Introduction to Computer Science

PTI CS 103 Lecture Notes

The Prison Teaching Initiative

This version: May 21, 2019

Acknowledgements

TK.

Contents

1

Introduction

1.1 Overview of course

Knowing just a little bit of computer science can get you started right away in actual applications. One of the goals of this course is to learn about the fascinating subject of computer science. Another is to develop algorithmic thinking skills that will help with day-to-day critical problem-solving skills. But perhaps the most important goal of the course is to develop coding skills, which will not only open up new job opportunities but also make you more effective in most areas of business.

In the first semester, we will spend the first two classes of each week on computer science theory and special topics. The final day of each week will be a lab day, where we actually start practicing coding skills.

In the second semester, we will start focusing more on practical coding, with a single day a week for theory and 2 lab periods per week for coding.

Broadly, we will cover the following topics:

- How modern computers work
  - Hardware
  - Software
  - Computer networks and information systems
- Algorithms for quickly solving complex problems
  - Searching
  - Sorting
- Data structures
  - Arrays
  - ArrayLists
- Applications of Computer science
  - Basic coding in Java
  - How to use productivity software

1.2 Brief history of computer science

Timeline (credit: https://www.worldsciencefestival.com/infographics/a_history_of_computer_science/):

- Invention of the abacus (2700-2300 BC, Sumerians)
- Design of first modern-style computer (Charles Babbage, 1837)

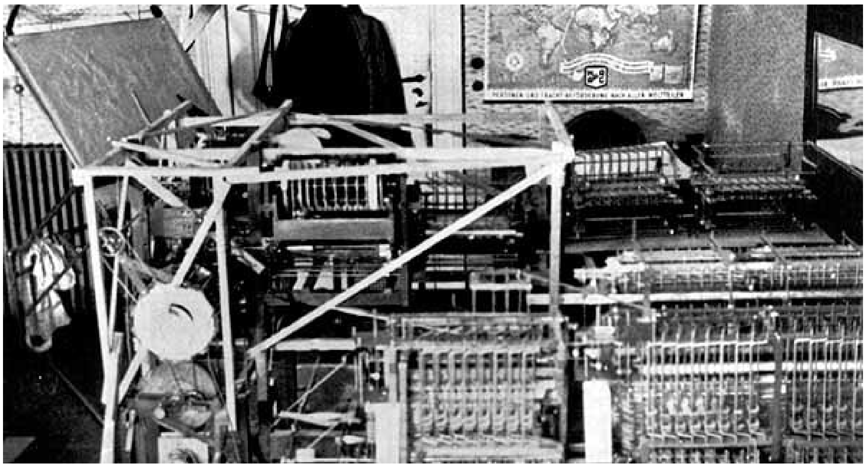


Figure 1.1: Construction of Konrad Zuse's Z1, the first modern computer, in his parents' apartment. Credit: https://history-computer.com/ModernComputer/Relays/Zuse.html

- Design of first computer algorithm (Ada Lovelace, 1843)
- Invention of first electronic digital computer (Konrad Zuse, 1941)
- Invention of the transistor (Bell labs, 1947)
- Invention of the first computer network (early Internet) (DARPA, 1968)
- Invention of the World Wide Web (Sir Tim Berners-Lee, 1990)

1.3 Components of a computer

A computer is an electronic device used to process data. Its basic role is to convert data into information that is useful to people.

There are 4 primary components of a computer:

- Hardware
- Software
- Data
- User

1.3.1 Hardware

Computer hardware consists of physical, electronic devices. These are the parts you actually can see and touch. Some examples include

- Central processing unit (CPU)
- Monitor
- CD drive
- Keyboard
- Computer data storage
- Graphic card
- Sound card
- Speakers
- Motherboard

We will discuss these components in more detail in lecture 3.

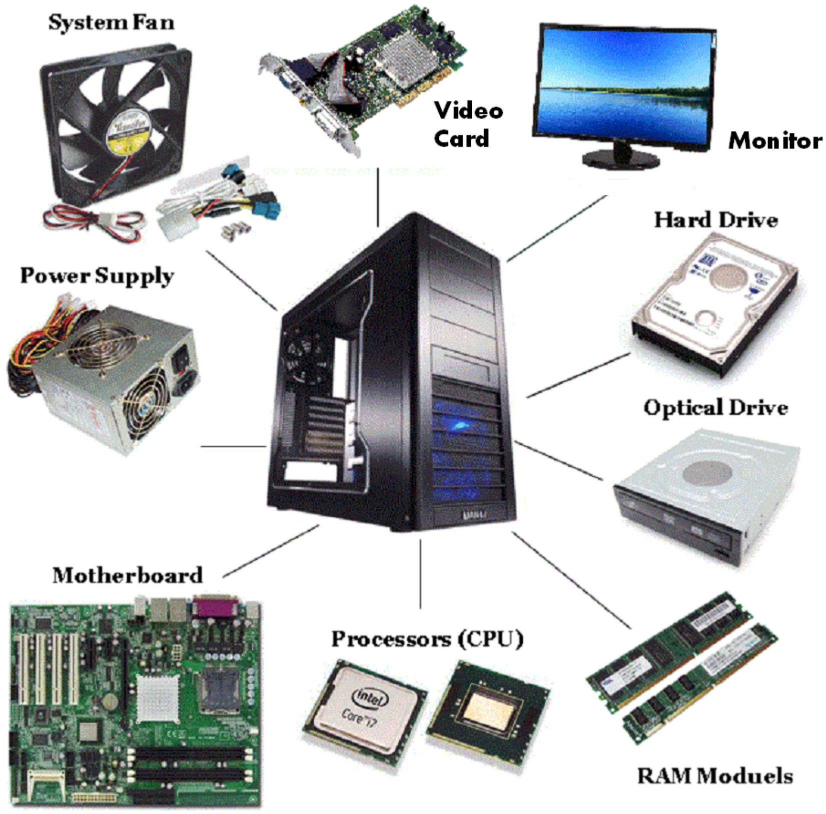


Figure 1.2: Examples of hardware components of a personal computer. Credit: https://www5.cob.ilstu.edu/dsmath1/tag/computer-hardware/

1.3.2 Software

Software (otherwise known as ”programs” or ”applications”) are organized sets of instructions for controlling the computer.

