabetic and `false` otherwise
`islower()` takes a `char` as input, and returns `true` if the character is lowercase and `false` otherwise
`isupper()` takes a `char` as input, and returns `true` if the character is uppercase and `false` otherwise
`tolower()` takes a `char` as input, and returns the character converted to lowercase if possible. If it's not possible, it returns the original character unchanged.
`toupper()` takes a `char` as input, and returns the character converted to uppercase if possible. If it's not possible, it returns the original character unchanged.

In `math.h`:

`ceil()` takes a `double` as input, and returns the smallest integer that is not less than the input, as a `double`
`cos()`, `sin()`, and `tan()` each take a `double` as input, and return the cosine, sine, or tangent of the input
`floor()` takes a `double` as input, and returns the largest integer that is not greater than the input, as a `double`
`pow()` takes two `double`s as input, and returns the value of the first input raised to the second value power
`lround()` takes a `double` as input, and returns a `long int` representing a rounded version of the number
`log()`, `log10()`, and `log2()` take a `double` as input, and return the logarithm of the number (base e, base 10, and base 2, respectively)

In `stdio.h`:

`printf()` takes a `string` as input, and prints it to standard output, displaying it on the screen

In `stdlib.h`:

`atoi()` takes a `string` as input, and converts the `string` to an `int` if possible, returning the `int`
`rand()` returns a pseudorandom integer, and will usually be seeded with `srand()` first

In `string.h`:

`strlen()` takes a `string` as input, and returns the length of the string, not including the null terminator
`strcmp()` takes two `string`s as input, and returns `0` if they are equal, less than `0` if the first `string` comes before the second, and greater than `0` if the first string comes after the second one
`strstr()` takes two `string`s as input, and finds the first occurrence of the second `string` in the first.

There are many more functions defined in these libraries and other libraries, and it is often a good idea to explore the existing C libraries to see what functions are available to you so that you don't re-create code that you could use library functions for instead. Check out reference.cs50.net for more information on C Library Functions.

CS50

Linear Search

Overview

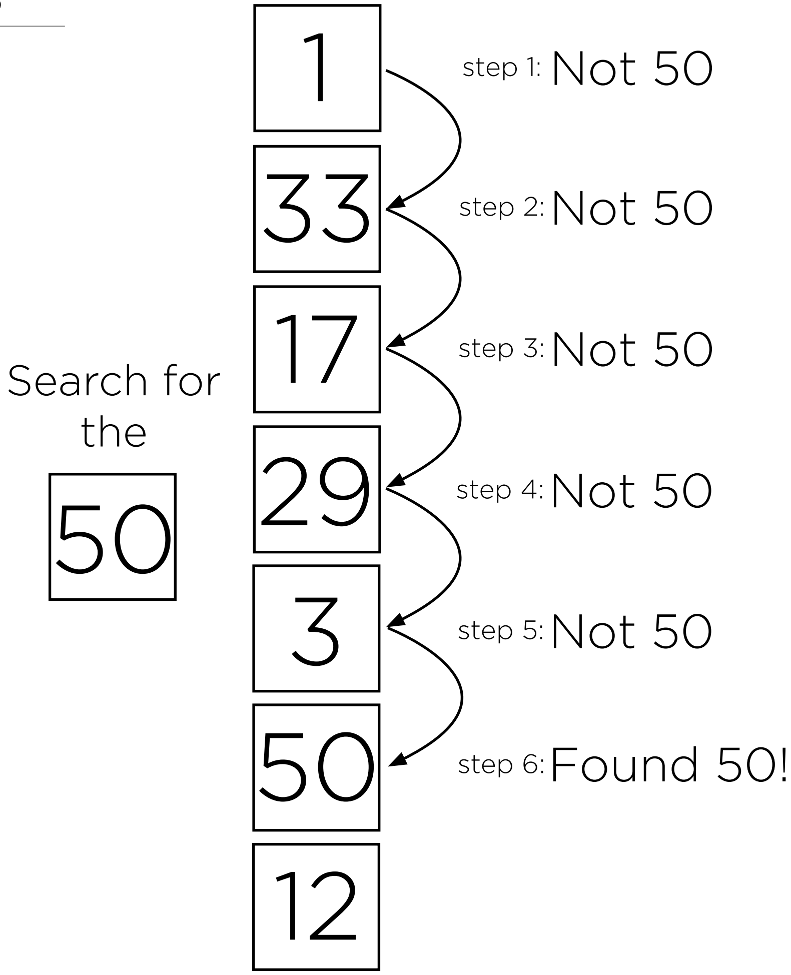
There are many different **algorithms** that can be used to search through a given list. One such algorithm is called **linear search**. This algorithm works by simply checking every element in the list in order. It will start at the beginning of the list and increment through the list until the desired element is found. In this way, linear search would require checking every **element** before reaching the conclusion that the element does not exist in the list.

Key Terms

- algorithm
- linear search
- element
- computational complexity
- constant

Efficiencies and Inefficiencies

While the linear search algorithm is correct, it is almost never the most efficient. The worst, or least efficient, case would be when the desired element is the last element in the list or not in the list at all (both have the same efficiency). We would have to look through every element in the list to find the one we're looking for. This would take *n* steps, where *n* is the length of the list. The **computational complexity** of linear search would be O(*n*). That may seem pretty daunting, especially if we have a list that has millions of elements. However, the best case scenario is much better; if the desired element is the first in the list, we would find it in one step. Although linear search is not usually the most efficient, linear search can be useful in certain situations. Take for instance a list that you know nothing about, like a stack of papers that are out of order. It is just as efficient to search this stack linearly as it would be to search for elements randomly. In this case, we cannot search for elements in a more efficient way since we don't know anything about the list. There is no information about the list's organization for us to leverage. Now you can see why it would seem rather convenient to sort a list before searching it. Granted, sorting also takes up time and space, but this additional step will save you some time if you plan on searching a list multiple times or if you have a very large list.



Examples of Linear Search

Recall the phone book example. While looking for Mike Smith, linear search meant going through each page of the phone book one at a time. Even the algorithm where we flipped through two pages at a time can be considered linear. In fact, any algorithm in which the there is a **constant** being multiplied by the total number of elements in the list to determine the search time, is considered linear. For example, if you are turning three pages of the phone book at a time (remembering to make the correction if you overshoot), the search time would be *n/3*. Since 1/3 is being multiplied to *n*, this algorithm would be considered linear. Linear search should typically be avoided, as there is usually a better algorithm that can implemented to make the search more efficient. Sorting a list first, is one such way to search more quickly. Once a list is sorted we are able to leverage some of the concepts learned earlier to make a faster, efficient, and more elegant program.

CS50 Loops

Overview

Loops are a way for a program to execute the same code multiple times. Instead of copying and pasting the same lines back-to-back, loops allow for code to be repeated. The resulting code is better designed: if you need to change the code that gets repeated, you only need to change it once. C has multiple different types of loops: all of which can accomplish the same things, though some may be preferable to others depending on the circumstances.

Key Terms

- loops
- for loop
- while loop
- infinite loop
- do while loop

For Loops

```
1  for (int i = 0; i < 10; i++)
2  {
3      printf("hello!\n");
4  }
5  for (int j = 0; j < 10; j++)
6  {
7      printf("%i\n", j);
8  }
```

The first type of loop in C is the **for loop**. Defining a `for` loop requires three parts (included in parentheses after the word `for`, and separated by semicolons), demonstrated at left (lines 1-4).

The first part is the initialization: we create a variable `i` initially set to `0`. This variable keeps track of which iteration the for loop is currently on. Second is the condition: as long as the condition `i < 10` is `true`, everything within the curly braces will keep running. As soon as the condition is `false`, then the loop ends. The third part is the loop modification: this code is executed at the end of every loop. In this case, we modify our loop by increasing the value of `i` by `1`.

Thus, each time the loop finishes, `i` will increase in value by `1`. As soon as `i` is no longer less than `10`, the condition fails and the loop will end. The end result is that `"hello\n"` is displayed 10 times.

By taking advantage of loop modification, you can also get a loop to do something slightly different each time the loop iterates. In the second `for` loop example (lines 5-8 above), `j` is initially `0`, and so `0` is printed. Then `j` increments to `1`, and `1` is printed in the next loop iteration. This continues until `j` is no longer less than `10`. The result is that each number from `0` to `9` is printed on its own line.

While Loops

C also includes a type of loop called a **while loop**. A `while` loop checks the condition it is given: if it is true, it executes the code within the braces, and then checks the condition again. This process repeats until the condition is false. The example at right (lines 9-14) does exactly the same thing as our second `for` loop (lines 5-8): printing out the numbers from `0` to `9`.

If the `while` loop is given a condition that is always `true` (like the boolean value `true` itself), then the loop will never stop running. The example at right (lines 15-18) is an example of an **infinite loop**: since the condition will never be false, the loop will continue running indefinitely. `While` loops are particular useful when you don't know in advance how many times a loop should run.

```
 9  int k = 0;
10  while (k < 10)
11  {
12      printf("%i\n", k);
13      k++;
14  }
15  while (true)
16  {
17      printf("hello!\n");
18  }
```

Do-While Loops

```
19  int j;
20  do
21  {
22      j = get_int("Positive Number: ");
23  }
24  while (j <= 0);
```

The **do-while loop** is similar to a while loop in the sense that it repeats a loop until a condition is false. However, a `do-while` loop, unlike a `while` loop, will always execute at least once, regardless of the condition. This is often valuable in cases where user input is required: the program should definitely ask for input once, and may or may not need to ask for input more times if the input is invalid.

In the example at left, the user will be prompted to enter an integer, and will be re-prompted continuously until a positive one is provided.

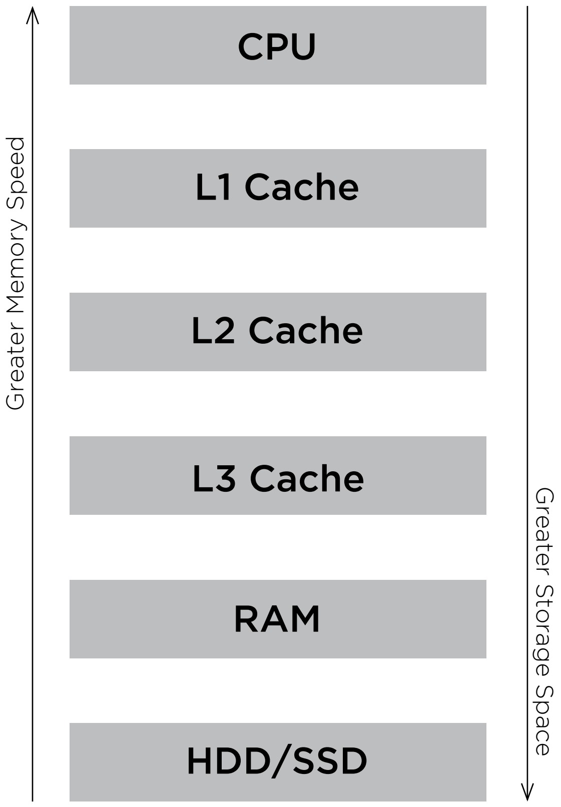
CS50 Memory

Overview

In order for computers to be able to perform computations, they need to be able to store information (as bits and bytes) into **memory**, which is just a way to keep data in a place where it can be retrieved by the computer later. However, computers don't just have one type of memory: they have multiple different types of memory, which vary in their speed, and the amount of information that they can store inside of them. Each type of memory has its own advantages and disadvantages.

