Software

- Software - code that runs on the hardware
- I'm going to simplify things a bit here
- CPU implements "machine code" instructions

  --Each machine code instruction is extremely simple

  --e.g. add 2 numbers

  --e.g. compare 2 numbers
- Code we have seen `pixel.setRed(10)`

  --To run, expanded to about 10 machine code instructions

MACHINE CODE

"Software" is the general category of code which runs on the hardware. If the hardware is a player piano, then the software is the music. The common case is a "program" like Firefox -- software you run on your computer to solve a particular problem. A computer can run multiple programs at the same time, keeping their use of memory, drawing in windows etc. separated so they hopefully do not interfere with each other.

A CPU understands a low level "machine code" language (also known as "native code"). The language of the machine code is hardwired into the design of the CPU hardware; it is not something that can be changed at will. Each family of compatible CPUs (e.g. the very popular Intel x86 family) has its own, idiosyncratic

machine code which is not compatible with the machine code of other CPU families.

INSTRUCTIONS AND PROGRAMS

The machine code defines a set of individual instructions. Each machine code instruction is extremely primitive, such as adding two numbers or testing if a number is equal to zero. When stored, each instruction takes up just a few bytes. When we said earlier that a CPU can execute 2 billion operations per second, "operations" there refers to these simple machine code instructions.

A program, such as Firefox, is made up of a sequence of millions of these very simple machine code instructions. It's a little hard to believe that something as rich and complicated as Firefox can be built up out of instructions that just add or compare two numbers, but that is how it works. A sand sculpture can be rich and complicated when viewed from a distance, even though the individual grains of sand are extremely simple.

WHAT IS A PROGRAM?, WHAT IS RUNNING?

- "Program" e.g. Firefox, millions of simple machine code instructions

  --Instructions, like grains of sand making up sculpture

- CPU runs a "fetch/execute cycle"

  --fetch one instruction in sequence

  --execute (run) that instruction, e.g. do the addition

  --fetch the next instruction, and so on

- "loop" instruction: jump back 10 instructions

  --Loops are implemented this way

- "if" instruction: skip ahead if a certain condition is true

  --If statements are implemented this way

The CPU runs instructions using a "fetch-execute" cycle: the CPU gets the first instruction in the sequence, executes it (adding two numbers or whatever), then fetches the next instruction and executes it, and so on. Some of the instructions affect the order that the CPU takes through the instruction sequence .. for example an instruction might direct the CPU to jump back to an earlier point in the instruction sequence (loops are implemented this way), or to skip over the next instruction if a particular condition is true (if-statements are implemented this way).

DOUBLE CLICK A PROGRAM

- What is a program, like Firefox.exe (.exe is a Windows convention)
- The file Firefox.exe is mostly the bytes of millions of instructions
- Double click Firefox.exe to Run

  --The instruction bytes are copied up into RAM

  --The CPU is directed to start running at the first instruction

In the file system, a file like Firefox.exe just contains the bytes of the machine code instructions that make up the program (".exe" is a windows convention to mark a file as a program). Each machine code instruction takes up about 4 bytes, and whole program is just an enormous sequence of instructions.

On my machine, Firefox is 80 MB in size. Assuming all those bytes are instructions, and each instruction is 4 bytes, how many machine code instructions make up Firefox? (Answer: 80 MB is 80 million bytes, so that would be 20 million instructions.)

When the user double clicks a program file to run it, essentially the block of bytes of the instructions for the program are copied into RAM, and then the CPU is directed to begin running at the first instruction in that area of RAM.

OPERATING SYSTEM

- Who starts Firefox?
- Operating System
- Set of supervisory programs, run when computer first starts
- Administration behind the scenes
- Starting/managing/ending other programs

  --Modern computers can run multiple programs at the same time

  --Operating system keeps each program run isolated

  --Program has its own RAM, its own windows on

screen

--vs. accidental or malicious action between programs

- e.g. Laptop
- e.g. Digital camera

The "operating system" of a computer is like a first, supervisory program that begins running when the computer first starts up ("boots up"). The operating system plays an invisible administrative and bookkeeping role behind the scenes. When a desktop or laptop starts up, the operating system typically gets things organized and then launches a "file explorer" program which displays windows and menus etc. that show the user what file systems are available, allowing the user to navigate and operate on the files.