There are two main classes of software:

- Applications software: programs allowing the human to interact directly with the computer
- Systems software: programs the computer uses to control itself

Some more familiar applications software include

- Microsoft Word: allows the user to edit text files
- Internet Explorer: connects the user to the world wide web
- iTunes: organizes and plays music files

While applications software allows the user to interact with the computer, systems software keeps the computer running. The operating system (OS) is the most common example of systems software, and it schedules tasks and manages storage of data.

We will dive deeper into the details of both applications and systems software in lecture 4.

1.3.3 Data

Data is fundamentally information of any kind. One key benefit of computers is their ability to reliably store massive quantities of data for a long time. Another is the speed with which they can do calculations on data once they receive instructions from a human user.

While humans can understand data with a wide variety of perceptions (taste, smell, hearing, touch, sight), computers read and write everything internally as ”bits”, or 0s and 1s.

Computers have software and hardware which allow them to convert their internal 0s and 1s into text, numerals, and images displayed on the monitor; and sounds which can be played through the speaker.

Similarly, humans have hardware and software used for converting human signals into computer-readable signals: a microphone converts sound, a camera converts pictures, and a text editor converts character symbols.

1.3.4 Users

Of course, there would be no data and no meaningful calculations without the human user. Computers are ultimately tools for making humans more powerful.

As we will see in the next section, however, different types of computers have different roles for the user.

1.4 Types of computers

1.4.1 Supercomputers

These are the most powerful computers out there. The are used for problems that take along time to calculate. They are rare because of their size and expense, and therefore primarily used by big organizations like universities, corporations, and the government.

The user of a supercomputer typically gives the computer a list of instructions, and allows the supercomputer to run on its own over the course of hours or days to complete its task.

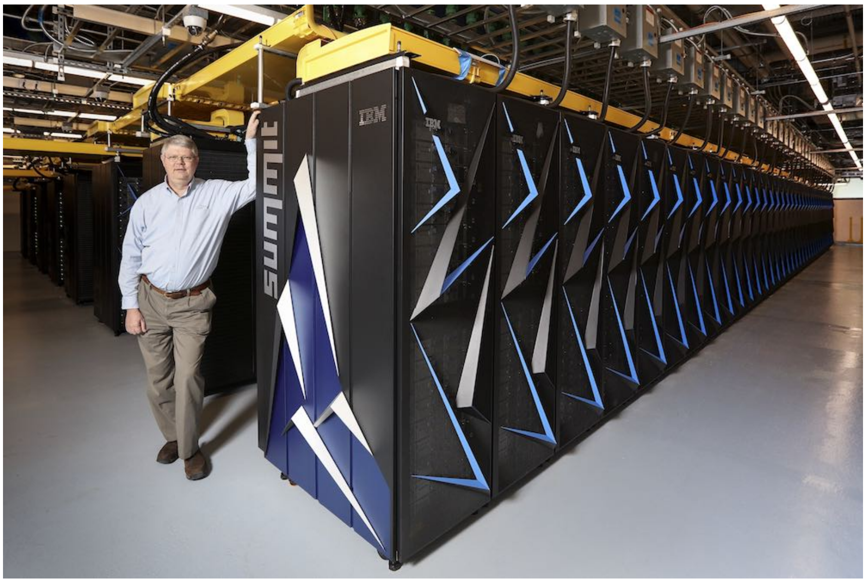


Figure 1.3: Summit, a world-class supercomputing cluster at Oak Ridge National Laboratory in Tennessee. Credit: https://insidehpc.com/2018/11/new-top500-list-lead-doe-supercomputers/

1.4.2 Mainframe computers

Although not as powerful as supercomputers, mainframe computers can handle more data and run much faster than a typical personal computer. Often, they are given instructions only periodically by computer programmers, and then run on their own for months at a time to store and process incoming data. For example, census number-crunching, consumer statistics, and transactions processing all use mainframe computers

1.4.3 Personal computers

These are the familiar computers we use to interact with applications every day. Full-size desktop computers and laptop computers are examples

1.4.4 Embedded computers

In the modern ”digital” age, nearly all devices we use have computers embedded within them. From cars to washing machines to watches to heating systems, most everyday appliances have a computer within them that allows them to function.

1.4.5 Mobile computers

In the past 2 decades, mobile devices have exploded onto the scene, and smartphones have essentially become as capable as standalone personal computers for many tasks.

1.5 Why computers are useful

Computers help us in most tasks in the modern age. We can use them, for example, to

- write a letter
- do our taxes
- play video games
- watch videos
- surf the internet

- keep in touch with friends
- date
- order food
- control robots and self-driving cars

Example 1.5.1: What are some other tasks a computer can accomplish?

This is why the job market for computer scientists continues to expand, and why computer skills are more and more necessary even in non-computational jobs.

According to a Stackoverflow survey from 2018 (https://insights.stackoverflow.com/survey/2018/), 9% of professional coders on the online developer community have only been coding for 0-2 years. This demonstrates two things:

1. The job market for people with coding skills is continually expanding
2. It doesn't take much to become a coder

Some examples of careers in computer science include

- IT management / consulting
- Game developer
- Web developer
- UI/UX designer
- Data analyst
- Database manager

References

Computer Science: An Interdisciplinary Approach, Robert Sedgewick and Kevin Wayne.