Key Terms

- memory
- CPU
- HDD
- SSD
- RAM
- cache



CPU, HDD, and RAM

A computer's processor, or **CPU**, has some, but very limited memory of its own. 32-bit processors store just 32 bits in the CPU at a time, while 64-bit processors can store 64 bits of data at any given time. Although the CPU will often have to process files and data that is much larger than 32 or 64 bits, it will only manipulate and compute with 32 or 64 bit blocks at any given time, before being fed the next block. The CPU is able to process these bits extremely quickly.

On the other end of the spectrum is the hard disk drive (**HDD**), which is able to store significantly more data than the CPU. Modern consumer hard drives can store gigabytes or even terabytes worth of data. However, although HDDs can store significantly greater amounts of data in memory, it takes much longer in order to read and write data. There is also a newer kind memory that does everything a HDD does called a **SSD**. Whereas HDDs rely on a mechanical arm to read and write information, SSDs, or solid state drives, do not have any moving parts, they are consequently much faster.

Random access memory, or **RAM**, is much faster at reading and writing data than the HDD and SSD, so it is used to store the memory for applications that are currently running and files that are currently open, so that they can be accessed more quickly. However, computers generally have less RAM than they do hard drive space.

L1, L2, and L3 Cache

There are several smaller groups of memory that are faster at reading and writing information than RAM, but have less memory space available as a result. This memory is known as the L1, L2, and L3 **Cache**. The L1 Cache is the fastest (and the smallest) among the three, storing just a few kilobytes of data that can be very quickly given to the CPU for processing. The L2 cache is slightly larger, but also slightly slower than the L1 cache. The L3 cache is the largest of the three (often storing a few megabytes in memory), but also the slowest of the three. However, even the L3 cache is faster than RAM.

Tradeoffs

In general, the tradeoff for memory is one of space versus speed. The types of memory that are faster also tend to have less available. Faster memory also tends to be more expensive per unit of storage space. For instance, the price of RAM per gigabyte is greater than the price of a hard drive per gigabyte.

RAM is used by the operating system in order to run applications concurrently. If too much RAM is being used, some modern operating systems will employ "virtual memory," whereby they transfer some information from RAM to the hard drive temporarily, and retrieve it when it's needed by the user.

CS50

Merge Sort

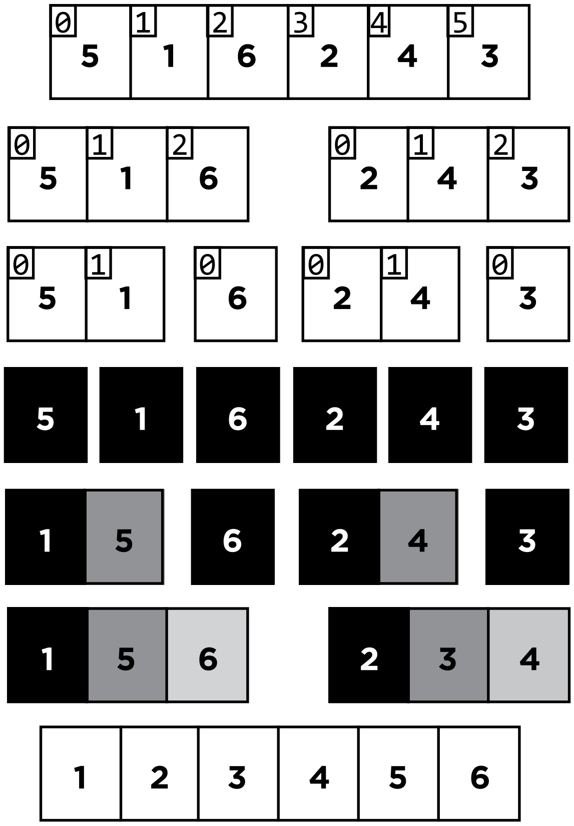
Overview

Sorting algorithms like selection sort, insertion sort, and bubble sort all suffer from the same general limitations and thus have the same worst-case runtime of *O(n²)*. **Merge sort**, on the other hand, is fundamentally different, leveraging recursion to "pass the buck" of sorting, accomplishing a drastically superior runtime: *O(n log n)*!

Key Terms

- merge sort
- array
- recursive
- pseudocode

Step-by-step process for merge sort



Implementation

Merge sort works by breaking an **array** into sub arrays and merging the subarrays back in a **recursive** way. To understand how this works, let's take a look at the following **pseudocode**:

```
merge sort:
    if number of elements < 2
        return
    else
        sort the right half
        sort the left half
        merge sorted halves
```

Using the lines above and the array on the left (containing these numbers: 5 1 6 2 4 3), we are going to sort the left and right halves of the elements and merge them together. Note that when running merge sort, we only need enough space to store two copies of the array, despite the fact that the diagram on the left appears to require more space. At this point, we have no way of sorting the right or left halves, so we are going to recursively call the merge sort function. Similarly, we are going to continue to do this until we are left with all arrays of size 1. We'll need to handle running into an odd number of elements in a consistent way. Here, we implemented our program such that the left side of the split will have one more element than the right if the array has an odd number of elements.

After the elements are broken down into arrays of size 1, we are able to merge the sorted halves, since any array of size 1 is considered sorted. When we merge the two halves, we are removing the smallest numbers from the subarrays and appending them to the merged array, repeating until all elements of both subarrays are used up. (Note: The smallest elements will always be at the beginning of the subarrays, so we only need to check the first elements in the respective subarrays.) Since 6 was a single element array in the previous iteration, it does not need to be merged. We continue to do this until all the right and left halves are sorted from the previous iteration. Upon the next iteration, when we merge the arrays back into arrays of size 3, we need only look at the 0th index of each subarray to find the smallest element of the newly-merged array. In this case, this would be 1 and 6 for the left half and 2 and 3 for the right. Since 1 and 2 are the lowest numbers of their respective sides, they go into the 0th indices of the newly-merged array. And we'll continue to merge arrays in this way until the array is fully sorted.

Sorted Arrays

Like selection sort, merge sort has the same runtime in the best and worst case scenarios. Consider running merge sort on an already sorted array: since our program would have no way of knowing that it had already been sorted, it would have carry out the entire process the same way that it would with an unsorted array.

CS50 Models and Simulations

Overview

Computers are powerful because they can make complex calculations very quickly. One of the ways we can take advantage of this power is by creating models and simulations to represent complex objects and/or events. By creating virtual representations of things we are curious about, we can increase safety, save money and time, stretch our imaginations, and ultimately learn more about the world around us.

Key Terms

- model
- simulation

Building Models and Simulations

A **model** is a static representation of an object or a system. On a computer, models are often made up of algorithms, equations, and/or visual reconstructions. A **simulation** is a dynamic representation of a model, or the running of a model. Simulations allow us to change variables of a model and observe the effects of those variable changes. Where models seek to replicate features of an object or a system, simulations seek to replicate behaviors.

Models and simulations are implemented on a computer by creating a set of parameters that define the represented system. For example, to create a simulation of a bouncing ball, you would program the mass of the ball, the force of gravity, the springiness of the ball, and the type of collision, all as mathematical equations. The computer would then be able to mimic the way a ball bounces in reality, providing an isolated and controlled reality for scientists to play with. However, more often than not, models and simulations don't attempt to represent all factors of a system. Instead, they focus on the relevant ones. Relevant factors can include factors of interest for a particular research project, or factors that have the most significant effects on a system. Models and simulations are often designed to be simplified versions of a real object or phenomena, making them easier for us to analyze and understand. Ultimately, because we input parameters of models and simulations, we have full control of what we'd like to get out of them.

Applications and Limitations

Because models and simulations essentially create virtual copies of our world, they are used in nearly every field today. Models and simulations allow people to engage in test runs without going through the real thing, providing low-risk training for less time and money. For example, simulations are used by the military to train soldiers for dangerous situations and by medical schools to train doctors for various medical procedures. Simulations can also be used in therapy for those with mental health issues. Beyond training, models and simulations can be used to learn more about a system or phenomena. Meteorologists use simulations to predict upcoming weather, economists use models to analyze markets and make predictions, and ecologists use models to better understand living systems. Even things like traffic can be modeled using simulations, helping us develop better solutions to practical problems. Models and simulations can be as large or small scale as we want. We can use simulations to analyze anything from molecular interactions to the Big Bang; the scale and detail of a model is a part of its design. Outside of research and analysis, models and simulations can be used for entertainment. Models and simulations are used in movies and video games to create computer-generated imagery (CGI), enabling us to construct realistic yet fantastical worlds and beautiful animation.

However, models and simulations are not the perfect solution to everything. While they clearly have many potential benefits, it is important to remember that models and simulations are abstractions of reality and may not be entirely accurate. Models must be calibrated, verified, and validated before they can be considered accurate, and even then, there is potential for error. In addition, there are significant costs for models and simulations. They require a significant investment to create, and depending on the complexity of the model or simulation, they may require a lot of computational power, therefore costing a lot of money. Nevertheless, computer models and simulations have been an invaluable tool in nearly all disciplines. As computers continue to become more powerful, models and simulations will too.

CS50 MVC

Overview

With anything you build, whether it's a house or a computer, there's an underlying structure that keeps the project organized. For a house, that structure may be a floor plan, designating the organization of a kitchen and a few bedrooms. Web applications are no different. One popular structure for web applications is called MVC, or model view controller. We use this model, or architectural pattern, to organize our code into parts that all have their own functions.

