CS101
Overview of Programming, Virtual Machines, System Software

What is programming? Quite simply it is planning or scheduling the performance of some task or event. In the context of computers, computer programming is the planning of a sequence of steps for a computer to follow. Consequently, a computer program is a list of instructions to be performed by a computer.

Writing a program

There is no little man inside the computer. It is a mindless automaton that only does exactly what you tell it to do. So you, the human programmer, must first design a solution to your problem, and then implement it. Here is one lifecycle for designing and implement computer systems:

1. Problem-Solving and Specification Phase
    a. Analysis, Specs of the problem and solution
    b. Design solution, algorithm
    c. Verify solution
2. Implementation Phase
    a. Translation solution into program
    b. Testing
    c. Debugging
3. Maintenance Phase
    a. Use
    b. Maintain, perhaps looping back to step 1

There are many other development methodologies, many of which are discussed in more detail in an Information Systems or Software Engineering course. We will weave some of these concepts into this course as we progress. One of the big dangers of programming is the temptation to jump straight to the implementation. This is dangerous because it may result in a sloppy solution that was not thoroughly designed.

Algorithms to Programs

A key component prior to implementing a computer program is the design of an algorithm. As we have seen before, an algorithm is simply a sequence of steps to solve a particular program.

After the algorithms have been designed and analyzed, programs can be constructed. This is the process of converting the English algorithms to a strict set of grammatical rules that are defined by the programming language. There is syntax to the rules as well as semantics. Syntax refers to the order of instructions, like grammatical rules ("favorite the 201 class computer" is syntactically incorrect). Semantics refers to the understanding of the syntax ("green ideas sleep furiously" is syntactically correct but

semantically vague). An incorrect application of either will lead to errors or bugs; the semantic bugs are the most difficult to find.

Once again, sometimes it is tempting to take a shortcut and not spend the time defining the problem and jump straight to coding. At first this saves lots of time, but in many cases this actually takes more time and effort later if the program needs to be redesigned due to mistakes. Developing a general solution before writing the program helps you manage the problem, keep your thoughts straight, and avoid mistakes that can take much longer to debug and maintain than to code.

Documentation is another key part of the programming process that is often ignored. Documentation includes written explanations of the problem being solved, the organization of the solution, comments within the program itself, and user manuals that describe how to use the program.

Brief History of Programming Languages

1958: Algol defined, the first high-level structured language with a systematic syntax. Lacked data types. FORTRAN was one of the reasons Algol was invented, as IBM owned FORTRAN and the international committee wanted a new universal language.

1965: Multics – Multiplexed Information and Computing Service. Honeywell mainframe timesharing OS. Precursor to Unix.

1969: Unix – OS for DEC PDP-7, Written in BCPL (Basic Combined Programming Language) and B by Ken Thompson at Bell Labs, with lots of assembly language. You can think of B as being similar to C, but without types (which we will discuss later).

1970: Pascal designated as a successor to Algol, defined by Niklaus Wirth at ETH in Zurich. Very formal, structured, well-defined language.

1970's: Ada programming language developed by Dept. of Defense. Based initially on Pascal. Powerful, but complicated programming language.

1972: Dennis Ritchie at Bell Labs creates C, successor to B, Unix ported to C. "Modern C" was complete by 1973.

1978: Kernighan & Ritchie publish "Programming in C", growth and popularity mirror the growth of Unix systems.

1979: Bjarne Stroustrup at Bell Labs begins work on C++. Note that the name "D" was avoided! C++ was selected as somewhat of a humorous name, since "++" is an operator in the C programming language to increment a value by one. Therefore this name suggests an enhanced or incremented version of C. C++ contains added features for object-oriented programming and data abstraction.

1983: Various versions of C emerge, and ANSI C work begins.

1989: ANSI and Standard C library. Use of Pascal declining.

1998: ANSI and Standard C++ adopted.

1995: Java, designed at Sun Microsystems, goes public, which some people regard as the successor to C++. Java is actually simpler than C++ in many ways, and cleaned up many of the ugly aspects of C++.

Note that this is not a history of all programming languages, only C++ to Java! There are many other languages, procedural and non-procedural, that have followed different paths.