University of Wisconsin-Madison CS 202 Lectures, Andrea Arpaci-Dusseau. (http://pages.cs.wisc.edu/ dussea F11/)

2

Hardware

Today's lecture will focus on computer systems, a complex group of devices working together to perform a common task (or tasks). A user will interact with a computer through a variety of input and output devices (e.g. keyboards, mice, speakers, microphone, and monitors). A user's input will be processed, some computations will be performed, and then the resulting output will be displayed to the user. When most of us thinking about computers, we often think of a desktop or laptop computer, that come equipped with a keyboard, mouse, and monitor as seen in Figure 2.1; however, many things we interact with daily are computerized, including cell phones, cars, traffic lights, smart watches, televisions, and manufacturing lines. Today each of these items have sensors to perceive the real world, use an embedded computing device to understand the sensory input, and use a combination of display and mechanical devices to interact with the real world.

Example 2.0.1: For intersections across busy roadways, some traffic lights are computerized to optimize road traffic. These lights will stay green along the busier of the two roads, and use cameras or pressure sensors to detect the presence of cars along the less busy of the two roads, thus switching to allowing the cars on the less busy road to cross when it arrives. Overall, providing a less congested intersection by relying on an embedded computer.

Today we will introduce three fundamental parts of computer systems: input and output devices, memory, and the central processing unit (CPU). These components work together to perform the basic building blocks of input processing, storage, control, and output. Understanding how the three parts work together will allow us to create powerful information processing tools. We will introduce each of these parts in turn. In figure 2.2, we see how these parts come together to form a computer system (similar to the ones you'll use to program in this course).

Figure 2.1: A variety of computer systems: desktops, laptops, tablet, and smart phone.

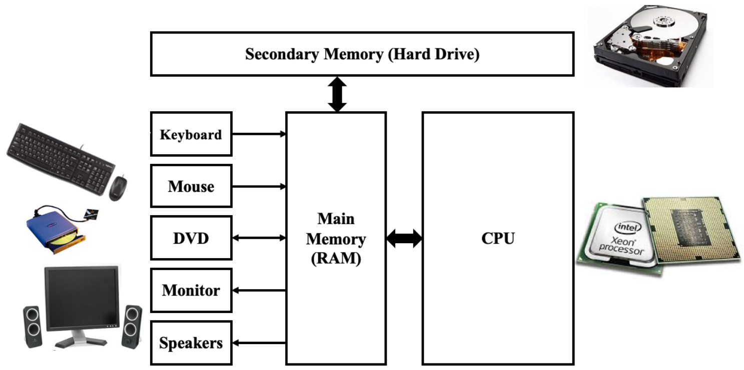


Figure 2.2: Interconnected parts of a computer system (keyboard, mouse, monitor, DVD player / burner, speaker, hard drive, CPU).

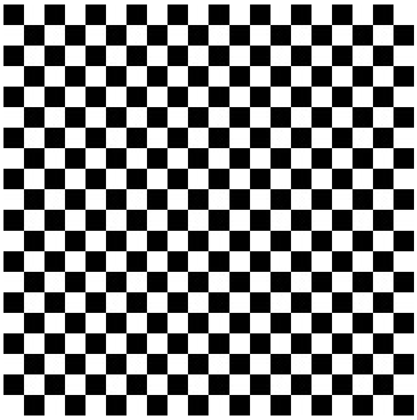
2.1 Input and Output Devices

We will begin by discussing input and output devices. These devices allow the computer to interact with users and the world directly. Without these devices, a computer system would be very boring, always performing the same computation each time it's used. Even if it did compute a different value we would be unable to examine the value. A computer needs to be able to accept input and allow output. The first computers would occupy a large room in an office building and connect to a terminal (a keyboard and a text screen) in another room for users to interact with. Thanks to Moore's Law, computers many orders of magnituted more powerful can fit in the palm of your hand. Likewise, the variety of input and output devices has multipled. We still have the keyboard and monitor, we've added the mouse for interacting with graphical displays. Today's phones are more computer than phone, coming equipped with: speakers, microphones, touch screens, cameras, fingerprint scanners, radio transmitters, and much more. Computers even come embedded in other devices like cars, traffic lights, X-ray machines, and thermostats to both control and monitor the devices. As shown in Figure 2.2, these devices connect to the rest of the computer through the computer's memory. This kind of input and output is called memory mapped I/O (input and output). Creaters of input (or output) devices are assigned a section of the computer's memory to write (read resp.) data. The computer will then read (write resp.) data to those locations to communicate with the given device. The creaters of these devices, agree upon a known format to read and write data.

Example 2.1.1: You can think of this communication between devices and computer similar to leaving messages for a friend in a locker. Only you and your friend have access to this locker, which only holds space for one message. The format you agree upon is which language you'll use to speak (e.g. English) and any special keywords or phrases. You might agree with your friend, that if either of you write a message saying “The morning is upon us” that the other will wait until “The night has come” before leaving any new messages.

The format that devices and computers communicate in are generally very simple and structured to permit fast and easily understandable communication for computers and devices.

Example 2.1.2: A monitor is a graphical display for computers. Let's consider a monitor connected to a computer that only displays in black and white images that are 20 x 20 pixels large. The monitor and keyboard agree upon using the following



```
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
```

Figure 2.3: An example checkered image and its encoding — newlines and spaces added for readability.

format to communicate. The format is black and white images that are 20 x 20 pixels large. Each pixel's value is represented at 0 for black and 1 for white. Then an image is represented as a 400 = (20 x 20) long sequence of pixel values. The sequence is ordered left to right, top to bottom. Now that both the monitor and computer agree upon the communication format, the computer can write images to the section of memory dedicated to the monitor and the monitor will read the image and display the image on it's screen. Figure 2.3 displays an example image, a 20 x 20 chekerboard with its encoding.

Note: while this is a simplified example, this is similar to how modern graphical displays communicate with computers.

2.2 Memory

Another fundamental part of a computer, is the memory. By memory, we mean the ability to store and recall data. This is very similar to physical storage of items. Figure 2.4 shows three storage locations — a storage closet, a garage, and a warehouse. Each of the three locations have tradeoff between convenience of location and storage capacity. The closet can contain a few things and is the same room you need it. The garage can fit even more things and is only a walk outside (or through) your home, and the warehouse can fit practically anything you would want to store but you have to drive to the warehouse to pick-up or store your items. Similarly, a computer's memory makes the same trade-offs.