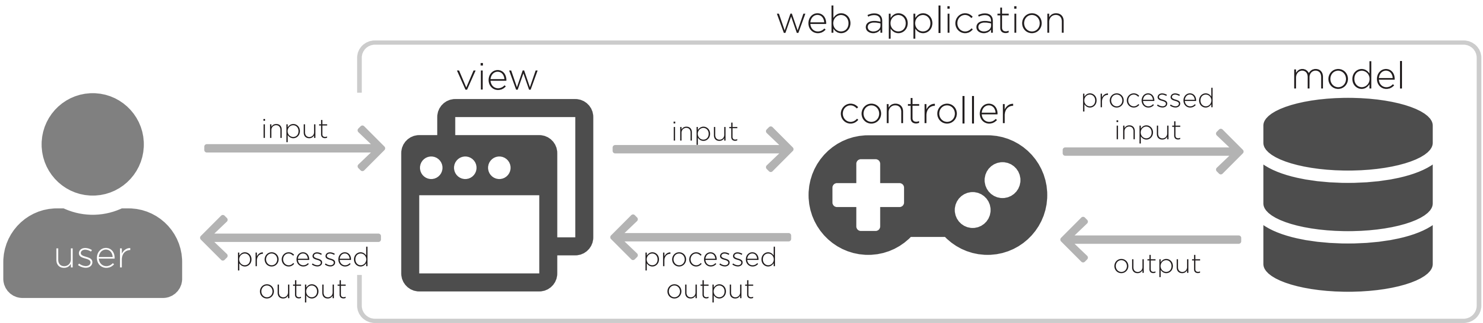
Key Terms

- MVC
- model
- view
- controller
- encapsulation

MVC

MVC is an architectural pattern that splits a web application into 3 logical components: model, view, and controller. The **model** stores and manages the application's data; this could consist of a database or some other file containing data. The **view** concerns the user-facing presentation of the application; it is application's output. The view might include files written in HTML, CSS, and Javascript. The **controller** is the logic that connects the model and the view. It is in charge of moving information between the user and the model as well as processing the given information. The controller might consist of programs written in Python and/or SQL.

For example, imagine an application that takes in a location and outputs nearby restaurants. The view would have an interface prompting the user to input a location. The location would then be passed to the controller and the controller would use the location to select the desired information from the model. In this case, the model could be a database containing a list of restaurants and their addresses. The controller would be in charge performing the necessary calculations to tell the model which restaurants it wants. Once the controller has retrieved a list of nearby restaurants from the model, the controller can send that information to the view to present to the user.



The view is responsible for enabling communication between the user and the application. May include HTML, CSS, and Javascript files.

The controller is the mastermind; it takes input from the view, processes it, uses it to send requests to the model, takes output from the model, processes it, and then passes it back it the view. May include programs written in Python and/or SQL.

The model stores data. It responds to requests from the controller to return particular data or update itself. May include a database.

Why MVC?

MVC is popularly used in industry because of its **encapsulation** of different parts of a web application. Although the model, view, and controller work together, their functions are independent of each other. Therefore, the model, view, and controller can be developed separately and simultaneously, making the MVC scalable and extensible. It also allows for the delegation of tasks among a large team of people, leading to a faster development process. Within these teams, the model, view, and controller can be independently tested. After a web application is built, MVC allows updates to be made without needing to update all parts of the application; any one part can be changed without changing the other, as long as the interactions between them remain the same. This compartmentalization also enables code reuse between different web applications.

CS50 ncurses

Overview

ncurses is a C library for writing **text-based user interfaces** that runs portably across various terminals and terminal emulators. ncurses conceives of the terminal screen as a grid of (y, x) character positions, where y counts rows down from the top of the screen and x counts rightwards. It provides an application programing interface (**API**), which is a series of functions for manipulating the terminal screen. Hundreds of terminal-based applications use ncurses.

Key Terms

- ncurses
- text-based user interface
- API
- header file

Using ncurses

To use functions from any library in C, we need to `#include` the **header file** at the top of our source code file. In this case, we'll use `#include <ncurses.h>`. When compiling our code, we'll also have to link the library (with `-lncurses` when working with ncurses, typcially in a file called a Makefile, which tells `make` what to do) so that the resulting object code knows how to execute the functions.

Some Common ncurses Functions

`initscr()` takes no arguments, but, as a side effect, initializes all the appropriate data structures and flushes the screen.
`endwin()` is the antidote to initscr, this time quitting `ncurses` and returning the terminal window to its state preceeding the program.
`getch()` takes no input, but returns a character typed in at runtime by the user.
`move()` moves the cursor to a `(y, x)` position on the screen.
`addch()` adds a character at the cursor's location.
`mvaddch()` takes a `(y, x)` position and a character, combining the above two functions into one.
`mvaddstr()` takes a `(y, x)` position and a string, writing the start of the string at that position and going rightwards.

An example

In the code below, we first initialize ncurses with `initscr()`. Then we use `raw()` to prevent the terminal from buffering the characters that a user may type. That way, the progam can detect as soon as a user types even a single character.

The `for` loop takes care of each line in the diagonal, one by one. Notice how we can use move and addch equivalently to mvaddch. For entire strings, the mvaddstr function can be used.

Notice how we place the characters or strings at `(y, x)` indices, where `y` counts rows down and `x` counts rightward.

This different coordinate system makes more sense for programs that read left to right, top to bottom.

```c
// initialize ncurses
initscr();
raw();

for (int i = 0; i < 10; i++)
{
    move(i, i + 2);
    addch('*');
    mvaddstr(i, i + 11, "/////");
    mvaddch(i, i + 24, '*');
}
// quit on any input
getch();

// close ncurses
endwin();
return 0;
```

```
     $ make diagonal
     $ ./diagonal
  *        /////        *
   *        /////        *
    *        /////        *
     *        /////        *
      *        /////        *
       *        /////        *
        *        /////        *
         *        /////        *
          *        /////        *
           *        /////        *
```

Operators

Overview

You're probably familiar with **operators** from math: the + symbol means addition, the - symbol means subtraction, etc. C also has operators, which you can use to modify or combine values. In addition to having operators that perform basic mathematical operations like addition, subtraction, multiplication, and division, C also has operators that perform other functions: like finding the remainder when dividing, or updating the value of a variable.

Key Terms

- operator
- arithmetic operators
- assignment operators

Arithmetic Operators

C's **arithmetic operators** perform mathematical functions on numbers. The `+` operator adds two numbers, the `-` operator subtracts one number from another, the `*` operator multiplies two numbers, and the `/` symbol divides one number by another. See lines 1 through 4 of the code to the left to see how such operators work.

When working with `int`s and dividing, it's especially important to be aware that an `int` cannot store non-integer values. For instance, in line 5, we try to store the value of `10 / 3`. C sees a division of two integers, and tries to make the result an integer as well. But since the "real" value of `10 / 3` isn't a whole number, everything after the decimal gets cut off (or "truncated") and `e` is set to just `3`. In order to save the value with the decimal included, we would need to use floating-point numbers, like `float e = 10.0 / 3.0`.

C has another operator, `%`, which is called the modulus operator. The modulus operator gives us the remainder when the number on the left of the operator is divided by the number on the right. Line 6 demonstrates the modulus operator: the remainder when dividing `13` by `3` is `1`, so the value of `f` is set to `1`.

1	`int a = 2 + 8;`	a	10
2	`int b = 10 - 3;`	b	7
3	`int c = 4 * 7;`	c	28
4	`int d = 10 / 2;`	d	5
5	`int e = 10 / 3;`	e	3
6	`int f = 13 % 3;`	f	1

Assignment Operators

C also provides **assignment operators**, which provide a variety of ways to update the value of a variable. The standard assignment operator (`=`) is demonstrated on line 7: it sets the value of `e` to be equal to whatever's on the right side of the equals sign: in this case, the current value of `f` added to `1`.

The variable you're assigning can also be on the right of the equals sign itself. On line 8, the value of `e` is set to the existing value of `e` plus one. While `e = e + 1` might not make logical sense in algebra, it's valid in C. Updating the value of a variable based on its existing value is so common that C has special syntax for it: the operators `+=`, `-=`, `*=`, and `/=` will set a variable to its existing value plus, minus, multiplied by, or divided by some other number.

C also includes special syntax for increasing the value of a variable by one or decreasing the value of a variable by one, by writing the name of the variable followed by `++` or `--`. So a statement like `e++` on line 11 takes the value of `e` and increases it by 1.

7	`e = f + 1;`	e	2
8	`e = e + 1;`	e	3
9	`e += 1;`	e	4
10	`e *= 7;`	e	28
11	`e++;`	e	29

CS50 Python

Overview

There are many programming languages used in the world of computer science. Many of these languages can accomplish equivalent tasks, but programmers choose a language depending on the project they're working on. Think of different programming languages as different kinds of shoes. Shoes all share core functionality: protecting your feet from the ground. But while athletic shoes are great for running, they are much worse in the cold than a pair of warm boots. Similarly, while programming languages all share the same basic functionality, each emphasizes particular features that are optimal for specific uses.

Key Terms

- Python
- high-level
- low-level
- list
- tuple
- dict
- dynamically typed

Python

Python is a **high-level** programming language developed to be easy to learn, easy to read, and broadly applicable. By abstracting away **low-level** technical details like memory management, Python reads more similarly to a natural language than a low-level language like C. Python is also different from C in that it is interpreted at run time rather than compiled to machine code beforehand. This allows us to run a single command (for example, `python hello.py`) to run a program. Because of its convenience, Python is widely used and compatible with other technologies such as databases, graphical user interfaces (GUIs), and web programming.

Syntax

```
from cs50 import get_string
```

```
def main():
    print("hello, world")

if __name__ == "__main__":
    main()
```

```
while True:
    print("hello, world")
```

```
for i in range(50):
    print("hello, world")
```

```
if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x is equal to y")
```

```
import sys

for s in sys.argv:
    print(s)
```

Built-in data types

list - an ordered and changeable collection of items (can be updated and length can be changed)

```
>>> mylist = ["foo", "bar"] create a new list
>>> mylist.append("baz") append "baz" to mylist
>>> mylist show mylist
["foo", "bar", "baz"] value of mylist
```

tuple - an ordered and unchangeable collection of items

```
>>> mytuple = ("foo", "bar", "baz") create a new tuple
>>> mytuple[0] show the value at the 0 index of mytuple
"foo" the value at the 0 index of mytuple
```

dict (dictionary) - an unordered, changeable list of key value pairs in which the key is the index to access a value

```
>>> fruitcolor = {"apple": 3, "lime": 10} create a new dict
>>> fruitcolor["apple"] show the value of "apple"
"red" value of "apple"
```

Things to Remember

- Tabs and new lines are used to denote the end of commands and functions, no semicolons here!
- Python uses colons similar to the way we use open curly braces ({) in C, but these should be on the same line as the code, not a line all by themselves.
- Python is a **dynamically typed** language, meaning it infers data types at the time of assignment. + acts as both arithmetic addition and string concatenation depending on the variable types.
- Python comes with more functions out of the box than C. Additionally, there are lots of libraries that exist for Python so it is typically a good idea to check if functions exist in Python's many libraries before you implement them yourself.

CS50 Python for Web Programming

Overview

In addition to using Python for writing local programs and algorithms, Python is often used for web programming. In web programming, Python is used for **server-side scripting**. In other words, it's used on the back end of a web application to implement web servers. While Python is one of many languages used for back end web programming, its readability, simplicity, and convenience all contribute to its popularity. In particular, Python has a built in HTTP server library called http.server that includes functions for listening, managing, and responding to HTTP requests. Because Python has a history of being used for web programming, many external web programming tools are built using and compatible with Python, such as Django and Flask. Python also offers easy integration with other tools and programming languages.