What is a Programming Language – Assemblers, Compilers, Interpreters

Internally, all data is stored in binary digits (bits) as 1's and 0's. This goes for both instructions and data the instructions will operate on. This makes it possible for the computer to process its own instructions as data.

Main memory can typically be treated like a large number of adjacent bytes (one byte is 8 bits). Each byte is addressable and stores data such as numbers, strings of letters, ASCII codes, or machine instructions. When a value needs to be stored that is more than one byte, the computer uses a number of adjacent bytes instead. These are considered to be a single, larger memory location. The boundaries between these locations are not fixed by the hardware but are kept track by the program:

Memory Address	Contents	
…	…	
Byte 4000	11101110	(2 byte memory location at 4000)
Byte 4001	11010110	
Byte 4002	10101011	(1 byte value, e.g. ASCII char)
Byte 4003	11010000	(3 byte memory location at 4003)
Byte 4004	00000000	e.g. string
Byte 4005	11011111	
…	…	

There is only a finite amount of memory available to programs, and someone must manage what data is being stored at what memory address, and what memory addresses are free for use. For example, if a program temporarily needs 1000 bytes to process some data, then we need to know what memory addresses we can use to store this data. One of the nice things about Java is that much of this memory management will be done for us by the Java runtime environment.

Computer instructions can be programmed directly as machine code. When computers were first developed, the machine code was the only way to write programs.

Ex: 110110 might be the instruction to add two numbers
110100 might be the instruction to increment a number
etc.

As you have likely surmised from the previous exercises with machine code, it is very tedious to write programs in direct machine code. One step above machine code is assembly code. Assembly replaces the machine codes with more English-like codes that are easier to remember. These codes are called **mnemonics**.

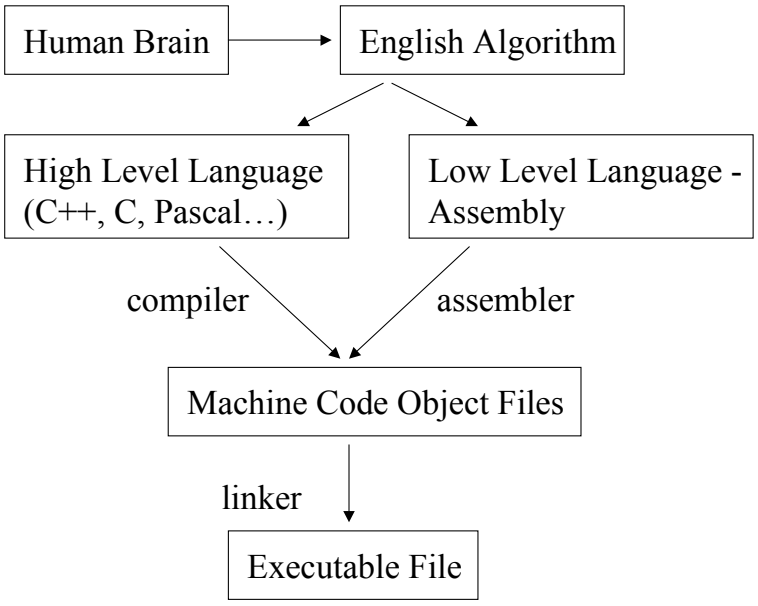
Ex:

ADD	110110
INC	110100
Etc.	

Although the assembly codes are easier for humans to work with, the computer cannot directly execute the instructions. But people write programs to translate the instructions into machine code. These programs are called assemblers.

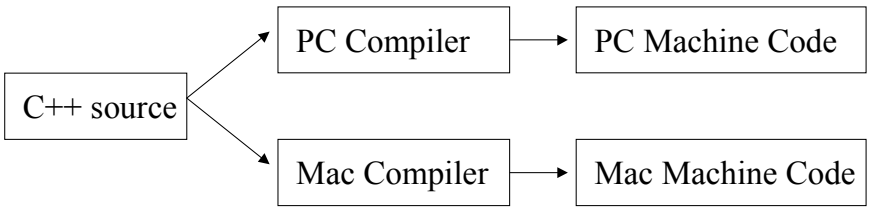
Assembly is still a lot of work for programmers to use because one must know exactly what machine-level instructions are available. Today most programmers use **high-level programming languages** that are easier to use than assembly due to increased generality and a closer correspondence to English and formal languages.