There are two major types of memory, Main Memory (RAM), and Secondary Memory (e.g. hard disks, solid-state drves, tape drives, etc.) Main memory is volatile, meaning that the contents of the memory is not preserved when a computer is turned off and back on. On the other hand Secondary Memory, is meant to be persistent (the opposite of volatile). Main Memory can be thought of as the "scratch paper" the computer uses for computations. Computers will also use Main Memory as a conduit for communicating between the CPU and all other parts of the computer. Staying with the analogy from Figure 2.4 main memory is closer to a garage (where you can lose items when you turn off the lights) — there is enough room to fit most items you use regularly and is close enough to not worry about the time it takes to get to the garage.

In most modern computers, programs are treated as data. That is the individual instructions that combine to form a program are stored in memory just as data is. It is the job of the computer to properly understand if a segment of memory is data or a program. The computer is able to fetch data from Secondary Memory to Main Memory or persist data in Main Memory to Secondary Memory when needed; however, this process of transferring data between Secondary and Main Memory can cost a lot of time relative to keeping data in Main Memory only.

Figure 2.4: Storage closet, garage, and warehouse trading off between capacity and locality.

2.3 Central Processing Unit

The final part of a computer we will introduce today is the central processing unit (CPU) — processor, main processor, etc. The CPU is the physical circuitry of a computer that performs instructions. The CPU has several key components: the control logic, arithmetic and logic unit (ALU), registers, program counter (PC), and clock. These components work together to fetch, decode, and execute all instructions — the building blocks of all programs. Instructions vary between different brands of CPUs, but, in general, they will include arithmetic, control, read (from memory), and write (to memory) functionalities. Example 2.3.1 shows several instructions that together would perform `x = x + y`, given `x` is stored in memory location 16 and `y` at memory location 20. These instructions are quite low level, and harder for humans to read than the programs we will write in this course. However, the programs we write will be translated into these instructions to be easily understood and executed by the CPU.

Example 2.3.1:

```
load R1 16    -- Load value at memory location 16 into register 1
load R2 20    -- load value at memory location 20 into register 2
add  R1 R2 R1 -- add the value in register 1 to the value in register 2
                 and store in register 1
store R1 16   -- store the value in register 1 to memory location 16
```

A key component of a CPU is its clock. The clock allows the CPU to progress in time, triggering the time to progress from time `t` to time `t + 1`. A single time segment is referred to a clock cycle. You might have heard about a computer's CPU speed (e.g. my computer runs at 2.4 GHz = 2.4 billion clock cycles a second). This is determined by how fast the clock transitions from one time to the next. This clock drives the progress of the CPU. The time an instruction takes is measured in instruction cycles. The rate of the CPU's clock is determined by the slowest operation of the CPU (e.g. fetch, decode, execute stage).

In addition to clocks, the CPU contains a group of memory locations called registers. A single register is capable of holding a single word of information (the smallest unit of data in a computer). The key benefit of registers is the ability for the CPU to immediately read and write the contents of the CPU. The value of registers can be updated on each clock trigger (i.e. on the change from time `t` to `t + 1`). Most modern CPU's will have between 16 and 64 registers that programs may use. For comparison, accessing Main memory can take 10's or even 100's of instruction cycles to access while registers are immediately available to the CPU.

The control unit and program counter (PC) will fetch, decode, and output the controls for the execution of each instruction. The program counter holds the location of the next instruction to be executed. The next instruction is then fetched to the CPU. The CPU's control unit then decodes the fetched instruction and outputs control signals (commands) to main memory and the ALU. The CPU may read data from memory (e.g. store) and then the ALU will then execute the action specified by the control unit (e.g. add, subtract, multiply, compare to 0, etc.), and then possibly write the output to memory.

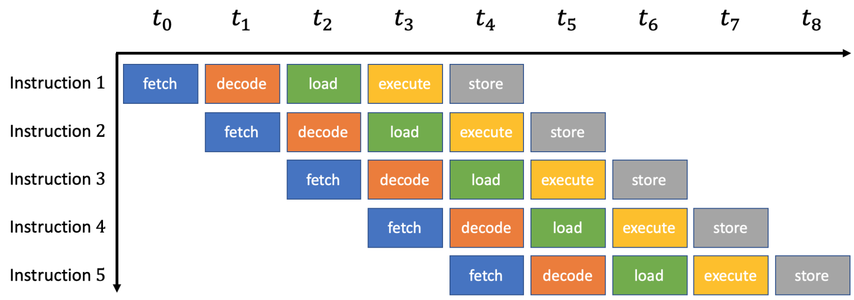


Figure 2.5: Instruction Pipelining

In most computers, the CPU and its constiuent parts are responsible for all computing needs of the computer. In some select systems, there will be additional hardware to perform specialized operations (e.g. graphics processing units for processing / producing images). It is the CPU's responsbility to control the computer and coordinate with devices to execute programs. As such, the CPU has seen a quick evolution to increase it's processing power. Electrical engineers, originally focused on making the CPU smaller and smaller and thus quicker, following Moore's Law : every two years the size of a CPU shrinks in half. Additionally, CPU's were designed with a pipelining architecture (i.e. multiple instructions are executed in quick succession). This is done by noting that each stage of the five stages — fetch, load, decode, execute, and store — can be performed independently. Thus, while one instruction is being executed, the next instruction can be decoded. Figure 2.5 shows how pipelining is peformed by executing the different parts of the pipeline in parallel for five consecutive instructions.

Due toe the decline of Moore's Law in recent years, many CPU designers focus on increasing processing power by improving parallelism (i.e. being able to execute multiple instructions at a time). This allows instructions that do not depend on each other to be executed at a time. These CPU are refered to as multi-core, as they have multiple cores that each have thier own set of registers, control unit, ALU, and PC but share main memory and a single clock. In this class we, will not teach how to effectively harness parallel programming, but note that this is an important progress in how hardware has evolved.