Key Terms

- server-side scripting
- Flask
- web framework
- micro framework
- Jinja

Flask

Flask is a micro web framework written in Python that provides programmers with tools to easily and quickly implement web applications. A **web framework** is software that provides tools, libraries, and extra technologies to build web applications, and a **micro framework** is a framework that is not highly dependent upon external resources. Because of tools like Flask, it is unnecessary for web programmers today to build web servers from scratch. Instead, by abstracting away lower level details, web programmers can focus on the logic of their specific web applications.

The code on the right shows the Python file of a simple web application. In the first line, we import some functionality from the Flask package. Then, we use Flask syntax create a new web application, allowing Flask to set up some low level details. Below that, we tell our applciation to go to the function index when the "/" route is requested. Within index, we've called the Flask function render_template to send the file index.html to our user's browser.

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route(“/”)
def index():
    return render_template(“index.html”)
```

As you can see, building a web application with Flask is incredibly simple. In addition to this basic functionality, Flask offers many other features that are useful for building web applications. To use these features, simply check out Flask's documentation.

Jinja

Jinja is a template engine built into Flask. One of its features is a templating language that allows you to use dynamic elements (variables, loops, conditions) in static HTML/CSS files. Jinja expressions are very similar to Python expressions, making Jinja even more convenient to use. Jinja also enables inheritance of HTML/CSS files, minimizing rewritten code.

In the code on the left, the file message.html is using Jinja to inherit code from layout.html. Inheritance of templates demonstrates better programming design because it reduces repeated code, maintains consistency, and makes templates more convenient to update.

Like Flask, you can learn more about features of Jinja by looking at the Jinja documentation.

layout.html

```
<!DOCTYPE html>

<html>
    <head>
        <title>Hello</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

message.html

```
{% extends “layout.html” %}

{% block body %}
    Hello, world!
{% endblock %}
```

CS50

Pseudocode

Overview

Computer programs are generally written in a **programming language**, which is a formal computer language used to provide instructions for a machine. Programming languages require that code be written in a very particular syntax. To express algorithms without using a programming language, many computer scientists will instead choose to use **pseudocode**, which is a programming tool that lets us present algorithms in a natural language.

Key Terms

- programming language
- pseudocode
- assignment
- indent

```
let n = 0
for each person in room
    set n = n + 1
```

```
1 stand up
2 assign yourself the number 1
3 until only one person remains standing
4     pair off with someone else standing
5     add your numbers together
6     assign yourself the new number
7     choose one member of the pair to sit
8     if you are chosen
9         sit down and do nothing else
```

An Example of Pseudocode

Consider how we might write an algorithm to count the number of people in a room. We might start by thinking of the number 0, and then for each person in the room, think of the number one greater than the one we're currently thinking of.

The first block of pseudocode to the left expresses this idea. It's not written in a programming language, but it's described formally so that the steps are very precise and clear. We start by giving a name, like `n`, a value: `0`. This process is called **assignment**, and we use it so that we can refer to our value by name later in our code. Now, for each person in the room, we can re-assign `n` to be `n + 1`, so that the value increases by one for each person. Now, at the end of the algorithm, `n` is the number of people in the room.

We can try another method of counting the number of people in the room, to demonstrate some more complicated (but more efficient) pseudocode. Start by having everyone stand up, and assign themselves the number 1. Now, everyone pairs off with someone else standing, adds their numbers together, and then one person sits down. If this process repeats until there's only one person left standing in the room, then the number that they have been assigned should be the total number of people in the room.

This algorithm is expressed by the second block of pseudocode above. Notice how, while not expressed in a programming language, the algorithm is still precise. The code also **indents** some of the lines (shifts them to the right by a certain amount of space) to show what blocks of code go with which statements. For instance, since lines 4-9 are all indented, it's a sign that all of those lines should repeat 'until only one person remains standing,' per the instructions on line 3. Likewise, since 'sit down and do nothing else' on line 9 is indented, it's a sign that it should happen 'if you are chosen' per the instructions on line 8.

Elements of Pseudocode

There's no one correct way to write pseudocode. Sometimes your pseudocode will be more or less detailed, depending on what your purpose is. Unlike a programming language, there's also no defined syntax for how pseudocode needs to be written.

There are, however, some elements that are likely to reoccur in pseudocode. Pseudocode will often involve concepts like assigning values—such as in the above examples. Pseudocode may also contain conditions (where certain code blocks will only be executed under certain conditions) as well as loops (where certain code blocks will be repeated a number of times). These concepts, which can be represented in pseudocode, are also important concepts for programming in a programming language. Even after you've learned a programming language, pseudocode can still be helpful for expressing the steps of an algorithm without having to worry about syntax, so that you can better understand your program's logic.

CS50 Principles of Good Design

Overview

Design is a very important aspect of programming and product development. Good design differentiates programs that work from programs that work well. Programs with robust, consistent, and nonrepetitive code are generally considered to be well-designed. Other measures of design are program **efficiency** and modularization. In order produce portable, scalable, and reusable code, we must keep design in mind while programming.

Key Terms

- efficiency
- magic numbers
- tradeoffs

Loops and Conditionals

Loops are very powerful and often used in programming. However, as they are somewhat costly, we should make sure we use them efficiently. We can check that we are doing so by asking ourselves the following questions: Are each of my loops essential? Can I combine any loops? And am I taking advantage of every iteration of my loops?

Along similar lines, it's important to use conditionals (if, else if, and else) efficiently. Consider a program that takes in a birth month and outputs a corresponding birthstone. We could implement it by checking if the user inputted `"january"`, then checking if the user inputted `"february"`, and so on until we reach `"december"`. But, if we already know that a user inputted `"january"`, why bother checking any of the other months? In this case, we could improve our program's design by using else if statements or switch statements instead of all if statements.

Constants

Magic numbers are hard-coded constants in code. We consider using them to be bad design since they reduce the scalability and readability of code. Furthermore, making changes to hard-coded values must be done manually. Using variables instead can facilitate making such changes. Additionally, we can use `#define` to define constants that will not change, like the number of letters in the alphabet (26) or the value of a nickel in cents (5). We do this at the beginning of our code with `#define` after our header files and outside of our main function.

Functions

It's typically good design to break code out into functions when needed. For instance, if we were performing the same set of mathematical operations to multiple different values, it might make sense to write the set of operations as a function and simply call that function multiple times. Similarly, it's also a good idea to break really long code into different files, linking between them so they can all work together smoothly. In these ways, we can make code that could otherwise be very tedious and complicated to get through be easier to make sense of.

Tradeoffs

Design is subjective and debatable. What one programmer may think is better design, another might fundamentally disagree with. For instance, someone could write code using an uncommon function that makes the program shorter and more concise. However, a person that had never seen the function before and had to look up its documentation could very well argue that the program was not written clearly.

At right are two different implementations for an algorithm that takes in a number of t-shirts and tells us how many boxes we need to store them, if our options are boxes that fit 12 shirts, 10 shirts, 3 shirts, or 1 shirt. Which is best designed? Well, different programmers could argue in favor of either one, since each come with their own set of **tradeoffs**. What do you think?

```
num_boxes += (shirts_left / 12);
shirts_left %= 12;
num_boxes += (shirts_left / 10);
shirts_left %= 10;
num_boxes += (shirts_left / 3);
shirts_left %= 3;
num_boxes += shirts_left;
```

```
int boxes[] = {12, 10, 3, 1};
for (int i = 0; i < 3; i++)
{
   num_boxes += (shirts_left / boxes[i]);
   shirts_left %= boxes[i];
}
```

CS50

Routers

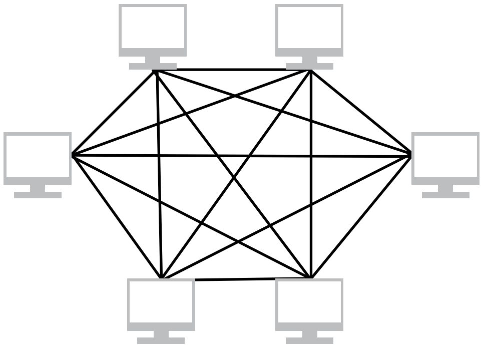
Overview

The Internet allows information to be sent from one device to another. To facilitate this process of passing data between computers, the Internet makes use of **routers**, which direct packages of data across various networks. Routers follow a very specific set of instructions in order to ensure that the data they are routing across the Internet ends up at the correct location.

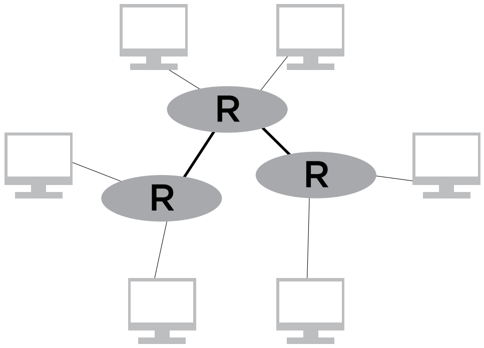
Key Terms

- router
- routing table

Network Without Routers



Network With Routers



The Routing Model

Devices on the Internet need to be able to communicate with other Internet-connected devices. One way to organize such a network is as per the diagram on the left, where every computer on the network is physically connected to every other computer on the Internet.

Such a model would certainly be fast, since to get information from one computer to another, it could be sent directly to its destination. However, such a model would require an infeasible number of physical connections. Note how complicated this connection web is, even with just six computers connected to the Internet. Imagine what it would look like with millions, or even billions, of Internet-connected devices! Clearly, having every computer physically connected to every other computer is not at all reasonable.

Instead, the Internet makes use of routers. Routers act as intermediaries between devices on the Internet. Every computer is connected to a router, and each router is connected to other routers. You can think of router as a post office. A package gets sent from post office to post office until it reaches the post office closest to its destination. Just as anyone can send anyone else a package, every computer on the Internet can communicate with every other computer through these routers – just not directly.

Computers can send information to one other on the Internet by passing data through one or more routers. This data is sent in packets, which travel through the Internet via routers, getting passed along from router to router until reaching their final destination: the router which is connected to the destination computer.

Routing Tables

Routers are programmed with instructions on how to figure out where to send each packet of data based on the destination's IP address. These instructions are often stored in what's known as a **routing table**. Routers can discern, based on the initial digits of an IP address, the direction in which packets need to be sent.

But routers don't need to have information about the exact overall path the data packet needs to take in order to get to its destination: the router just sends the packet one step closer to the destination and then lets the next router take care of the rest. Furthermore, there often won't be just one route that data must take in order to get from one location on the Internet to another. Routers will frequently move different packets of data across different routes, even if they are intended for the same location.

CS50

Recursion

Overview

Recursive solutions to problems are typically contrasted with iterative ones. In a **recursive solution**, a function (or a set of functions) repeatedly invokes slightly modified instances of itself, with each subsequent instance tending closer and closer to a base case. In the meantime, the intermediate calls are all left waiting, having "passed the buck" to a downstream call to give it the answer it needs. Recursive procedures, when contrasted with iterative ones, can sometimes lead to incredibly efficient, elegant, and, some might even say, beautiful solutions.

