## **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** 

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.

derivative C++ are old and popular computer languages that use this strategy, although they tend to have fewer

One common computer language strategy is based on a "compiler". The computer languages C and its

- 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

strung together. Compiler

instructions. However, the operation can be accomplished by a longer sequence of machine code instructions

- "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

program.exe

instruction1 instruction2

instruction3

- Does not work backwards -- having the .exe, you cannot recover the source code (well)
- instruction4 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

modifications

the source)

Source code

"hello";

 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

Compiler

- new version of the .exe **Open Source** software • The source code is available to all, along with the right to make
- 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
- 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** 

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

## "Open Source" refers to software where the program includes access to its source code, and a license where the

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. Interpreter

languages can be implemented by an "interpreter", and I will lump them into the category of "dynamic"

# Interpreter is a program which "runs" other code

--Browser "runs" bits of Javascript code in a web page, such as ours Interpreter looks at one line at a time

e.g. web browsers includes a Javascript interpreter

- 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;

the actions specified in the source code directly.

- e.g. Interpreter looks at a = 1; , does it • e.g. Interpreter looks at b = a + 2; , does it
- 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 =

The compiler translates source code to equivalent machine code

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

**Compiler vs. Interpreter Evaluation**  Disclaimer: there are many languages, no one "best" language, it depends!

#### The .exe tends to be "lean" .. compiler has removed overhead, some decisions Dynamic/interpreter languages

Compiled code tends to run faster

- --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

slowly than the equivalent compiled program.

lighten programmer workload

compiled code

CPU/memory use

superior programmer-friendly features.

expensive compared to the CPU.

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.

**Dynamic Languages - More Features / Slower** Dynamic languages tend to have more features (programmer-friendly)

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

#### e.g. Memory Management --C and C++: partially manual, some programmer input required --Dynamic languages: automatic memory management, no programmer input needed

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

--However the dynamic code runs a bit more slowly compared to

--Automatic memory management not free: spending CPU cycles to

- Current trend is towards dynamic languages -- **Programmers** are scarce --It's attractive to save some programmer time at the expense of some
- 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

solved, and the dynamic language will take care of managing the 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

pattern that dynamic languages run with more overhead (i.e. more slowly) than compiled languages, but offer

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

Overall, different computer languages have different strengths and weaknesses, and best language for a

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

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

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

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
- 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

Active area of research, works pretty well

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++.