2.4 Conclusion

In this chapter, we covered the three fundamental parts of a computer system: input and output devices, Main and Secondary Memory, and the CPU. We discussed their roles, relationships, and basic capabilities. I hope that this will help you better understand how hardware works at a high level to better improve how to write programs that will eventually run on these computer system. In the next chapter we will begin discussing the concept of Software and how its similarities and differences to hardware and where the boundary between the two lies.

2.5 Learning Objectives

After covering this chapter, you should be able to answer the following questions:

1. What three parts comprise a computer system?
2. What are examples of common input and output devices?
3. Name an uncommon example of a computer system and explain how it may work.
4. How does memory mapped input and output work?

5. Name four kinds of memory devices and explain the difference between Main and Secondary Memory.

6. What parts comprise the central processing unit (CPU)?

7. Describe three possible methods to increase the computation power of a CPU.

3

Arrays

Consider this snippet of code:

```
if      (day ==  0) System.out.println("Monday");
else if (day ==  1) System.out.println("Tuesday");
else if (day ==  2) System.out.println("Wednesday");
else if (day ==  3) System.out.println("Thursday");
else if (day ==  4) System.out.println("Friday");
else if (day ==  5) System.out.println("Saturday");
else if (day ==  6) System.out.println("Sunday");
```

What does this code do? It prints the day of the week after conditioning on the value of an integer `day`. But this code is repetitive. It would be useful if we had some way of creating a list of days of the week, and then just specifying which of those days we wanted to print. Something like this:

```
System.out.println(DAYS\_OF\_WEEK[day]);
```

To achieve this in Java, we need arrays.

> **Definition 3.0.1.** An *array* is an ordered and fixed-length list of values that are of the same type. We can access data in an array by *indexing*, which means referring to specific values in the array by number. If an array has `n` values, then we think of it as being numbered from `0` to `n-1`.

To *loop* or *iterate* over an array means that our program accesses every value in the array, typically in order. For example, if we looped over the array in the diagram, that would mean that we looked at the value at the 0th index, then the value at the 1st index, then the value at the 2nd index, and so on.

When we say that the array is "ordered" is that the relationship between an index and its stored value is unchanged (unless we explicitly modify it). If we loop over an unchanged array multiple times, we will always access the same values.

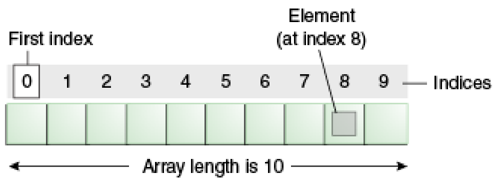


Figure 3.1: Diagram of an array (Credit: https://www.geeksforgeeks.org/arrays-in-java/)

Arrays are *fixed-length*, meaning that after we have created an array, we cannot change its length. We will see in the next chapter [TK: confirm] that `ArrayLists` are an array-like data structure that allows for changing lengths.

Finally, all the values in an array must be of the same type. For example, an array can hold all floating point numbers or all characters or all strings. But an array cannot hold values of different types.

3.1 Creating arrays

The syntax for creating an array in Java has three parts:

1. Array type
2. Array name
3. Either: array size or specific values

For example, this code creates an array of size `n = 10` and fills it with all `0.0`s

```
double[] arr;                         // Declare array
arr = new double[n];                  // Initialize the array
for (int i = 0; i < n; i++) {         // Iterate over array
    arr[i] = 0.0;                     // Initialize elements to 0.0
}
```

The key steps are: we first declare and initialize the array. We then loop over the array to initialize specific values. We can also initialize the array at compile time, for example

```
String[] DAYS_OF_WEEK = {
//   Indices:
//   0      1      2      3      4      5      6
    "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"
};
```

Notice the difference in syntax. When creating an empty array, we must specify a size. When initialize an array at compile time with specific values, the size is implicit in the number of values provided.

Finally, in Java, it is acceptable to move the brackets to directly after the type declaration to directly after the name declaration. For example, these two declarations are equivalent:

```
int arr[];
int[] arr;
```

3.2 Indexing

Consider the array `DAYS_OF_WEEK` from the previous section. We can *index* the array using the following syntax:

```
System.out.println(DAYS_OF_WEEK[3]);  // Prints "Thu"
```

In Java, array's are said to use *zero-based indexing* because the first element in the array is accessed with the number `0` rather than `1`.

Example 3.2.1: What does `System.out.println(DAYS_OF_WEEK[1]);` print?

Example 3.2.2: What does this code do? What number does it print?

```
double sum = 0.0;
double[] arr = { 1, 2, 2, 3, 4, 7, 9 }
for (int i = 0; i < arr.length; i++) {
    sum += arr[i];
}
System.out.println(sum / arr.length);
```

3.3 Array length

As mentioned previously, arrays are *fixed-length*. After you have created an array, it's length is unchangeable. You can access the length of an array `arr[]` with the code `arr.length`.

Example 3.3.1: What does `System.out.println(DAYS_OF_WEEK.length);` print?

Example 3.3.2: Write a `for` loop to print the days of the week in order (Monday through Sunday) using an array rather than seven `System.out.println` function calls.

3.4 Default initialization

In Java, the default initial values for numeric primitive types is `0` and `false` for the `boolean` type.

Example 3.4.1: Consider this code from earlier:

```
double[] arr;
arr = new double[n];
for (int i = 0; i < n; i++) {
    arr[i] = 0.0;
}
```

Rewrite this code to be a single line.

3.5 Bounds checking

Consider this snippet of code.

Example 3.5.1: Where is the bug?

```
int[] arr = new int[100];
for (int i = 0; i <= 100; ++i) {
    System.out.println(arr[i]);
```

```
}
```

The issue is that the program attempts to access the value `arr[100]`, while the last element in the array is `arr[99]`.

This kind of bug is called an “off-by-one error” and is so common it has a name. In general, an off-by-one-error is one in which a loop iterates one time too many or too few.