Key Terms

- recursive solution
- iterative solution
- base case
- recursive case
- call stack
- active frame

Recursion versus Iteration

Recursive solutions can often replace clunkier iterative ones. One great example of this is with programs that calculate the factorial of a number. Remember that "n factorial," or `n!`, simply represents the product of all integers less than and including `n`. So `3!` would be six (`3 * 2 * 1`). Consider the two approaches on the right for implementing a function to find the factorial of an integer. The first implementation looks familiar. This is known as an **iterative solution** as we are iterating through a loop, substituting in different values for `i`. In this solution, we have declared two variables (`product` and `i`), while in the recursive solution, we have not declared any. Also, note that `recurse()` is fewer lines shorter than `iterate()`.

```
int iterate(input)
{
    int product = 1;
    for(int i = input; i > 0; i--)
    {
        product *= i;
    }
    return product;
}
```

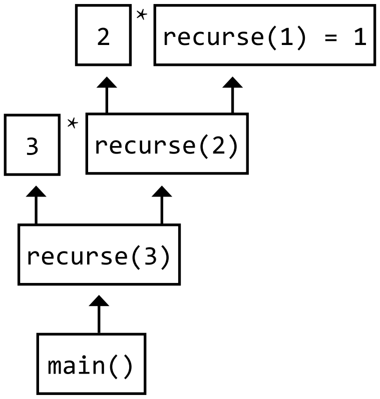
```
int recurse(input)
{
    if(input == 1)
    {
        return 1;
    }
    return input * recurse(input - 1);
}
```

Implementation

Recursive solutions to problems are made up of two parts: the base case and the recursive case. The **base case** is what allows us to break out of an infinite loop. Without a base case our program would continue to run until it no longer had the space to do so and resulted in a segmentation fault. In our example, the base case is when input == 1. The **recursive case** is where the function invokes itself. This appears in the last line of `recurse()`, where recurse is called again. In this way, recurse repeatedly calls itself until `1` is the value being passed into the function.

Call Stack Representation

When a recursive function (or any function for this matter) is called, it creates a new frame on the stack. Every subsequent function called within `main()` is created on top of the previous frame. This stack, where all of our function calls exist, is called the **call stack**. This means that the function at the top of the stack is the most recently called function. We call this the **active frame**. Say we pass in 3 as the input for `recurse()`, then `main()` will call `recurse(3)`, which will call `recurse(2)`, and so on. This process will continue to occur until the base case is met. Once this happens, that return value trickles down and is plugged back into the function calls left open in the call stack. In our example, 1 would be plugged into `recurse(1)`, destroying this frame in the call stack and leaving `recurse(2)` as the active frame. The number 2 would be passed into `recurse(2)` in the same way and so on until `recurse(3)` returned 6 and passed that back to `main()`. At this point, `main()` would be the only function left in the call stack, since all the other calls to `recurse()` would have been destroyed after returning a value.



Note that in our sample iterative solution above, there would only be one function called, `iterate()`. So in some ways iterative solutions are simpler than recursive ones. And since iterative solutions can usually solve the same types of problems as recursive ones, there will almost never be a real world problem that requires us to use recursion as a means to solve it. Rather, recursion can be used to make our code more elegant and efficient.

CS50

Syntax

Overview

C is a programming language with which you can write programs. Programming languages like C require you to write using a very specific **syntax**: a set of rules that describe how to arrange words and symbols (like brackets and parentheses) in order to write working statements that together can form a complete program. C's syntax might seem complicated at first, but with practice, the syntax of the language will start to become second nature to you.

Key Terms

- C
- syntax
- function
- string
- compile

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("hello, world\n");
6  }
```

Your First C Program

The code to the left is an example of a simple program in C which displays "hello, world" in the terminal window when it runs. Line numbers have been added to the left side of each line for reference, but they shouldn't be included in the code itself.

On line 1, `#include <stdio.h>`, tells your program to access a set of pre-written **functions** stored in a file called `stdio.h`, where a function is a collection of programming statements that performs a particular task. By including `stdio.h`, your program can now take advantage of code that people have already written in the past: in particular, a function called `printf`, which displays text on your screen.

On line 3, `int main(void)` defines a function which acts as the beginning of your program, serving the equivalent of the "When Green Flag Clicked" button in Scratch. When your C program runs, it will look for the `main` function to know where to start. The curly braces on lines 4 and 6 hold the code of the `main` function together. Anything inside of the curly braces is therefore a part of the `main` function.

In this program, the `main` function has just one programming statement: `printf("hello, world\n")`. `printf` is a function (that was written in `stdio.h`) which displays a **string** (which is just a fancy way of saying "text") on the screen. In C, strings are always surrounded by double quotation marks.

Within the parentheses for `printf`, we've provided `printf` with a string as input, so that `printf` knows what string to display on the screen. In this case, our string is `"hello, world\n"`. The `\n` character tells `printf` to display a new line. The result of displaying `"hello, world\n"`, then, is to print out the words `hello, world` on the screen, followed by a new line. Finally, at the end of line 5 is a semicolon (`;`), which is C's way of defining the end of a programming instruction.

Compile and Run Your Program

Now that you've written your program, it should be saved in a file (typically ending with `.c`). In this case, we might call our file `hello.c`. This is your source code file. However, computers can't understand C code directly: remember that computers can only understand sequences of 0s and 1s. First, we need to **compile** our program: converting it from source code to object code, which is just sequences of 0s and 1s. Once the source code is compiled into object code, it can be executed. Any lines of code that begin with `//` or are enclosed with `/*` and `*/` will be ignored by the compiler: these lines are called comments, and will frequently be used by programmers to document their code so that people reading the code later (including themselves) can understand what's happening in the code.

Several compilers exist, including `clang` and `gcc`. We can also compile our program by typing the command `make hello` at the command line, which uses the `clang` compiler. If the program compiles successfully (without errors), then we can run the program by typing `./hello` at the command line. If all goes well, `hello, world` should be printed to the screen.

CS50 Structures and Encapsulation

Overview

At a certain point, the usual suspect data types no longer suffice for the kind of work we need to do. Rather, we need to be able to encapsulate data more broadly, allowing us to group related information together. For example, students have names (probably represented by strings), ages (probably represented by integers), and grade-point averages (probably represented by floating-point numbers)--but none of those things matter independently. Instead, all of these things come together and are part of some larger overall entity: the student. Wouldn't it be nice to be able to "bundle" these things together, perhaps allowing us to abstract away some of the underlying specifics? In more modern programming languages, we might do this with a so-called object, but in C, we have a more basic mechanism for this: the **data structure**.

Key Terms

- data structure
- struct
- member

Arrays versus Structs

```
#define STUDENTS 3
string names[STUDENTS];
int classyears[STUDENTS];
float gpas[STUDENTS];
```

```
typedef struct
{
  string name;
  int year;
  float gpa;
}
student;
```

Up until now, if we wanted to group data together, we were limited to an array, each element in which needed to be of the same type. Furthermore, we had to declare the size of the array beforehand. To create a group of variables related to students using arrays, each variable needs to be its own array. And to increase or decrease the amount of students, we need to change the `#define STUDENTS` line accordingly. One advantage of this setup is that, so long as we know the index associated with them, we can directly access every student.

Another way to group data together is with a **struct**. Structs allow us to make new data types out of existing ones. Here we created a type student that has a string, an int, and a float associated with it. We will refer to these as **members**. In this way, we can refer to a specific member of the student type via the line `student.member`, where "member" is the name of whichever member we want to access. A tradeoff of using structs is that we cannot iterate through each field like we can in arrays. In C, arrays are static; so too are the fields in structs. These attributes, such as a name or a year, must be defined. One of the main advantages of storing data in a struct is that we can group data of different types together. Another benefit is that we don't have to declare how many 'students' there will be. Remember that in our earlier implementation of an array, we had to include a `#define` line, but in structs, we can have as many students as we like without having to define that number somewhere in our code.

Implementing Structs

In the lines of code above, we defined a new type called 'student'. Similar to how int is a type, so too is student a type – once we define it. We can now pass variables of type 'student' into functions or into other structs. To create a new a new variable of type 'student', we need to write a line similar to what we would write if we needed to declare a variable of type int: `student s1 = {‘Zamyla’, 2014, 4.0}`. Now, to access s1's gpa, we can type `s1.gpa`. To pass s1 to a function, let's again look back at how we would do so with ints. For example, `int foo(student x)`, is valid. Similarly, to pass in Zamyla's student information, we can write `foo(s1)`. And if the function takes an argument of type int, we could simply pass in just the year member of s1 by using the line `function(s1.year)`.

CS50

SQL

Overview

Imagine that you have a sheet of paper with someone's username on it and a separate sheet of paper with the matching password. It wouldn't be very difficult to keep them together. But now imagine that you have thousands of these papers, all with corresponding pieces of information. How would you keep them organized? That's where databases come in. A **database** is a program that stores data in an easily accessible, manageable, and updatable form. By organizing data in a database, programs can effectively and efficiently keep track of enormous amounts of information.

Key Terms

- database
- field
- record
- primary key
- persistent
- SQL

Databases

id	name	age	favorite food
1	Elphie	2	vanilla ice cream
2	Milo	6	duck dog treat
3	Mochi	3	mochi

Databases look very similar to spreadsheets, like those found in Excel or Google Sheets. The data is organized in a table in **fields** (columns) and **records** (rows). Fields describe the data in a column and records link related pieces of information. In the example on the right we've stored the names, ages, and favorite foods of a few dogs. The field names of this table are id, name, age, and favorite food. Each row ties pieces of information together; we know that Elphie is age 2, and loves vanilla ice cream. In this table, the id is set to be the **primary key**, or a unique identifier for each record. Each table can only have one primary key. Similar to variables, each field has a particular data type. In this table, id and age are of the type INTEGER, and name and favorite food are of the type TEXT. There are also other SQL data types, such as BLOB (binary data), NULL (no value), REAL (floating-point value), DATETIME (dates and times), and NUMERIC (any kind of number).

Say we also wanted to store the number of calories of these favorite foods. Instead of recreating my table to have an additional field, we can create a new database that stores a list of foods with their corresponding calorie count. Since these tables have the same foods on them, the information from both tables can be linked together. Databases work with programs, but they are separate files from your code. For this reason, databases are **persistent**: any changes made to the database remain after the program exits.

SQL

Structured Query Language, or **SQL**, is the standard language for managing, or "talking" with, a database. Using SQL, we can request, search, and filter data from our database. Take a look at these common commands for manipulating databases:

```
CREATE TABLE ‘dogs’ (‘id’ INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, ‘name’ TEXT, ‘age’ INTEGER, ‘favorite food’ TEXT)
```

This creates a table named dogs with specified fields and data types. The id is set to be the primary key and is automatically created when a new record is inserted. This way, we will never have a record with no id and no ids will ever repeat.

