Overview of Programming

(Last Updated: 8/23/2018)

What is a computer?

A computer is a general purpose machine. Most machines are designed to do a single task, such as a toaster or a car jack. But a computer can do many different tasks. Initially computers required hardware changes to make them do different tasks. The ***stored program*** computers that we now use do not; sequences of instructions (or "programs") stored in the computer's memory make it do various tasks. The instructions control the computer ***hardware***, and are known as ***software*** by way of contrast. (There is also ***firmware***, but we can skip that for now.)

Computers contain hardware that can read data from devices such as storage, keyboards, mice, temperature sensors, etc.; can send data to storage, networks, and displays; and examine, compare, and manipulate data (e.g., add two integers, change lowercase letters to uppercase, ...). Other hardware can be controlled by computers too, if they are connected: robot arms, speakers, printers, telephones, cameras, etc. Such external devices are called ***peripheral*** devices.

To understand the purpose and use of programs you need to understand the basic hardware of a general purpose ("PC") computer a little:

- **CPU** The conductor or coordinator of a computer; this is the part that examines instructions and sends electrical signals to the other components, to carry out the instructions,
- **ALU** Does the math and other data manipulations and comparisons (**Multi-core/SMP** computers have multiple CPUs and ALUs),
- **I/O** (communicates with peripheral devices),
- **Memory** (ROM, RAM, cache, and other types),
- **Hardware Clock** (and battery, so you don't need to set the time at power on),
- **Bus** (allows all the components to send signals to each other; actually a computer may have many buses for different purposes, including PCI, PCI-X, ISA (old), AGP (for graphics), SCSI, USB, etc.),
- **Storage** (disk or optical).

Note not all "smarts" are in the CPU; some tasks are complex and can best be handled by special hardware devoted to that task, such as graphics, audio, network, and disk access.

> A CPU/ALU, bus, and memory system are all designed to work on data of a certain number of bits (*binary digits*). This number is known as the ***word size***. Early computers worked on 8 bits (a.k.a. a ***byte*** or an ***octet***) at a time. Some very early computers worked on smaller word sizes! Today most PC computers and servers have a 32 bit word size, although 64-bit is becoming more common.

Why do computers use binary numbers?

It is a matter of cost and reliability. Some early computers weren't digital at all, they used *analog*. A number was represented by a voltage. These early computers were not accurate at all, affect by temperature, humidity, etc. Digital computers use a range of voltages to represent a digit, so if the voltage changes a little, it won't affect the value stored. Early digital computers were indeed base-10, using 10 different voltage ranges to represent the numbers zero through ten. But they were very expensive!

It turns out that using only two digits gives very high reliability for very little cost. So any voltage below a certain level means the digit zero and any voltage above that means one. **Because only two digits are used, the system is called *binary*.** Any value can be represented using either normal base-10

digits or in binary. For example the base-10 number "012" is the same as the binary number "01010".
A binary digit is called a **bit**.

Because it is tedious for humans to read or write binary numbers other notations are often used; the computer can translate these notations into binary for us. For example:

```
10.2.135.120   = 00001010000000101000011101111000
"hello"   = 0110010101101000011011000110110000001010
66CC99    = 011001101100110010011001
```

(The last example is hexadecimal (or *hex*), representing the color green in HTML.)

What is an Operating System ("OS")?

Early computers only supported **batch processing**, where a single program could be run at a time. That one program had total control of all hardware.

An **operating system** changes that. When a computer is powered on a single program, the OS, is loaded into memory and run. The OS then has total control of all hardware. The OS in turn loads and runs programs. Early OSes ran one program at a time, but were still more convenient than batch processing. Later, OSes developed the ability to **multi-process**; that is, run multiple programs simultaneously.

> Although with a single CPU, the OS can only run a single program at a time. It fakes the appearance of multi-processing by switching rapidly between programs, running each for a tiny fraction of a section before switching to the next. Since CPUs can process billions of instructions per section, a human rarely notices.

In the olden days, every application needed to know about types of hardware. Each different make and model disk (or in today's terms, USB disk) would respond to different signals, and each and every bit of software had to be changed for each type. In addition programs had no protection from one another; one program can read or write any memory location or change any file on the system. So an error in any one program would crash the whole system!

Any attempt to access hardware by a running program is blocked by the OS. Instead, programs need to send a request to the OS to read or write data. The OS thus provides security and protection.

> **Software doesn't access hardware directly, but only through the OS.** Thus only the OS needs updating if the hardware changes. (The software that gets installed in an OS to manage specific hardware is known as a **device driver**.)

Note different OSes have different APIs (a.k.a. platforms, e.g., "Windows x64" or "Win32" or "Linux x86"), so when buying software must buy for the correct platform. This is why programs written for Widows won't run on Macintosh or Linux: the Windows software sends messages to the OS that a non-Windows OS won't understand.

> A **platform** is the combination of an operating system, CPU type, and word size. However Intel CPUs have dominated the market for years and most others emulate them today. This is called an "x86", (or "Pentium"), named after the Intel CPU integrated circuit chip model number (the first was "8086"). 64 bit Intel compatible computers are called "x64".
>
> You can generally (but not always) run a 32 bit program on a 64 bit computer, but not vice-versa.

Virtual memory (paging) is a feature found on all modern computers. Most textbooks only show "memory", but in reality memory used by a program is divided into "pages" (typically 4 kilobytes each). If some program accesses data or instructions found on a different page, the operating system must pause the program in order to fetch (copy) the correct page from disk. This *paging* can dominate program performance and invalidates traditional assumptions about best program design.

What is a programming language, and what are the differences between them?