Example 3.5.2: Where is the off-by-one-error?

```
int[] arr = new int[100];
for (int i = 0; i < array.length; i++) {
    arr[i] = i;
}
for (int i = 100; i > 0; --i) {
    System.out.println(arr[i]);
}
```

Example 3.5.3: Fill in the missing code in this `for` loop to print the numbers in reverse order, i.e. `5, 4, 3, 2, 1`:

```
int[] arr = { 1, 2, 3, 4, 5 };
for (???) {
    System.out.println(arr[i]);
}
```

3.6 Empty arrays

This code prints five values, one per line, but we never specified which values. What do you think it prints?

```
int[] arr = new int[5];
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]);
}
```

In Java, an unitialized or empty array is given a default value:

- For `int`, `short`, `byte`, or `long`, the default value is `0`.
- For `float` or `double`, the default value is `0.0`
- For `boolean` values, the default value is `false`.
- For `char`, the default value is the null character `’0̆000’`.

Note that an array can be partially initialized.

Example 3.6.1: What does this code print?

```
char[] alphabet = new char[26];
alphabet[0] = 'a';
alphabet[1] = 'b';
```

```
for (int i = 0; i < alphabet.length; i++) {
    System.out.println(alphabet[i]);
}
```

3.7 Enhanced for loop

So far, we have seen how to iterate over arrays by indexing each element with a number:

```
char[] vowels = {'a', 'e', 'i', 'o', 'u'};
for (int i = 0; i < vowels.length; ++ i) {
    System.out.println(vowels[i]);
}
```

We can perform the same iteration without using indices using an "enhanced `for` loop" or `for-each` loops:

```
char[] vowels = {'a', 'e', 'i', 'o', 'u'};
for (char item: vowels) {
    System.out.println(item);
}
```

3.8 Exchanging and shuffling

Two common tasks when manipulating arrays are *exchanging two values* and *shuffling* values. (*Sorting* is more complicated and will be address later.)

To exchange to values, consider the following code:

```
double[] arr = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 };
int i = 1;
int j = 4;
double tmp = arr[i];
arr[i] = arr[j];
arr[j] = tmp;
```

Example 3.8.1: What are the six values in the array, in order?

To shuffle the array, consider the following code:

```
int n = arr.length;
for (int i = 0; i < n; i++) {
    int r = i + (int) (Math.random() * (n-i));
    String tmp = arr[r];
    arr[r] = arr[i];
    arr[i] = tmp;
}
```

Example 3.8.2: What does this code do:

```
for (int i = 0; i < n/2; i++) {
    double tmp = arr[i];
    arr[i] = arr[n-1-i];
```

```
    arr[n-i-1] = tmp;
}
```

Exercises

3.1 Write a program that reverses the order of values in an array.

3.2 What is wrong with this code snippet?

```
int[] arr;
for (int i = 0; i < 10; i++) {
    arr[i] = i;
}
```

3.3 Rewrite this snippet using an enhanced `for-each` loop (for now, it is okay to re-define the array):

```
char[] vowels = {'a', 'e', 'i', 'o', 'u'};
for (int i = array.length; i >= 0; i--) {
    char letter = vowels[i];
    System.out.println(letter);
}
```

3.4 Write a program that uses `for` loops to print the following pattern:

```
1********

12*******

123******

1234*****

12345****

123456***

1234567**

12345678*

123456789
```

3.5 Write a program `HowMany.java` that takes an arbitrary number of command line arguments and prints how many there are.

References

Computer Science: An Interdisciplinary Approach, Robert Sedgewick and Kevin Wayne.

4

ArrayLists

A *collection* is a group of objects. Today, we'll be looking at a very useful collection, the `ArrayList`. A *list* is an ordered collection, and an `ArrayList` is one type of list. We will see at the end of this chapter how an `ArrayList` is different from an `array`.

4.1 Creating an `ArrayList`

Let's create a class `NameTracker` and follow along in it. Before we can use an ArrayList, we have to import it:

```
import java.util.ArrayList;

public class NameTracker {

    public static void main(String args[]) {
    }
}
```

Next, we call the constructor; but we have to declare the type of object the `ArrayList` is going to hold. This is how you create a new `ArrayList` holding `String` objects.

```
ArrayList<String> names = new ArrayList<String>();
```

Notice the word "String" in angle brackets. This is the Java syntax for constructing an ArrayList of String objects.

4.2 `add()`

We can add a new String to names using the `add()` method.

```
names.add("Ana");
```

Example 4.2.1: Exercise: Write a program that asks the user for some names and then stores them in an ArrayList. Here is an example program:

```
Please give me some names:
Sam
Alecia
Trey
```

```
Enrique
Dave

Your name(s) are saved!
```

We can see how many objects are in our ArrayList using the size() method.

```
System.out.println(names.size()); // 5
```

Example 4.2.2: Modify your program to notify the user how many words they have added.

```
Please give me some names:
Mary
Judah

Your 2 name(s) are saved!
```

4.3 `get()`

Remember how the `String.charAt()` method returns the `char` at a particular index? We can do the same with names. Just call `get()`:

```
names.add("Noah");
names.add("Jeremiah");
names.add("Ezekiel");
System.out.println(names.get(2)); // ``Ezekiel''
```

Example 4.3.1: Update your program to repeat the names back to the user in reverse order. Your solution should use a for loop and the size() method. For example:

```
Please give me some names:
Ying
Jordan

Your 2 name(s) are saved! They are:
Jordan
Ying
```

4.4 `contains()`

Finally, we can ask our names `ArrayList` whether or not it has a particular string.

```
names.add("Veer");
System.out.println(names.contains("Veer")); // true
```

Example 4.4.1: Update your program to check if a name was input by the user. For example:

```
Please give me some names:
Ying
Jordan

Search for a name:
Ying

Yes!
```

4.5 ArrayLists with custom classes

An `ArrayList` can hold any type of object! For example, here is a constructor for an `ArrayList` holding an instance of a `Person` class:

```
ArrayList<Person> people = new ArrayList<Person>();
```

where `Person` is defined as

```
public class Person {

    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }
}
```

Example 4.5.1: Modify our program to save the user's input names as Person instances. Rather than storing String objects in the ArrayList, store Person objects by constructing them with the input name. You'll need to use the Person constructor to get a Person instance!

