Part 1: What Is Programming?

Programming and Computers (1)

- In order to understand what programming is, we need to know what a computer is
- A computer is a machine that executes lists of instructions
  - We feed a list of instructions to the computer and the computer executes them
  - The computer may apply the instructions on additional information fed to the computer (the input)
  - The computer may produce information as a result of executing this list of instructions (the output)
- Programming a computer involves two things:
  - Designing lists of instructions that will make the computer solve specific problems

Programming and Computers (2)

- Having the computer execute the instructions
- The purpose is to have the computer solve the problem for you instead of you solving the problem by hand
Lists of Instructions

• In order to be able to write instructions that will enable someone (or something) to solve a problem, we have to understand how we solve this problem when we solve it ourselves "by hand"

• Suppose you had to write instructions that your 6-year old cousin could follow in order to perform these tasks:
  – Cook scrambled eggs
  – Find a given card in a deck of cards
  – Count the number of cards in a deck that are of a given suit
  – Convert a monetary amount from one currency to another

• What would you write?

Cooking Scrambled Eggs (1)

• Input (things we need to follow the instructions):
  – Two eggs
  – A tablespoon of oil
  – A pan
  – A stove

• Instructions:
  – Add oil to pan
  – Heat pan on stove
  – Crack eggs into pan
  – Mix until light and flaky

Cooking Scrambled Eggs (2)

• Output (things that result of following the instructions):
  – Scrambled eggs

Finding a Card In a Deck (1)

• Input:
  – The deck
  – The card we are looking for

• Instructions:
  – Assume we have not found the card
  – While we have not found the card, and there still cards in the deck that we have not looked at, do the following:
    • If the current card is the card we are looking for, then we have found the card we were looking for; stop
    • Otherwise, move on to the next card in the deck
  – We have not found the card
Finding a Card In a Deck (2)

- **Output:**
  - Whether the card we are looking for was *found* in the deck

Counting Cards of a Given Suit (1)

- **Input:**
  - The *deck*
  - The *suit* we are looking for

- **Instructions:**
  - Take a blank piece of paper
  - While there are still cards in the *deck* that we have not looked at, do the following:
    - If the current card is the card belongs to the *suit*, make a *mark* on the piece of paper
    - Move on to the next card in the *deck*
  - Number of cards that belong to the *suit* = number of *marks* on the piece of paper

Counting Cards of a Given Suit (2)

- **Output:**
  - The number of cards in the *deck* that belong to the *suit*

Currency Exchange

- **Input:**
  - *Amount*
  - *Source currency*
  - *Desired currency*

- **Instructions:**
  - Look up current exchange rate for desired currencies in table of exchange rates
  - Calculate *converted amount* as *amount* * exchange rate

- **Output:**
  - *Converted amount*
More on Instructions

• Suppose your cousin is not very creative and follows your instructions exactly
  – What happens if you forget to mention some details?
  – What happens if you change the order of the instructions?
• For example, in the instructions for cooking scrambled eggs:
  – To what temperature should one "heat the pan on the stove"? Does it matter? What will a person following the instructions do if it matters but it's not specified?
  – What happens if one tries to "mix until light and flaky" before "cracking the eggs into the pan"?

Instructions and Precision

• The instructions for finding a card in a deck or counting the number of cards of a given suit in a deck are very precise and unambiguous
• Your 6-year old cousin (or anyone else) will be able to follow them and arrive at the solution, even if
  – he / she has no creativity or imagination
  – he / she has no ability to fill the gaps in your instructions (because you did not leave any gaps in your instructions)
  – he / she does no understanding of the purpose of your instructions or what they are supposed to accomplish, and does not care
• Writing lists of instructions like these is the very essence of programming a computer

Programs (1)

• Programs are essentially lists of instructions that tell a computer how to accomplish a task
  – Each instruction tells the computer to do something: an action, a calculation, a comparison
• However, the computer does not understand the instructions or their purpose, it merely performs them without question
• Therefore, your instructions must be precise
  – The computer cannot fill the gaps if you forget to specify certain things
  – In those cases, it will either return the wrong result or choke
• Your instructions must also be ordered correctly
  – The computer will not be able to detect incorrectly ordered instructions

Programs (2)

– Again, in those cases, it will merely return the wrong result or choke
• Programs also have:
  – Input (things that they work on)
  – Output (things that they produce)
• But programming a computer is consists of more than just writing precise and unambiguous lists of instructions...
Computer Languages

- The computer does not understand English or any other human language
- Therefore, not only must your lists of instructions be precise and unambiguous, they must be translated from human language to a language the computer understands
- Learning this language is similar to learning a foreign (human) language, but there are important differences