```
INSERT INTO “dogs” (“name”, “age”, “favorite food”)
VALUES (“Willow”, 4, “watermelon”)
```

The INSERT command inputs data for a new record, specifying field names and their corresponding values. Because we set id to be autoincremented, we don't have to worry about assigning it ourselves.

```
SELECT * FROM “dogs”
```

The asterisk (*) means all in SQL. This allows us to select all records from dogs.

```
UPDATE “dogs” SET “age” = 2
WHERE id = 4
```

This updates the name of the record with the id 4. The primary key comes in handy whenever we need to retrieve a particular record.

```
DELETE FROM “dogs” WHERE id = 4
```

Delete the record at id 4

Selection Sort

Overview

Sorted arrays are typically easier to search than unsorted arrays. One algorithm to sort is bubble sort. Intuitively, it seemed that there were lots of swaps involved; but perhaps there is another way? **Selection sort** is another sorting algorithm that minimizes the amount of swaps made (at least compared to bubble sort). Like any optimization, everything comes at a cost. While this algorithm may not have to make as many swaps, it does increase the amount of comparing required to sort a single element.

Key Terms

- selection sort
- array
- pseudocode

Implementation

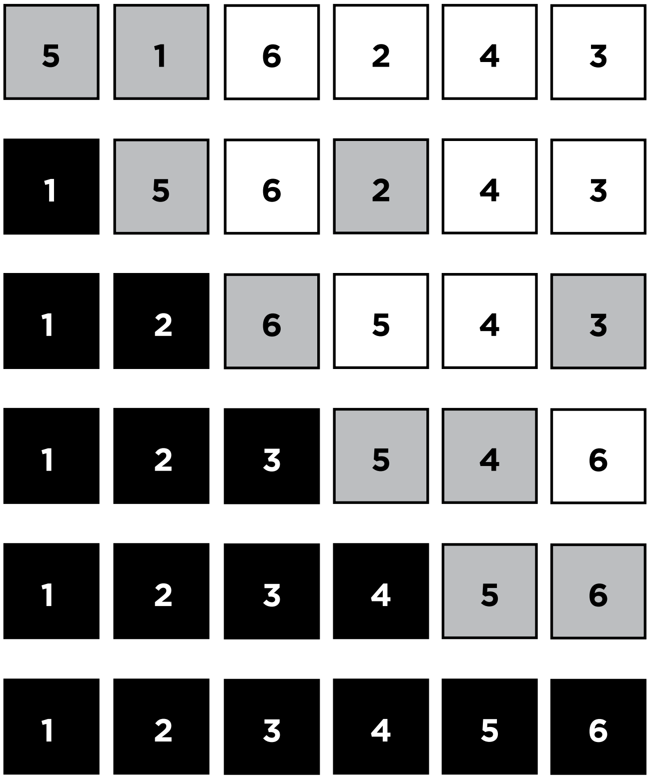
Selection sort works by splitting the array into two parts: a sorted **array** and an unsorted array. If we are given an array of the numbers 5, 1, 6, 2, 4, and 3 and we wanted to sort it using selection sort, our **pseudocode** might look something like this:

```
repeat for the amount of elements in the array
    find the smallest unsorted value
    swap that value with the first unsorted value
```

When this is implemented on the example array, the program would start at `array[0]` (which is 5). We would then compare every number to its right (1, 6, 2, 4, and 3), to find the smallest element. Finding that 1 is the smallest, it gets swapped with the element at the current position. Now 1 is in the sorted part of the array and 5, 6, 2, 4, and 3 are still unsorted. Next is `array[1]`, 5 is our current element and we need to check all elements to the right to check for the smallest (note that by only checking to the right we are only looking at the unsored array). Finding that 2 is the smallest element of the unsorted array we swap it with 5, and so on.

Notice that in the second-to-last iteration, we can clearly see that the array is now sorted, but a computer cannot look at the larger picture like we can. It can only process the information directly in front of it, and so therefore, we continue the process. Once 5 is recognized as the smallest element of the unsorted array, only then can the algorithm be stopped, since the "unsorted" array is of size one and any list of size 1 is necessarily sorted.

Step-by-step process for selection sort



5	1	6	2	4	3
1	5	6	2	4	3
1	2	6	5	4	3
1	2	3	5	4	6
1	2	3	4	5	6
1	2	3	4	5	6

Sorted Arrays

Unlike bubble sort, it is not necessary to keep a counter of how many swaps were made. To optimize this algorithm, it might seem like a good idea to check if the entire array is sorted after every successful swap to avoid what happened in the last two steps of the pseudocode above. This process too, comes at a cost, that is even more comparisons that we have to make. We are guaranteed though, that no matter the order of the original array, a sorted array can be formed after *n-1* swaps, which is significantly fewer than that of bubble sort. In selection sort, in the worst case scenario, n^2 comparions and *n-1* swaps are made. Unfortunately, that's also the case in the best-case scenario!

CS50

Typecasting

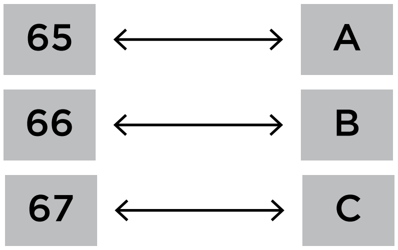
Overview

Recall that C has several different data types, including `int`s, `float`s, and `char`s. It may sometimes be necessary to convert variables from one data type to another data type. C allows us to do this via **typecasting** (or just "casting"). Typecasting allows you to cast data from one type to another type which is equally or less precise, but you cannot cast data from a type that is less precise to a type that is more precise.

Key Terms

- typecasting
- explicit typecasting
- implicit typecasting

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x = 65;
6      printf("%i\n", x); // 65
7      printf("%c\n", (char) x); // A
6  }
```



Chars and Ints

The ASCII standard, as you may recall, gives every letter a unique number to identify it: capital A is represented by the number 65, capital B by 66, and so on. Using typecasting, we can convert between integer values and `char` values.

Say, for instance, that we assigned an integer variable `x` to hold the value `65`. If we were to print the variable out on the screen (like on line 6 of the code to the left), then it would display the number `65` to the console.

On line 7, however, we've included a placeholder for a `char` instead of an `int` (as denoted by the `%c` symbol). We're still passing in `x` as an argument, but the code first casts `x` into a `char`. This is done by writing `(char)` in parentheses before the name of the variable. Placing a new type name in front of an existing variable to evaluate the variable as a different type is called **explicitly typecasting**: we are directly providing instructions to convert types.

While explicit typecasting in this situation is good practice from a style perspective (so that people reading your code can better understand what's happening), it's not actually necessary. If we were to exclude the `(char)` symbol from before the `x` in line 7 of the above code, the code will still print out the letter `A` (the ASCII mapping of the value `65`). Since we've included a placeholder for a `char`, the compiler is expecting a `char` to be passed in. If we pass an `int` in, the compiler will automatically try to interpret the value as a `char` instead. This is called **implicit typecasting**.

Ints and Floats

Typecasting is also valuable for converting between floating-point numbers and integers. Take the example at right. On line 6, we might want `b` to store the value of 28 divided by 5, which is 2.4. But line 6 actually sets `b` to be `2.0`. This is because the compiler sees a division between two `int`s, and thus presents the answer as an `int`, even though we're storing the value inside of a `float`. To get around this, we can first explicitly cast `a` to be a float, and then perform the division, as is done on line 7. In this case, `c` now correctly equals `2.4`.

Implicit typecasting can also be valuable when dealing with `int`s and `float`s. Since `int`s do not store digits past the decimal point, typecasting a `float` to an `int` is an easy way to truncate a number into an integer. On line 10 to the right, when we try to assign an `int` to be the floating-point value `d`, `d` is implicitly cast to be an `int`, getting rid of everything after the decimal point. The value of `e` is now `28`.

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int a = 28;
6       float b = a / 5;
7       float c = (float) a / 5;
8
9       float d = 28.523;
10      int e = d;
11  }
```

CS50 Trust Models

Overview

Downloading a piece of software from the Internet requires a substantial amount of trust on part of the user. The user must trust that the piece of software that is being downloaded doesn't contain malicious code. In theory, any software downloaded onto a computer could delete all of the files on that computer. Yet, we still trust that the software we download is safe and secure. This is the basis of **trust models**.

Key Terms

- trust model
- backdoor

Backdoors

To the right is an excerpt of a hypothetical login program written in C, which checks a username and password to determine whether a user's account credentials are valid. In reality, login programs would probably compare the user's inputs against username and password values stored in a database. Furthermore, these would most likely be encrypted in some way and not just stored as plain text. Still, we'll use this simplified version for the sake of example.