4.6 Arrays versus ArrayLists

The most important difference between an array and an `ArrayLists` is that an `ArrayLists` is dynamically resized. Notice that when we created an `ArrayLists`, we did not specify a size. But when we create an array, we must:

```
int[] numbers1 = new int[10];
ArrayList<Integer> numbers2 = new ArrayList<Integer>();
```

In the first line of code, we create an array of length 10, and we cannot add more than 10 elements to `numbers1`. In the second line of code, we create an `ArrayList`, but we do not need to specify a size.

The second distinction between arrays and `ArrayLists` is syntax. In Java, nearly everything is a class. An `ArrayList` is a class with useful methods such as `get()` and `contains()` and is therefore more "Java-like". Arrays are not quite primitives such as `ints` and note quite classes. Most likely, their syntax was borrowed from the C or C++ programming languages.

Exercises

4.1 Write a class BlueBook that tells the user the price of their car, depending on the make, model, and year. You should use Car.java and the stencil file provided, BlueBook.java. Your program depends on what cars your BlueBook supports, but here is an example program:

```
What is your car's make?
Toyota
What is your Toyota's model?
Corolla
What is your Toyota Corolla's year?
1999

Your 1999 Toyota Corolla is worth \$2000.
```

4.2 Notify the user if the car is not in your BlueBook.

4.3 Clean up main by putting your code for creating the ArrayList in a separate method. What type should the method return?

4.4 If the car is not in the BlueBook, ask the user to input the relevant data, construct a new Car instance, add it to your ArrayList.

References

1. https://github.com/accesscode-2-1/unit-0/blob/master/lessons/week-3/2015-03-24_arraylists.md

5

Systems Development

5.1 Introduction

Creating good software requires good coding skills, so you can get the computer to do what you want. Additionally, however, it requires good "systems development" skills so that (1) your code is organized / quickly updateable by yourself and others and (2) designing the product your users want, rather than what you *think* they want. Systems development is a useful concept not just for computer science projects, but for any type of product development.

For code organization, we use **Application Programming Interfaces** (APIs), which are a set of functions and procedures to allow different portions of code in a big project to communicate with each other. For user-guided product design, we use **iterative design**, which is the process of making many imperfect iterations of a project to prototype to users before trying to make the final product.

5.2 Code Organization

5.2.1 Modularity

Definition 5.2.1. **Modularity** means that code can be separated into many smaller components that are relatively independent.

Modularity has two primary benefits:

- **Teamwork**: It allows different people to work on different portions of the code without interfering with one another. This is how software companies like Google, Amazon, and Microsoft can have tens of thousands of teammembers all working on the same project.
- **Updateability**: It ensures that if one portion of the code needs to be updated or fixed, the rest of the code will not be impacted.

To illustrate this, we will consider modularity's usefulness in cars. Cars are incredibly complicated, but they can be understood by considering them as a collection of interacting components. Each component has its own functionality and relationship to the rest of the car.

A few components of a car include:

- Tires: a round object that can attach to an axle
- Engine: a device that can spin an axle
- Axle: a rod that can be connected in the center to an engine, and at the edges to tires

Figure 5.1: Tires, engine, and axle can be combined to make the core axle-tire-engine unit of a car.

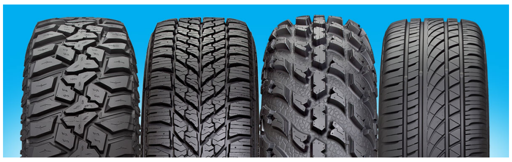


Figure 5.2: A variety of tires are available, and each must simply (1) be built to roll and (2) be connectable to an axle

There are different ways to design each of these. They can each be made from a number of materials, and there are different sizes and varieties of each (e.g. 4 vs 6 cylinder engine, snow vs all-season tires, etc.). Furthermore, any *combination* of these different varieties can be put together: we can change the type of tires we have without thinking about what type of axle or motor is in the car.

We will now look at the benefits of this modularity in car design:

- **Teamwork**: Car companies can divide up labor in multiple teams, and those teams won't interfere with each other as they work. One team can spend months developing the axle: finding the right alloy, shape, and weight to handle the load from the car. Another team can spend months developing the engine: trying out different arrangements of the cylinders or different numbers of cylinders. What's more, car manufacturers almost always outsource the production of their tires to tire manufacturers: they leave the decisions on the type of rubber and treads to a team totally outside of their company. The modularity of the car allows these teams to work totally separately, with the only communication between them the above list: the tire must roll and attach to an axle; the engine must spin the axle; and the axle must connect to the engine and the tires. Once the teams agree on the way they will bolt the systems together, they no longer have to communicate.
- **Updateability**: Modularity allows car enthusiasts to upgrade and customize their vehicles without having to redesign the whole thing. If a vehicle owner wants to replace their V6 with a V8, they can do that as long as the connections between the motor and the rest of the vehicle are the same (i.e. the connection to the axle, to the fuel source, etc). If a vehicle owner wants to replace their all season tires for snow tires in the winter, they can do that as long as the connection with the rest of the vehicle is the same (i.e. the connection to the axle is the same).

5.2.2 APIs

Definition 5.2.2. An **Application Programming Interface**, or API, is a contract between components in a system, expressing what each component can expect from the others.

Building on the previous section's example of a vehicle, we can consider the API between the engine manufacturer and the vehicle manufacturer.