The operating system keeps things organized in the background so that multiple programs can run at the same time, which is known as "multitasking". The operating system gives each program its own area of memory, so each program only accesses its own resources. attempting to limit what an erroneous or malicious program can do. Keeping the programs separate is sometimes known as "sandboxing" . mediating the access of each program so it operates independently, without interfering with other programs or the system as a whole. Similarly, each program has some access to the screen through a window, but this output area is separated from the output of other programs.

Recall that a .exe file or whatever is essentially just a file of machine code instructions. When you double-click the program, it is the operating system that "launches" the program, doing the housekeeping steps of allocating an area of memory within RAM for the program, loading the first section of the program's machine code into that memory, and finally directing the CPU to start running that code.

A digital camera is also a little computer. When it starts up, it does not run a file manager program. Instead, after the basic housekeeping is set up, the camera may just run a single program that draws the menus etc. on the camera's screen and responds to clicks on the camera's buttons and so on.

BOOT / REBOOT

- Chicken and egg problem. who runs the operating system?
- When first powered on, computer runs a tiny "powered on" program
- That program typically looks for a disk containing an operating system to run
- Etymology: "lift self over a fence by pulling on your bootstraps"
- Boot up -- start
- Reboot -- shutdown/start-fresh cycle

Computer Languages

From Programmer to the CPU

- We've written small Javascript programs
- We've seen large program like Firefox

- Computer language used by a person (e.g. Javascript)
- vs. the simple machine code instructions in the CPU
- What's the connection?
- (Here the basic themes, not the details)

COMPUTER LANGUAGES

It is extremely rare to write machine code by hand. Instead, a programmer writes code in a more "high level" computer language with features that are more useful and powerful than the simple operations found in machine code. For CS101, we write code in Javascript which supports high level features such as strings, loops, and the print() function. None of those high level features are directly present in the low level machine code; they are added by the Javascript language. There are two major ways that a computer language can work.

SOURCE CODE AND COMPILER

One common computer language strategy is based on a "compiler". The computer languages C and its derivative C++ are old and popular computer languages that use this strategy, although they tend to have fewer features than dynamic languages (below).

In C++, the programmer writes C++ code which includes high level facilities such as strings and loops (much as we have seen in Javascript). Here is some C++ code to append a "!" at the end of a string.

- Computer languages -- "high level" features

  --e.g. loops, if-statements, strings

- e.g. C, C++, Javascript, Java
- Programmer writes "source code" of a program in a language, say, C++
- Example C++ code -- how to get to the CPU?

```
// C++ code

a = "hi";

b = a + "!";
```

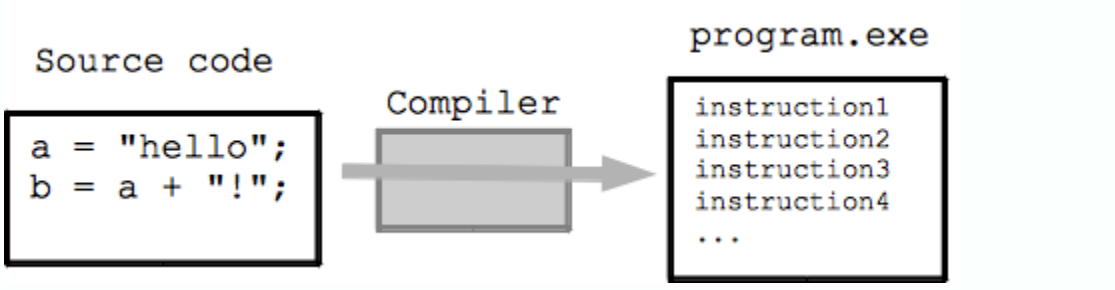
This code appends the string "!" on to the end of "hi", resulting in the string "hi!" stored into the variable b. The machine code instructions in the CPU are too primitive to implement this append operation as one or two instructions. However, the operation can be accomplished by a longer sequence of machine code instructions strung together.