Human and Computer Languages

- All languages that computers understand have a special structure which ensures that no valid statement expressible in these languages is ambiguous
  - Each statement has one possible meaning and only one possible meaning
  - Compare to the following English sentences:
    - "The lady hit the man with an umbrella"
    - "Time flies like an arrow"
- Computers are much more intolerant than humans of instructions which contain what would be spelling or grammatical errors in human languages
  - If your instructions contain such mistakes, the computer will simply not understand them

Complex Tasks

- Sometimes, we want the computer to perform complex tasks
- Writing a detailed list of precise instructions for this complex task would be very difficult
  - There are just too many instructions for humans to manage
- Solution:
  - Break down the complex task into a series of simpler tasks; if the simpler tasks are still too complex, break them down into even simpler tasks
  - Write a list of instructions for each of the simpler tasks
  - The list of instructions for each of the simpler tasks can be used as a single instruction in the list of instructions that performs the original task

The Bank Machine

- A bank machine has a lot of functionality:
  - It checks whether you are who you say you are
  - It gives you a choice of operations
  - Each operation does something different
  - It ensures that you satisfy the necessary conditions to perform the operation you have selected
- Let's try to break this complex task down
The Bank Machine: Login

- **Input**
  - The customer's *bank card*

- **Instructions:**
  - Set the *number of login attempts* to 0
  - While the *number of login attempts* is less than the *number of authorized attempts*, do the following
    - Read the *PIN* from the keyboard
    - If the *PIN* matches the one associated with the *bank card*, then *success*, otherwise, add one to the *number of login attempts*
  - *Failure*

- **Output**
  - *Success* if the login succeeds, *failure* otherwise

The Bank Machine: Main Program

- **Input:**
  - The customer's *bank card*

- **Instructions:**
  - Perform *login* subprogram
  - If *failure*, keep the *bank card* and wait for next customer; otherwise
    - Ask user for *operation*
    - While *operation* is not *exit*, perform subprogram associated with *operation* and ask user for another *operation*
    - Return *bank card* and wait for next customer

- **Output:**
  - (Depends on the operations requested by user)

The Bank Machine: Operations

- What would be a suitable list of instructions for:
  - Withdrawing money from an account?
  - Transferring money from one account to another?
  - Deposit money into an account?
  - Pay a bill?

- What does each operation require for input?
- What does each operation produce for output?

Software

- Is a software package merely a program?
- Software typically consists of many subcomponents, modules, and subprograms
- Each subprogram solves a particular problem or performs a particular task
- Software also includes some information to be used and manipulated by the subprograms
  - For example, in a currency conversion program, it includes a table with exchange rates
Goals of This Course (1)

• To learn how to design and write precise, unambiguous, and detailed lists of instructions that can be used to solve a problem or perform a task
  – Requires imagination, rigor, and understanding of how we solve the problem "by hand"
• To learn how to translate these instructions into a language the computer can understand
  – Requires remembering the "words" of this language, how they are "spelled", what they mean, what the "grammar" rules of this language are; similar to learning a foreign (human) language
• To learn how to deal with complex problems or tasks by:
  – Breaking them down into smaller subproblems or subtasks

Goals of This Course (2)

– Writing subprograms which solve each subproblems or perform each subtask
– Connecting the subprograms that solve each subproblem or perform each subtask to form one program which solves the original complex problem or performs the original complex task
• In short, to learn how to develop software
• At the end of this course, you will be able to write a program that simulates a bank machine (and many other things)

Part 2: How a Computer Works

Hardware and Software

• Hardware consists of the physical, tangible parts of a computer
  – Cases, monitors, keyboard, mouse, chips
  – Rule of thumb: If you can take it in your hands and it is part of a computer system, then it is hardware
  – It is the hardware which executes the instructions
• Software: Programs and data that they use
• A computer requires both hardware and software
  – Software cannot run without hardware; instructions are useless unless they are performed by someone / something
  – Hardware will not do anything without software telling it what to do
  – Therefore, each is essentially useless without the other
A Personal Computer

- Monitor / screen (output)
- Case; contains:
  - CPU
  - Memory
  - Disk drives
  - ...
- Speakers (output)
- Keyboard (input)
- Mouse (input)

Central Processing Unit (CPU)

- The "brain" of the computer
  - Basically controls the information / data in a computer
- Perform instructions
  - Arithmetic operations
  - Logic operations
  - Decisions
- The instructions it understands are much simpler and fine-grained than those we have seen in previous examples

Memory

- Memory holds the data
  - Think of a filing cabinet
- Main memory: Most of it is called RAM, which stands for Random-Access Memory
  - Data has to be in main memory so that the CPU can access it
  - Volatile: its contents are lost when the program terminates or when the computer shuts down
- Secondary storage: Hard drive / CD / DVD / Blu-Ray disc / USB memory stick
  - Persistent: its contents will not be lost when the computer shuts down
  - This is where you keep the data for long-term storage
  - Secondary storage has much slower access times than main memory

Devices

- Devices are hardware components which the CPU can access and control
  - Input devices are used to feed information to the computer
  - Output devices is what the computer uses to give information back to the user
  - Some devices are both input and output devices
- Examples of devices:
  - Cards (Video card, sound card, network card, ...)
  - Optical (CD / DVD / Blu-Ray) drives
  - USB ports
  - ...