The engine manufacturer can expect the vehicle to

- have an axle
- have a fuel source
- have a cooling source
- have a control system

The vehicle manufacturer the engine to

- attach to an axle
- spin at a given revolution speed

As we saw in the previous section, this type of contract allows division of labor between different teams working on a project to ensure that they can work independently: as the engine manufacturer,

Another API exists between a driver and the manufacturer: the driver can expect the car to have a

- device to turn on/off the car
- device to steer the car
- device to accelerate the car
- device to decelerate the car

The car maker can expect the driver to have

- arms and hands that can turn and push
- feet and legs that can press

Example 5.2.3: Make the API between an ATM and a user.

From the perspective of a user, an ATM's purpose is to take in a credit car and a specified dollar amount from a user, and output the requested amount of money. The user can expect an ATM to

- read their credit card
- specify the card's PIN number
- specify a requested dollar amount
- output money and notify their bank of the transaction

An ATM can expect the user to

- have a credit card
- type with their fingers
- read and respond to prompts on the screen

Note that a problem with the API translates to a problem with the end-product. This API doesn't guarantee that a blind person will be able to use the ATM, for example (they won't be able to read).

Example 5.2.4: Make the API between an ATM and a bank.

From the perspective of a bank, an ATM's purpose is to send in credit card info and the requested dollar amount, and fulfill a user request for money only if the bank authenticates the request.

The bank can expect an ATM to

- accurately send the bank credit card info and a PIN
- accurately send the bank the requested dollar amount
- complete the user's transaction only if the bank sends back a confirmation

The ATM can expect the bank to

- receive credit card info and a PIN
- send back a confirmation or denial

5.3 User-oriented Design

5.3.1 Waterfall vs Iterative Design

Example 5.3.1: Using

- 10 sticks of dry spaghetti
- one foot of string
- one foot of tape

build the highest tower possible in 6 minutes.
Adapted from https://tinkerlab.com/spaghetti-tower-marshmallow-challenge/

This activity generally demonstrates that iteration trumps pre-planning. It's faster to just trying out imperfect designs than to try to wait for a perfect idea. With a 6 minute time limit, iteration tends to work out better than pre-planning.

Definition 5.3.2. Waterfall design is a development process in which each stage of development is finished before the next is started.

The components of waterfall design, adapted from https://airbrake.io/blog/sdlc/waterfall-model, are the following:

1. Requirements: define what the application should do (essentially, write the API between a user and your product)
2. Design: decide what the product will look like based on the requirements, and how it will be implemented
3. Implementation: build the product based on the design
4. Test: ensure the product works as expected
5. Deployment: release the product to the users

In waterfall design, mistakes early in the process can kill you. You might spend a lot of time and money going through the full waterfall and developing a final product and then realize that the requirements were wrong. Going back to a previous example, suppose you had designed an ATM thinking that users would be able to read and later found out that blind users must also be able to access the ATM. You would have to completely redesign the system, perhaps having to put speakers into the ATM so that the prompts can be read aloud to the user.

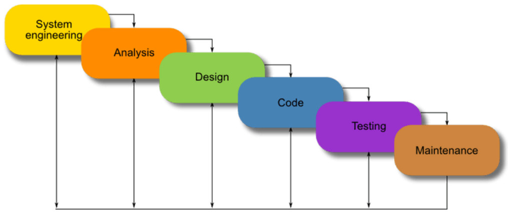


Figure 5.3: Waterfall design, courtesy https://airbrake.io/blog/sdlc/waterfall-model

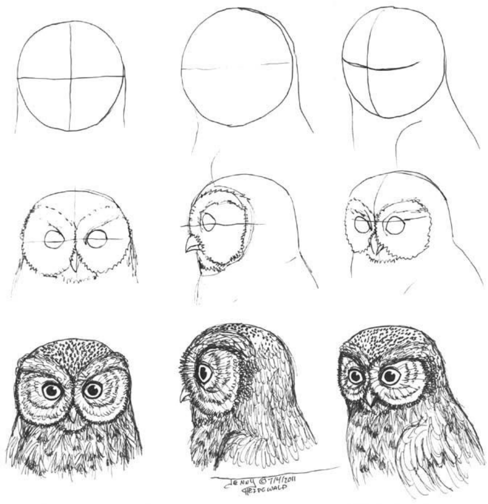


Figure 5.4: Iterative drawing of an owl. https://www.pinterest.com/pin/469289223655955022/

> **Definition 5.3.3.** In **iterative design**, many iterations of a project are built and floated to users with the expectation that the requirements and design may need to be adjusted in further iterations.

Iterative design entails going through the same 4 steps of specifying requirements, drafting a design, implementing, then deploying. However, less time and money is invested into trying to make the first iteration perfect. The first iteration of a project might look horrible, but users will be able to tell you the fundamental flaws (such as missing requirements) before you start implementing a perfect product for the wrong problem.

Iterative design is also used in drawing. As shown in Figure 5.4, professional artists start with a rough sketch of an object before starting to fill in details. The first iteration (top) doesn't look very good, but you might find out that you're missing basic requirements: you might be missing certain body parts, or decide you'd like to add a background or another object. By the second iteration, things look a little better, but you still might find more fundamental errors along the way. By the time you're ready to fill in all of the details in the bottom step, you're confident you're drawing the figure you want to draw.

5.4 Testing

Code rarely works the way we would like it to the first time around.

> **Definition 5.4.1.** A **software bug**, is an error in a computer program which causes an incorrect or unexpected result, or to behave in unintended way (Wikipedia).

Medium estimated that in 2016 approximately a trillion dollars was lost to the US economy due to software bugs. As one tangible example of the cost of bugs, in 1962

NASA's 18 million dollar Mariner 1 had to be self-destructed mid-flight because of a missing hyphen in the control code (https://medium.com/@ryancohane/financial-cost-of-software-bugs-51b4d193f107).

For this reason, developers are always expected to carefully test the code they write to ensure there are no bugs. In addition, even relatively small software firms often have an entire team whose sole job is to test code written by the developers. Their job is to use the product in a variety of potentially unexpected ways. In essence, they try their best to break the code. The product is not ready for the market until the testing team is unable to break it, and all of its behaviors are as expected by the API for the product.