```
if ((strcmp(username, "rob") == 0 &&
       strcmp(password, "thisiscs50") == 0) ||
    (strcmp(username, "tommy") == 0 &&
       strcmp(password, "i<3javascript") == 0))
{
    printf("Success!! You now have access.\n");
}
else if (strcmp(username, "hacker") == 0 &&
       strcmp(password, "LOLihackedyou") == 0)
{
    printf("Hacked!! You now have access.\n");
}
else
{
    printf("Invalid login.\n");
}
```

Notice that after performing the initial check for username and password combinations, the code offers an additional way to gain access to the system (by using the username "hacker" and the password "LOLihackedyou"). This method of accessing a system through an alternate means, one that differs from the way that users are supposed to access a system, is known as a **backdoor**.

In this case, any users who were to read the code of the login program would be able to identify the fact that there's a backdoor into the system. However, users who download software usually won't have the opportunity to see the code of a program before it's compiled.

Exploits in a Compiler

Even if a user sees a program's code before they download it and determines that there doesn't seem to be any malicious code or backdoors in the code, that doesn't necessarily mean that the program itself is secure. Compilers, the program that translates source code into object code, can also be the source of exploit.

There are a couple ways that compilers can be used to exploit users. A compiler could, for instance, be programmed to take a perfectly benign login program and inject code into it that creates a backdoor. Anyone who looked at the source of the login program code itself wouldn't detect any signs of a backdoor. But if the source code were compiled with the malicious compiler, then the resulting program would have the backdoor in it. Of course, in this case, anyone who were to look at the source code of the compiler would see that there was code in the compiler that injects malicious code into the login program.

Let's take this one step further. Imagine that we wrote a compiler that injected malicious code into the compiler itself (remember that compilers themselves need to be compiled). Then a hacker could theoretically take benign source code for a compiler and turn it into a malicious compiler. In this case, even if the compiler source files and the login program source files didn't contain any malicious code or backdoors, compiling the source files could still result in the injection of malicious code.

CS50

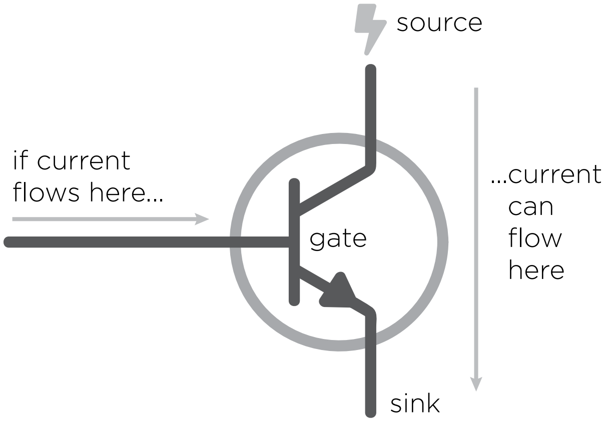
Transistors and Logic

Overview

Computers represent and process 1s and 0s using sequences of physical components called transistors. By linking different configurations of these transistors, computers can perform everything from basic arithmetic to playing a video. However, thanks to layers of abstraction, we don't need to constantly think about at the level of binary and transistors to program a computer. Sequences of transistors can be represented by Boolean logic, and these sequences of logic gates can then be packaged into chips (hardware) and code (software).

Key Terms

- transistor
- semiconductor
- true
- false
- Boolean logic



Transistors

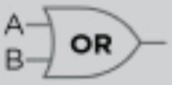
Transistors are small hardware devices made of semiconductors that act as switches for electric current. Because transistors are made of **semiconductors**, transistors can behave as both insulators (materials that inhibit electron flow) and conductors (materials that enable electron flow). When a small current is provided into a transistor's gate, the gate "opens" and current can flow from the source to the sink. When no current is available at the gate, the gate "closes" and current cannot flow from the source to the sink. By controlling current flowing into the gate, transistors can manipulate how current flows and therefore which signals are sent.

Boolean Logic

Since transistors either enable or disable the flow of electricity, we can use transistors to represent the binary values 1 and 0, or **true** and **false**. By linking these transistors in complicated webs, we can implement complex processes using a branch of math known as **Boolean logic**, created by the mathematician George Boole.

Boolean logic is built upon the two Boolean values, true and false, and the fundamental operators AND, OR, and NOT. Similar to arithmetic operations, these operations take value(s) as input and output one value. For example, in the statement 2 + 3 = 5, 2 and 3 are our inputs, + is our operator, and 5 is our output. Boolean logic looks very similar. In the statement "true AND false = false," true and false are our inputs, AND is our operator, and false is our output. The trick here is that while you know what the + sign does, you may not be familiar with the rules of Boolean operators. Luckily, they operate similarly to how these words are used in English. The AND operator requires that the first and second input are true to output true. Otherwise, it returns false. For example, the statement "the shirt is green AND striped" is only true if the shirt is both green and striped. The OR operator requires that either the first or second input is true for it to return true. The statement "the shirt is green OR striped" is true for a green shirt, a striped shirt, and a green striped shirt, but not true for any other shirt. The NOT operator, like a negative sign, only takes in one value and simply flips it, changing true to false and vice versa. The negation of the statement "the shirt is green AND striped" would be "the shirt is NOT green OR NOT striped."

By combining these fundamental operators in sequences, we can build gates like NAND (not and), NOR (not or), and XOR (exclusive or), and then eventually basic arithmetic, and then even more complex operations like editing a photo.

A	B	A B AND	 A B OR	A NOT
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

CS50

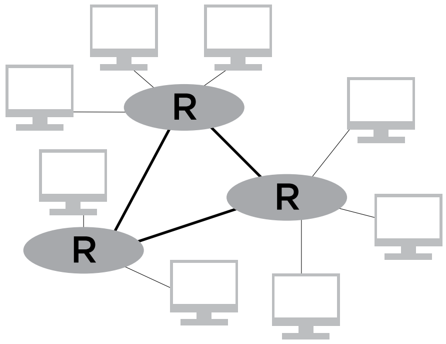
TCP and IP

Overview

In order for computers to communicate across the Internet, they need a standard set of rules—or **protocols**—to dictate how the communication should happen and how the data should get from one place on the Internet to another. Without these standard ways of communicating information, computers would not be able to guarantee that the receiver would get the information or that the receiving computer would know what to do with it. Two important protocols deal with this: the Transmission Control Protocol—also known as TCP—and the Internet Protocol—or IP. These are often collectively known as TCP/IP.

Key Terms

- protocol
- IP
- router
- IP Address
- packet
- TCP
- port



Internet Protocol

Recall that the Internet Protocol (**IP**) defines how information is transferred from one computer to another. It is structured as a web of connected **routers** (labeled as "R" in the diagram to the left), which are devices that help send information from one computer to another. Data will often need to pass through multiple routers to get from the sender's computer to its destination. Each router is programmed with a set of instructions (stored in a "routing table") that determine the direction in which the data must be sent so that it reaches its final destination.

IP Addresses

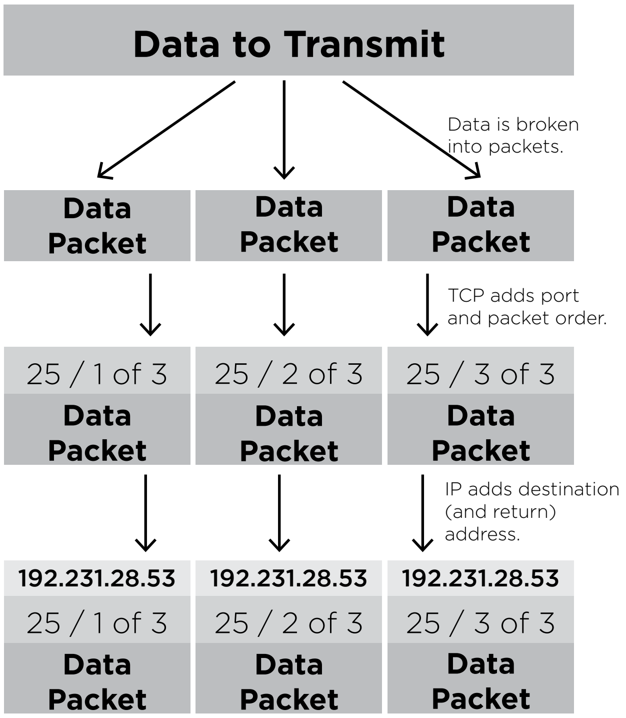
Just as homes need addresses so that mail can be delivered from one house to another, computers need addresses as well so that routers know where information is being sent from and where information should be sent to. These addresses are known as **IP Addresses**, and they take the form `#.#.#.#`, where each # stands for a number in the range 0 to 255. When a user types a web address (like google.com) into their web browser, a Domain Name System (DNS) server translates the web address to an IP address (like `172.217.0.46`).

Transmission Control Protocol

Instead of sending all of the data that one computer wants to send to another as one big packet, information on the Internet is sent in smaller data **packets**. The Transmission Control Protocol (**TCP**) is responsible for breaking up data into ordered packets. Since there is no guarantee that data packets will arrive at the destination at the same time, or even in the correct order, TCP labels each packet with the order it should go in. This way, the receiving computer can reassemble the packets in the right order.

TCP can also ask for the retransmission of lost data packets. Additionally, it assigns data a **port** number that indicates what type of internet service the data should be used for. For instance, SMTP (email) uses port 25, while HTTP (normal web browsing) uses port 80.

In summary, to get data across the Internet, TCP first breaks it down into smaller packets. Then TCP labels each packet with a port and packet number, IP tells the packet its destination, and the data is transmitted via routers which eventually direct the packet to its final destination.



Data is broken into packets.

TCP adds port and packet order.

IP adds destination (and return) address.

CS50 Unsolvable Problems

Overview

Today, it seems like computers have endless capabilities. But it turns out, there are some things computers will never be able to do. Problems that computers cannot definitively arrive at a solution for are called **unsolvable problems**.

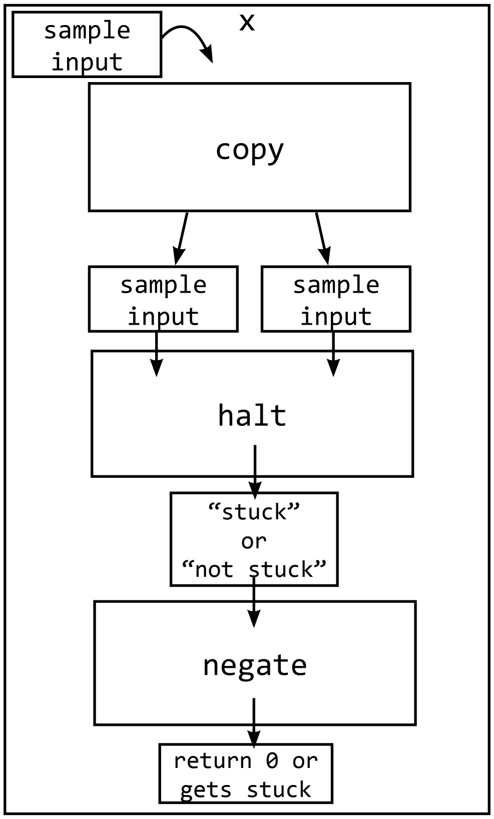
Key Terms

- unsolvable problems
- the halting problem
- heuristic

The Halting Problem

Generally speaking, computers operate by taking input and producing some output. But like any function, computers can only handle the inputs they were designed to handle. Take for example a calculator. It can handle inputs such as 3 + 8 and output 11, but it cannot sort an array of integers. Similarly, if you have a sorting program, it will not be able to handle arithmetic inputs.

The halting problem is the computability theory problem of determining if a program will finish running, or "halt", given a program and specific input. To examine this problem, let's imagine that we have a program, called halt, that takes in a program and some inputs to that program and then outputs whether or not the given program is going to get stuck. When we pass in calc – a calculator program – and "3 + 5" – inputs to calc – to halt, halt will print "not stuck," but if we pass in calc and "sort 1, 5, 2, 4, 9," halt will print "stuck." Although this theoretical program sounds simple enough, it is proven that this program cannot exist.



Consider a program, x, that has three main functions: copy, halt, and negate. Copy takes input and outputs two of whatever is inputted, halt takes a program and inputs to the given program and outputs whether the given program will get stuck or not. Negate takes halt's output as its input. If negate receives "stuck" it will return 0, and if negate receives "not stuck," it will get stuck.

Let's use a simple case to understand how x works. If the program calc is passed into x, copy will output 2 calcs, and halt will determine that calc would get stuck with an input of calc because all it can do is arithmetic. "Stuck" would then be passed into negate, and x would ultimately output 0. Now, let's pass in the program x as the input to x. Copy takes x and outputs two of them, then halt gets x with the input of x. From here there are two cases: halt can output "stuck" or "not stuck." If halt outputs "stuck," then negate would return 0. This implies that x given the input x does not get stuck, so halt is wrong. If halt outputs "not stuck," then negate would get stuck. This implies that x given the input x does get stuck, so halt is wrong again. Halt is wrong in both possible cases, therefore by contradiction, a program like halt cannot exist. It is impossible for a program like halt to be right 100% of the time using the same algorithm.

Heuristics

We know that there cannot exist an algorithm that can, in a finite amount of time, tell us if a program will halt or not. However, that doesn't mean there aren't ways to work around this logical impossibility. To develop good-enough solutions to impossible or very difficult problems, computer scientists often use heuristics. **Heuristics** are approaches to solving a problem that are not guaranteed to be completely correct, but that work well enough for the matter at hand. For example, using Google Maps to estimate how long it will take to travel home is a heuristic. While Google Map's prediction may not ultimately be correct, it adequately fulfills its purpose – it's good enough. For the halting problem, a possible heuristic could be testing a program for up to 1,000,000,000 seconds and then outputting whether or not the program gets stuck. Although it is plausible that the program could have finished its calculation on the 1,000,000,001th second, chances are the program would not have terminated in a practical amount of time, so for our case, we can deem it "stuck."

CS50 Virtual and Augmented Reality

Overview

It seems like virtual and augmented reality are talked about all the time as the future of technology these days. As the field grows and develops, we are led to wonder how VR and AR can change our world. What would happen if we can create thoroughly convincing versions of reality? Will VR and AR replace non-virtual interactions? For now, let's learn about how these technologies work, their technical challenges, and what we can do with them now.

Key Terms

- virtual reality
- augmented reality
- headset

Virtual and Augmented Reality

Virtual reality, or VR, refers to the field of technology that creates computer-generated environments and experiences that people can interact with as they would in real life. To do this, software and hardware technologies have to work together to appeal to all human senses in efforts to perfectly replicate how humans register their surroundings. In other words, technologies have to completely immerse the user in a virtual world; the user should be able to naturally interact with the world and the world should convincingly respond to the user's actions. By appealing to sight, sound, touch and less obvious senses like balance, virtual reality aims to minimize the user's awareness of the artificiality of the world. For example, in appealing to sight, virtual reality technologies have to take into account our peripheral vision, providing nearly 180° of graphics. Human physiology is central to the development of virtual reality technologies.

Augmented reality, or AR, is very similar to VR. However, instead of completely recreating a virtual realistic world, AR builds upon input from real life. AR often overlays some kind of visual information (graphics, text, etc) over a camera feed. Whereas VR can be used to create fantastical worlds, AR aims to enhance the experience of reality, connecting real life with resources and information of the virtual world.

Technologies

Creating immersive and interactive experiences requires new hardware devices and software technologies. Depending on the intended level of immersion, various types of hardware will be used. On a most basic level, as with all computers, the user needs some kind of output device to receive information about the virtual world and some kind of input device to interact with it. The most common VR device is some kind of **headset** that fits like a large pair of goggles. These headsets usually contain sensors to track a user's motions, lenses to reproduce how we see the world, and two display screens (one per eye). Along with a headset, VR may use input devices like joysticks and hand controllers to add haptic, or touch, interactions. VR and AR can also be used in more easily accessible devices such as smartphones and a headset with just a pair of lenses. On the software side, there are many JavaScript frameworks for programming VR and AR experiences. Additionally, Unity is a popular tool for creating 3D games and experiences.

Applications

VR and AR are being used in many fields today and will surely continue to expand their impact. VR and AR is perhaps most visible in the field of entertainment: many video games and films now take advantage of VR and AR, creating experiences of unparalleled engagement. VR and AR are also used often in education for professions in which realistic training can be risky or costly. This includes flight simulators for pilots and surgery simulators for doctors. Beyond professional training, VR and AR are used in treatments for psychological conditions like PTSD. VR and AR can also be used creatively. For example, they are used in architecture to create more realistic models and designs.

The cost of VR and AR technologies today remains quite high, however, as the cost is driven down, many predict that VR and AR will have a greater presence in our everyday lives. By hoping to unify technology more closely than ever with our experiences, VR and AR are revolutionary in their ability to humanize technology. As technology develops, the possibilities of VR and AR can only expand.

CS50 Version Control and Collaboration

Overview

Collaboration is an integral part of computer science; part of programming is constantly sharing and collaborating with peers. Sites like Facebook and Google are not written by one person. Rather, they require thousands of engineers, all working in teams to build specific pieces of the site. Furthermore, one team might rely on the code of many other teams. Effective collaboration is crucial.

Key Terms

- abstraction
- documentation
- comments
- Git
- commit

Abstraction

The saying "too many cooks spoil the broth" alludes to the idea that too many people working on the same thing is counter-productive. In cooking, like in programming, one can largely avoid this issue through **abstraction**, by building large projects out of a set of smaller, self-contained sub-projects and assigning these sub-projects to different people.

In a restaurant, for instance, one person might make the appetizer, another might make the main course or dessert, and still another might wait the table. In code, a project might rely on many individual programs, with programs' functions calling on still more library functions. The harmony that arises from the working together of many distinct parts is truly a beautiful thing!

Documentation and Comments

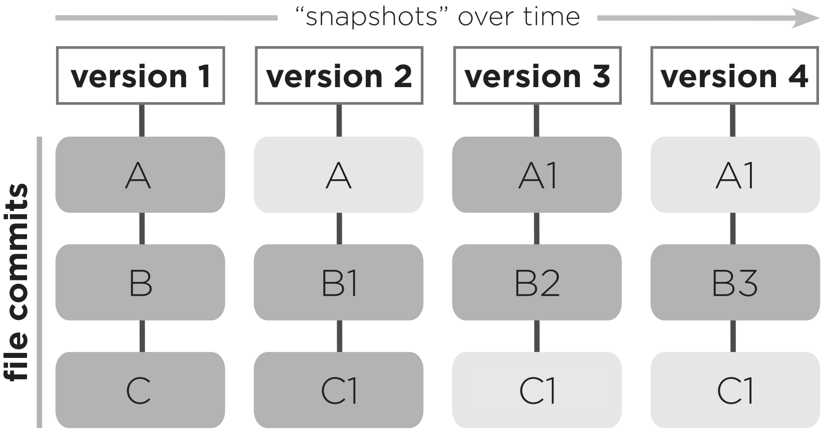
If abstraction requires breaking projects into smaller, independent pieces, documentation and comments allow engineers – and readers, more generally – to fit the pieces together. The man pages are one example of **documentation**: to use the library function `strlen()` from `string.h`, one shouldn't need to look at its actual code. Instead, the library's documentation acts as an executive summary, describing how and when to use a product, in this case `strlen()`.

There are cases in which looking at original code is useful or necessary. For these, **comments** in the code explaining what it does are incredibly important. These can make debugging one's own code or that of others much easier. Similarly, if we wanted to create a slightly different product based on someone else's code, clear and comprehensive comments would also be valuable. To promote consistency and clarity, best practices often instruct the use of a common style within a project or an organization.

Version Control and Git

There are many tools that enable coding collaboration, the most popular of which is a file tracking system called **Git**. The Git workflow is divided into three stages. First, we work on files in our working directory. Then, we pick what changes we want to store and add those to our staging area (also known as index). Finally, we **commit** those changes, which means that a "snapshot" of our project is stored in our repository (.git directory). Git also features branches – copies of a master project – that allow programmers to experiment with changes without actually affecting the original project.

Many version control systems save data as changes relative to the original files. Git works differently by saving "snapshots" of the entire project every time we commit. The diagram at right shows these "snapshots," or versions, over time. So version 2 represents our repository after our first commit. From this "snapshot," we can see we only made changes to files B and C. When files have not changed, Git links back to the previous file commit. That's denoted here with a lighter gray color, as is the case with file A in version 2.



Variables

Overview

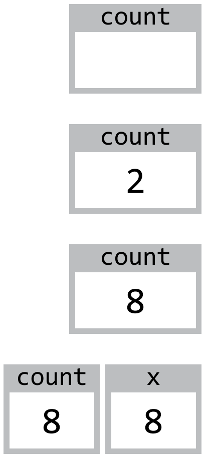
A **variable** is a storage container for data that is capable of holding different values that may change or update as programs execute. Your program can read the contents of a variable, update the contents of a variable, and display the value of a variable on the screen. Computer programs can use variables in order to remember useful information that the programs can then use later in the code.

Key Terms

- variables
- declaration
- type
- integer

Declaring and Setting Variables