COMPILER

- "Compiler" looks at the source code
- Compiler **translates** the source code into a large number of machine code instructions
- Suppose a high level construct, like an if-statement, can be implemented by a sequence of 5 machine code instructions

- e.g. Firefox -- written in C++

  --Compiler takes in Firefox C++ source code, produces Firefox.exe
- The compilation step can be done once and long before the program is run (e.g. produce Firefox.exe at Mozilla headquarters)
- The end user does not need to the source code or the compiler. Distribute the program.exe file in working form
- Does not work backwards -- having the .exe, you cannot recover the source code (well)



The **Compiler** for the C++ language, reads that C++ code and translates and expands it to a larger sequence of the machine code instructions to implement the sequence of actions specified by the C++ code. The output of the compiler is, essentially, a program file (.exe or whatever) made of many machine code instructions that implements the actions specified in the C++ code. The compiler produces the .exe file from the C++ code, and it is finished. Running the .exe can happen later, and is a separate step.

SOURCE CODE

- Having the .exe allows one to run the program
- To add feature or fix a bug, ideally you want the source code

  --Add a feature in the source code, then run the compiler again to make a new version of the .exe
- **Open Source** software
- The source code is available to all, along with the right to make modifications
- 1. Typically the software does not cost anything
- 2. **Freedom** the end user is not dependent on the original vendor to fix bugs, or perhaps the vendor goes out of business
- The user can make the change themselves (since they have access to the source)
- Insurance/freedom policy
- Often the license terms will require the change to made available to the wider community in some cases ... sharing back to the community

- Open source is a very successful model for some cases
- Talk about this more later, mentioning now since it is so close to the idea of what "source code" is

The "source code" is the high level code authored by the programmer and fed into the compiler. Generally just the program.exe file is distributed to users. The programmer retains the source code. Changing the program in the future generally requires access to the source code. For example to add a feature, the programmer would make changes in the source code, and then run the compiler to produce a new version of the program.

OPEN SOURCE

"Open Source" refers to software where the program includes access to its source code, and a license where the user can make their own modifications. Typically open source software is distributed for free. Critically, beyond the free price, open source software also includes **freedom/independence** since the user is not dependent on the original vendor to make changes or fixes or whatever to the source code. Since the source code is available, if a user feels strongly enough about some feature, they can add the feature themselves, or pay someone to add the feature. Open source means you are not dependent on some other part .. attractive as software is such a critical part of many organizations. Typically open source licenses include a requirement that such improvements to the source code be made available back to the community at large. We'll talk about open source more later on, but I wanted to touch on it here since it is a good example of the difference between a program and its source code.

DYNAMIC (INTERPRETER) LANGUAGES

- Dynamic languages (big tent here)

- e.g. Java, Javascript, Python
- Can be implemented by an "interpreter"

There is a broad category of more modern languages such as Java (the world's most popular language, used in Stanford CS106A), Javascript, and Python, which do not use the compiler/machine-code strategy. Instead, these languages can be implemented by an "interpreter", and I will lump them into the category of "dynamic" languages.

INTERPRETER

- Interpreter is a program which "runs" other code
- e.g. web browsers includes a Javascript interpreter --Browser "runs" bits of Javascript code in a web page, such as ours
- Interpreter looks at one line at a time
- Deconstructs what each line says to do
- The interpreter then does that action, in the moment
- Then proceeds to the next line

So in Javascript when we have code lines like:

```
// Javascript code

a = 1;

b = a + 2;
```

- e.g. Interpreter looks at `a = 1;`, does it
- e.g. Interpreter looks at `b = a + 2;`, does it
- The compiler **translates** source code to equivalent machine code
- The interpreter **does** the code, looking at each line and doing it

An interpreter is a program which reads in source code as its input, and "runs" the input code. The interpreter proceeds through the code given to it, line by line. For each line, the interpreter deconstructs what the line says and performs those actions, piece by piece. For example, Javascript which we have been using, is implemented by a Javascript interpreter which is built into Firefox.

The interpreter runs this code, by taking the lines one at a time, and for each, interpreting its actions. For "a = 1;" the interpreter reserves a few bytes to store the value of a, then stores the value 1 into those bytes. Then for "b = a + 2;" the interpreter evaluates (a + 2) getting the value 3, reserves some bytes for the b variable, then stores the 3 into the b bytes.

A **compiler** translates all the source code into equivalent machine code program.exe to be run later -- it is a bulk translation.
An **interpreter** looks at each line of code, and translates and runs it in the moment, and then proceeds to the next line of source code. The interpreter does not produce a program.exe, instead it performs the actions specified in the source code directly.

COMPILER VS. INTERPRETER EVALUATION

- Disclaimer: there are many languages, no one "best" language, it depends!