A program called a **compiler** translates programs in high level languages into machine language that can be executed by the computer. C++, C, Pascal, Ada, etc. are all examples of high level languages. (So is Java, but we'll get to that in a minute!)



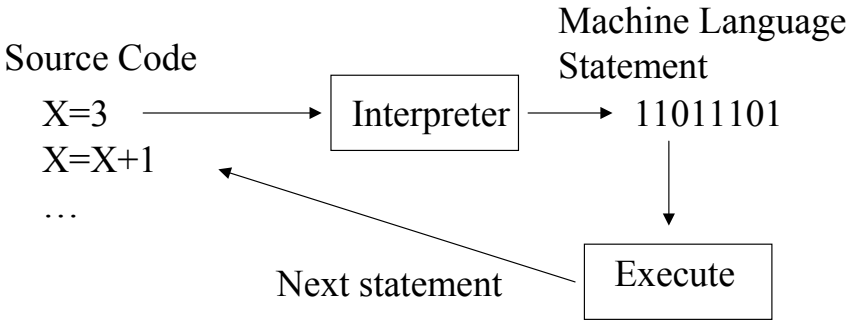
A program written in a high level language is called a **source** program. The compiler takes the source program and typically produces an **object** program – the compiled or machine code version of the source program. If there are multiple source files that make up a final program, these source programs must then be **linked** to produce a final executable.

Note that compilers on different machine architectures must produce different machine code. A macintosh cannot understand machine code intended for an Intel processor. However, if there is a standard version of the high level language, then one could write a program and have it compile on the two different architectures. This is the case for "standard" programs, but any programs that take part of a machine's unique architecture or OS features will typically not compile on another system.



Note that under this model, **compilation** and **execution** are two different processes. During compilation, the compiler program runs and translates source code into machine code and finally into an executable program. The compiler then exits. During execution, the compiled program is loaded from disk into primary memory and then executed.

C++ falls under the compilation/execution model. However, note that some programming languages fall under the model of **interpretation**. In this mode, compilation and execution are combined into the same step, interpretation. The interpreter reads a single chunk of the source code (usually one statement), compiles the one statement, executes it, then goes back to the source code and fetches the next statement.



Examples of some interpreted programming languages include JavaScript, VB Script, some forms of BASIC (not Visual Basic), Lisp, and Prolog.

Question: What happens if you modify the source on a compiled programming language (without recompiling) vs. an interpreted programming language and execute it?

Answer: the interpreted program will run with the changes, but a compiled program requires that the program be recompiled before new changes take effect.

What are the pro's and con's of interpreted vs. compiled?

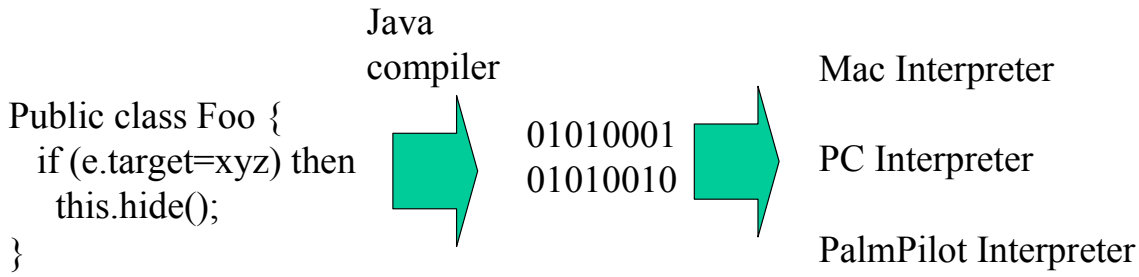
Compiled:

- Runs faster
- Typically has more capabilities
  - Optimize
  - More instructions available
- Best choice for complex, large programs that need to be fast

Interpreted:

- Slower, often easier to develop
- Allows runtime flexibility (e.g. detect fatal errors, portability)
- Some are designed for the web

In the midst of compiled vs. interpreted programming languages is Java. Java is unique in that it is both a compiled and an interpreted language. A Java compiler translates source code into machine independent **byte code** that can be executed by the Java **virtual machine**. This machine doesn't actually exist – it is simply a specification of how a machine would operate if it did exist in terms of what machine code it understands. However, the byte code is fairly generic to most computers, making it fairly easy to translate this byte code to actual native machine code. This translation is done by an interpreters that must be written on different architectures that can understand the virtual machine.