Motherboard

- Acts as a bridge between the CPU, memory, disks, and other devices
- Data transfers taking place between the hardware components of a computer take place via the motherboard

Main Memory Organization

<table>
<thead>
<tr>
<th>Address</th>
<th>Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>9278</td>
<td></td>
</tr>
<tr>
<td>9279</td>
<td></td>
</tr>
<tr>
<td>9280</td>
<td></td>
</tr>
<tr>
<td>9281</td>
<td></td>
</tr>
<tr>
<td>9282</td>
<td></td>
</tr>
<tr>
<td>9283</td>
<td></td>
</tr>
<tr>
<td>9284</td>
<td></td>
</tr>
<tr>
<td>9285</td>
<td></td>
</tr>
</tbody>
</table>

Main memory is divided into many memory locations (or cells).

Each memory cell has a numeric address which uniquely identifies it.

Each cell contains a data value (for example, 22)

Hardware Interaction

Bits and Bytes

- Memory quantities are measured in bits and bytes
- 1 bit: most basic unit of memory
  - 1 or 0, yes or no, true or false, on or off
- 1 byte = 8 bits
  - 1 byte can therefore represent $2^8 = 256$ different values
- In a computer, data values are stored as sequences of bits
  - But in memory, one cell contains one byte, **not** one bit

<table>
<thead>
<tr>
<th>Address</th>
<th>Data Value</th>
<th>Binary Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9278</td>
<td>22</td>
<td>00010110</td>
</tr>
<tr>
<td>9279</td>
<td>7</td>
<td>00000111</td>
</tr>
</tbody>
</table>
A program tells the CPU how to manipulate and/or move information.

The CPU repeatedly performs the three following operations:
- Reads the next instruction in the program
- Figures out what the instruction means
  - add two values?
  - load some value from memory?
  - store some value in memory?
  - compare two numbers?
  - ...
- Performs the instruction

This is called the *fetch-execute cycle*.

Suppose you want to write a program that reads a number from the keyboard, adds 1 to it, and displays the new value to the screen.

This program might consist of the following instructions:
- **READ** a value from the keyboard and store it in memory location \(x\)
- **LOAD** the value stored in memory location \(x\) into the CPU
- **ADD** 1 to the value stored in the CPU
- **STORE** the value currently in the CPU back into memory location \(x\)
- **DISPLAY** the value stored in memory location \(x\) to the screen
Programming Languages (1)

- A *programming language* specifies the words and symbols that we can use to write a program
  - We know that a program is a series of instructions, and that these instructions are performed by the computer
  - We use this language to explain the instructions to the computer
- A programming language employs a set of rules that dictate how the words and symbols can be put together to form valid *program statements*
  - Just like human languages have rules that dictate what words are valid in the language, and how they can be put together to form valid sentences

Programming Languages (2)

- Computers are very intolerant of incorrect programming language statements
  - Humans are much more tolerant of incorrect natural language statements

Ambiguity

- Human languages are ambiguous
  - Consider the following sentence: "The lady hit the man with an umbrella"
  - Did the lady use an umbrella to hit the man, or did she hit a man who happened to be carrying an umbrella?
- Programming languages cannot be ambiguous
  - Each statement can only have one meaning

Syntax and Semantics

- The *syntax rules* of a language define how what words and symbols are valid in this language, and how they can be combined to make a valid program
- The *semantics* of a program statement define what those symbols and statements mean (their purposes or roles in a program)
  - A program that is syntactically correct is not necessarily logically (semantically) correct
  - A program will always do what we tell it to do, not what we *meant* to tell it to do
Machine Language (1)

- Each instruction that a CPU understands is represented as a different series of bits
  - The series of bits are themselves represented by currents flowing into wires
  - For example, a current flowing in the wire could mean 1, and no current flowing in the wire could mean 0
  - The set of all instructions that a CPU understands directly forms the machine language for that CPU
- Each CPU type understands a different machine language
  - In other words, for each different model of CPU, a given series of bits could mean a different instruction

Machine Language Example

- Here is the beginning of a machine language program that displays "Hello!" to the screen
  - CPU: Intel Core 2 Duo T7250 (Machine language: x86-64)
  - Operating System: Ubuntu GNU/Linux 8.04
    01111111 01000101 01001100 01000110 00000010
    00000001 00000001 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000
    00000000 00000010 00000000 00111110 00000000
- Those are only the first 20 bytes of more than 8800!