```
1  int count;

2  count = 2;

3  count = 8;

4  int x = count;
```



The first step to using a variable in C is to let your program know that you want the variable to exist. This step is called the variable's **declaration** (also known as initialization). In C, this is done by first specifying the variable's **type**, which tells the program what kind of information will be stored inside of the variable, and then by specifying the variable's name (followed by a semicolon to end the programming statement).

For instance, in line 1 to the left, we've declared a new variable of type **int** to be named **count**. An **int** is a data type which stores an **integer**, which could be positive whole numbers, negative whole numbers, or zero (but not fractions or decimals). Currently, no value has been assigned to **count**: we've just told the program to create a space within which values can be stored later.

Once a variable has been declared, it can be manipulated in various ways. Line 2 takes the variable **count** and assigns its value to be **2**. Now, the number **2** is stored inside of the variable **count**. Optionally, we could have combined lines 1 and 2 into a single programming statement to declare a variable and set its value at the same time, via a line of code such as: **int count = 2;**.

After a variable has been given a value, its value can be updated. Line 3 updates the value of **count** again, this time to be **8**. Now, **count** forgets the number **2** and remembers the number **8** instead.

The value of a variable can be accessed just by using its name. For instance, line 4 declares a new variable (also of type **int**) this time named **x**, and initially sets its value to be **count**. This tells your program to go to the **count** variable, see what value is inside, and set the value of **x** to be that value. Since the current value of **count** is **8**, the value of **x** is set to also be **8**.

Variables from User Input

In many cases, a program may need to take input from the user and store the input as a variable. CS50 has written several functions (declared in a file called **cs50.h**) that serve this very purpose.

For instance, **get_int("prompt_string")** prompts the user to input an integer. In the program to the right, line 6 uses **get_int()** to take in an integer as input from the user with the prompt "**Integer please:**", and saves that integer in a variable called **i**.

```
1  #include <cs50.h>
2  #include <stdio.h>
3
4  int main(void)
5  {
6      int i = get_int("Integer please: ");
7      printf("i is %i \n", i);
8  }
```

Line 7 then displays the value of the variable on the screen. The **%i** in the string is a special syntax which acts as a placeholder for an integer. We tell **printf** what integer to use in that placeholder by passing it an additional argument, where an argument is just a value inside of the parentheses of a function. Inside of the parentheses next to **printf** we've included two arguments: the string **"i is %i"**, and the integer **i**, which will take the place of **%i**. If the user were to enter the number **28** as input on line 6, then line 7 would replace **%i** with the value of **i** (which is **28**) and display the string **"i is 28"** on the screen followed by a new line denoted by the **\n**.