The great benefit of Java is that if someone (e.g. Sun) can write interpreters of java byte code for different platforms, then code can be compiled once and then run on any other type of machine. Unfortunately there is still a bit of variability among Java interpreters, so some programs will operate a bit differently on different platforms. However, the goal is to have a single uniform byte code that can run on any arbitrary type of machine architecture.

Another benefit is we can also control "runaway" code that does things like execute illegal instructions, and better manage memory. These topics will be discussed more later.

System Software

The use of assemblers and compilers makes programming easier. However, we still need more to really make our computer usable. For example, what starts a program running in the first place? How do we load a program from disk into memory? What if we want to run lots of different programs? All of these tasks are taken care of by the **system software** also known as the **operating system (OS)**.

The purpose of the operating systems is to be the master controller for all of the activities that take place within the computer. It also provides the main interface that you use to interact with application software and the file system. A few common operating systems include:

- Windows
  - Windows 2000, NT, XP
  - Windows ME, 98, 95
- Mac OS
- DOS
- Unix
  - Linux
  - Ultrix
  - SunOS
  - Solaris
  - HPUX
  - Etc.
- BeOS

The OS provides access to the computer hardware, and then application software runs on top of the OS software. As such, the OS provides a nice, high-level interface to the basic functions on the machine to make writing application programs easier. Since programs are typically written in conjunction with the services provided by the operating system, sometimes the machine plus the operating system is called a **virtual machine**.

The OS provides *external and internal services*. External services are those that are visible to users, like starting programs, accessing files, displaying icons, displaying a graphical user interface, etc. You should be familiar with the external services from your own experience using an operating system like Windows.

Internal services are those that are mostly invisible, like controlling processor time, memory access, disk access, multitasking, and utilities.

Processor Time refers to what program gets to execute next (a *process* refers to a program that is running in memory). Most computers today support *multitasking* which is the process of allowing each program to run for a very short *time slice* before switching to another program. By switching quickly among all available programs, it looks like everything is running simultaneously. This is what allows you to be reading your email at the same time that your computer is downloading a file and also printing out a document. In reality, the computer is switching back and forth between the different tasks very quickly, giving you the illusion that it is running all tasks simultaneously.

One of the major issues that arise with multitasking computers is the issue of *deadlock*. What if program 1 and 2 both want exclusive access to two resources? Let's say both programs want to use the printer and the scanner. Program 1 first grabs access to the printer. This means that no other program gets to use the printer until program 1 is finished. Then program 2 then grabs the scanner before program 1 gets it. Now, both programs wait until the other resource is free – program 1 is waiting for the scanner to be free, and program 2 is waiting for the printer to be free. However in this case they will never be free, because each program is waiting for the other to release it. Techniques to avoid deadlock are discussed in the operating systems class.

Memory access refers to the difficult process of managing what goes into memory and where. There is only a finite amount of memory in the computer. Programs must be loaded somewhere into memory. Each program also will want to use memory to store data. The operating system's memory manager determines where programs and data can be stored so there are no conflicts. Most OS's today support a feature called *virtual memory* which uses the hard drive to simulate a large amount of RAM in the event that a program needs to use more memory than is physically available.

Finally, utilities are collections of routines that provide useful services to users or other parts of the operating system. For example, a text editor such as notepad or packages to perform common graphics routines are utilities.

Another type of OS is the network operating system, such as Novell's Netware. This type of operating system is becoming more popular today. In a network OS, a network *server* holds the applications, data, and performs many crucial functions. Computers that want to access these applications are called *clients*. For example, to run a word processor, instead of storing it locally on a client, the client instead must download it from the server. This allows for centrality and easy updates (imagine if you had to upgrade word processors – there is only one place to do it in the network OS, in the server, instead of in all the clients). However, the server may become a bottleneck if it becomes overloaded with requests. We'll talk more about networking later in the class (chapter 12).