- Compiled code tends to run faster
- The .exe tends to be "lean" .. compiler has removed overhead, some decisions
- Dynamic/interpreter languages

  --Tend to have a greater number of programmer-friendly features

  --i.e. programmers are often more productive in dynamic languages

  --The resulting program tends to run somewhat slower than compiled code

Compiled code generally runs faster than interpreted code. This is because many questions -- how to append to this string, how many bytes do I need here -- are resolved by the compiler at compile time, long before the program runs. The compiler has, in effect, pre-processed the source code, stripping out many questions and complications, leaving the program.exe as lean and direct as it can be to just run.

In contrast, the interpreter deals with each line in the moment, so all the deciphering and overhead costs of interpreting each line are paid as it runs. These overhead costs in effect make the interpreted program run more slowly than the equivalent compiled program.

DYNAMIC LANGUAGES - MORE FEATURES / SLOWER

- Dynamic languages tend to have more features (programmer-friendly)
- e.g. Memory Management

  --C and C++: partially manual, some programmer input required

  --Dynamic languages: automatic memory management, no programmer input needed

  --Automatic memory management not free: spending CPU cycles to lighten programmer workload
- Tradeoff

  --Dynamic languages often allow the programmer to get things done faster

  --However the dynamic code runs a bit more slowly compared to compiled code
- Current trend is towards dynamic languages

  --**Programmers** are scarce

  --It's attractive to save some programmer time at the expense of some CPU/memory use

  --Moore's law reinforces: CPU cheap, programmer relatively rare/expensive

Supporting features tends to be easier in dynamic languages compared to compiled languages, which is why dynamic languages tend to have a greater number of programmer-friendly features.

"Memory management" is the problem in a program of knowing, over time, when bytes of RAM are needed and when they can be reclaimed and use for something else. Every program must solve this problem. Memory management is an excellent example of a feature different between compiled and dynamic languages -- most modern dynamic languages manage memory automatically. The programmer can focus on the problem to be solved, and the dynamic language will take care of managing the memory.

In contrast, in C and C++, the programmer at times must think about memory management at times, and may have to author lines of code to help solve it. (Aside: many crashes in C and C++ programs are due to errors in the programmer's memory management scheme. It is a difficult problem to solve manually.)

The memory management in dynamic languages is not free. The CPU must run extra lines to solve the memory management. Dynamic languages, in effect, spend CPU cycles to manage the memory. This fits the general pattern that dynamic languages run with more overhead (i.e. more slowly) than compiled languages, but offer superior programmer-friendly features.

Because dynamic languages like Java and Python have more features, a programmer can often write the code to solve a problem more quickly in a dynamic language than they can in C++. The time and attention of programmers is generally quite scarce (translation: programmers are scarce and expensive, which is why you want to be a CS major, or at least a minor!). Therefore, dynamic languages which allow the programmer to produce a correct program more quickly and reliably are pretty attractive, even if the resulting program uses more CPU and more RAM. Aside: Moore's law in effect, keeps making the programmer relatively more expensive compared to the CPU.

Overall, different computer languages have different strengths and weaknesses, and best language for a particular problem depends on the situation. As above, dynamic languages like Java and Python can run slower and generally operate with higher overhead than C++ code, so for some problems, writing in C or C++ is the best

strategy. Also, Java and Python lack certain "low level access" features which are needed in rare cases.

JIT JUST IN TIME COMPILER

- JIT -- compile code of a dynamic language on the fly
- All major browsers now have a JIT for the Javascript code they run (Chrome)
- Best of both worlds
- Flexibility of dynamic languages
- Combined with most of the performance of the compiled world
- Active area of research, works pretty well

The most modern form of dynamic language is implemented with an interpreter paired with a Just In Time compiler (JIT) trying to get the best of both worlds. The JIT looks a sections of dynamic code that are being run very frequently, and for those, does a compile to native code for that section on the fly. So the interpreter is used for simple cases, but for important sections of dynamic code (like the inside of a loop), the JIT creates a block of machine code in RAM for that section. The machine code is run for that section of dynamic code, giving similar performance to C++, and is discarded when the program exits. Java and Javascript both use JIT technology extensively. The great speedup of browsers in the last few years has been largely due to the implemented of JIT technology for Javascript. The JIT erases most but not all of the "10x" penalty. Even with JITs, dynamic languages still have higher overhead use of resources compared to C and C++.