Machine Language Disadvantages

- Very tedious and confusing: machine language is extremely difficult for humans to read
- Error-prone
  - If you change one bit from 1 to 0 (or vice-versa), or forget a bit, your program's behavior will likely be not even close to what you expected
  - Moreover, errors are hard to find and correct
- Programs are not portable
  - Running the program on a different processor or CPU requires a complete rewrite of the program

- For example, on an x86-compatible CPU (Intel, AMD), the series of bits 10101010 could mean ADD, while on a PowerPC CPU (old Macs, PlayStation 3) it could mean LOAD
High-Level Languages (1)

• To make programming more convenient for humans, *high-level languages* were developed

• Basic idea:
  – Develop a language that looks like a mix of English and mathematical notation to make it easier for humans to read, understand, and write it
  – For each CPU type, develop a program that translates a program in high-level language to the corresponding machine language instructions
  – One instruction in a high-level language could become many machine language instructions

• Note that no CPU understands high-level languages directly
  – Programs written in these languages must all be translated in machine language before a computer can run them

High-Level Languages (2)

• Advantages of high-level languages
  – Much easier for humans to read and write in
  – Errors are easier to detect and correct
  – Portable: No need for a complete rewrite of the program as long as there exist a translation tool that translates the high-level language to the machine language for a given CPU

• Today, almost all programs are written in high-level languages and then translated to machine language using translation programs

High-Level Language Example

• Here is a complete program that displays "Hello!" to the screen, written in a high-level language called C:
  ```c
#include <stdio.h>

int main(void) {
    printf("Hello!\n");
    return 0;
}
```

• The machine language program whose first 20 bytes we saw earlier is the result of translating the above program in machine language using a translation program

Compilers (1)

• Translation from a high-level language to machine language for a particular type of CPU can be accomplished in several ways

• A *compiler* is a software tool which translates *source code* into a specific *target language*; often, that target language is the machine language for a particular CPU type
  – Input: files written in a high-level programming language
  – Output: file containing the equivalent instructions in the target language; often an executable binary file that can be processed by CPU directly

  – A different compiler is required for each CPU type
Compilers (2)

Source code (high-level)

Compiler (to CPU 1) → Compiler (to CPU 2)

Binary code (CPU 1) → Binary code (CPU 2)

CPU 1 → CPU 2

Interpreters (1)

- An interpreter is another software tool which takes source code and translates it into a target language
  - However, the target language instructions it produces are executed immediately
  - No executable file is created

Interpreters (2)

Source code (high-level)

Interpreter (for CPU 1) → Interpreter (for CPU 2)

CPU 1 → CPU 2

Programming Errors (1)

- A program can have three types of errors
  - Compile-time errors: the compiler finds problems with syntax and other basic issues
    - Compile-time errors occur mostly because the programmer tried to use words or symbols which are not defined in the language, or to combine valid words or symbols in ways the language does not allow
    - If compile-time errors exist, an executable version of the program is not created
    - Compile-time errors are also called syntactic errors
Programming Errors (2)

- **Logical errors**: the program runs, but produces incorrect results
  - `celcius = (5.0 / 9.0) * fahrenheit - 32;`  
    // Should be `(5.0 / 9.0) * (fahrenheit - 32)`
- **Run-time errors**: a problem occurs during program execution, and causes the program to terminate abnormally (or crash)
  - Division by 0
  - User enters data of the wrong type
- Both logical errors and run-time errors are **semantic errors**
  - The compiler cannot detect these kinds of error; the programmer can only become aware of their existence when the program executes

Compiling and Running Programs

- Type your program using a text editor (*not* a word processor) or an IDE (Integrated Development Environment)
- Save your program
  - Store all your files in one folder / directory for now
- Compile the program
  - If there are syntax errors, the compiler will not generate the target language program; it will report the errors instead
  - In this case, fix the program, save it, and try again
- Run the program and observe the results
  - If there are logical errors or run-time errors, they will be detected in this phase
  - In this case, fix the program, save it, recompile it, and try again

Development Life Cycle

```
Write program → Compile program → Run program
```

Errors may take a long time to debug!

**Important Note**: When you compile for the first time and see 150 errors, do not despair. Only the first 1 or 2 errors are relevant. Fix those and compile again. There should be fewer errors (like 50). Repeat until there are no more errors.