A *program* is a sequence of instructions for a computer system to carry out. Before stored-program computers, you changed the program using a soldering iron. (We'll skip that.) Early stored program computers required you to enter in the program in binary using switches on the computer. (Show PDP picture: pdp-1120.jpg). Later paper tape and punched cards were used. (Let's skip these too!)

Next *terminals* (a keyboard and CRT screen) were invented, followed by PCs. In all cases today, you create a program by typing in the instructions on a keyboard and saving them in a file.

> ***Programming*** is the job of determining the precise sequence of instructions to make a computer perform some task, and typing in these instructions and saving them in a file. Entering in the instructions is the easy part; figuring out what instructions to use and in what order is much harder!
>
> Because of this, entry level software jobs are mostly handling a help desk, testing and finding errors, and implementing a program from a design determined by a more senior programmer. It usually takes three to five years of training before a new programmer is trusted to design software on their own.

A CPU only can understand *machine language*, the binary numbers that the CPU interprets as instructions to fetch some data from memory into a register, add the numbers in two registers and put the result in a third, store data in memory, or tell a disk to copy some data into memory. And so on. All memory locations were known by their address (a number). Here's a sample of machine code:

```
c7 45 f0 0d 00 00 00
c7 45 f4 02 00 00 00
8b 45 f4
03 45 f0
89 45 f8
```

An early improvement to machine language was ***assembly language***, where the CPU instructions were given easy to use names such as LOAD, ADD, or STORE, and the memory locations could be given easy to use names such as COUNT or TOTAL. The assembly language is easier to read and write for humans, but a program called an ***assembler*** must be used to translate the program into machine language. The translation is stored in a file that can be executed (or run) by the OS. Such a file is often called a ***binary*** file. **The file containing the human readable instructions (in this case, the assembly language instructions) is called a** *source* **file (and the program's human-readable instructions are called** *source code* **or simply** *code***).**

Here's a sample of assembly:

```
DECLARE cost 0010a60e
DECLARE tax 0010a612
DECLARE total 0010a616

STORE cost, 10
STORE tax, 1
LOAD cost, R1
ADD tax, R1
STORE R1, total
```

In a ***high level language*** you get to ignore details of the hardware you don't care about, and just concentrate on the task. Unlike assembly language each statement in a high-level language is translated into many machine language instructions. The program that does this translation is called a ***compiler***. Here's a sample:

```
int cost = 13, tax = 2, total;
total = cost + tax;
```

(The C language program above is equivalent to the assembly and machine code samples show previously; I have simplified the assembly a little.)

One advantage to using high-level languages is that the source code is portable between different platforms, as long as you have a compiler handy that will translate your program into the correct type of machine language.

Early programming languages viewed software as a mathematical expression to be solved. (This is where the term *function* comes from.) While a human is smarter about solving formulas the computer is so much faster that even if it computes in a "dumb" way, it will be faster than a human.

As early as the 1950s computers were doing other, non-mathematical tasks. These languages viewed software as a sequence of statements that told the CPU what to do.

What is a scripting language? What is a compiled language?

In some cases the source code is not translated at once and the result saved in a file. Instead the source code is read and translated one statement at a time, each time you want to run the program. The translator that does this is called an *interpreter* not a compiler. A program in such a language is often called a script, and a programming language designed to be interpreted is called an interpreted or scripting language. Common examples of interpreted languages include DOS/Shell scripting, and PHP, JavaScript, and other "web programming" languages.

An interpreted program takes longer to run than a compiled one, but is faster to develop since any errors are displayed as you enter the program, and compiling can take many seconds, to minutes or hours for large programs. Also if you use an interpreter you must give the customer the source code, which can be valuable to your company.

> Proficiency in scripting languages is a plus for most programmers, and a must-have for some programming positions. Less complicated than programming languages designed to be compiled, scripting languages are generally easy to learn and use. Programmers should be able to automate routine tasks using scripting languages, including performing arbitrary tasks to manipulate files, parse text input, and translate data among multiple forms and formats.

Computers are fast enough today to use both methods. As you enter in a program it is interpreted to detect any errors, and when done the program gets compiled so it can run quickly.

What is structured programming (and structured design)?

As computer hardware became more capable (and affordable) people wanted it to do more. By the 1960s the programs became so long (millions of statements each) no mere human could understand them. Naturally software had many errors, or "bugs", and finding them was difficult, costly, and unpleasant. Developers were afraid to try to add new features to software for fear of breaking it!

> The cost of software is 20% for the design and initial development, and 80% for the maintenance over its lifetime. (Software has a lifetime because after a certain point it is not cost-effective to add new features.)

Software developers embraced a new style of software (which required newer computer languages) known as *structured programming*. One idea of structured programming was to design programs in a "top-down" manner: first you design a "big picture" view of the program, a series of **modules** that would do a sub-task each, and could be combined into one large program. This initial design was lacking in all details. Later programmers work on the individual modules. Since each module was mostly independent of the others work could be done in parallel by teams of programmers. Each module could be tested and "debugged" independently too.

What is object-oriented programming (and object-oriented design)?

Structured programs have modules focused on actions (verbs). While this is fine for some problems, most software designs are hard to express as a series of actions or tasks. Many software projects failed simply because the best design methods weren't up to the task; the projects would take longer (years) and cost more than it would be worth once it was delivered. (And even then the costs of maintenance would be very high.)

In the 1980s and 1990s a new style of programming and design was developed, based on the idea of making the modules focus on things (nouns) not actions (verbs). The modules are called *objects*. Most objects represent real-world things, events, and concepts. There are date objects, time objects, paycheck objects, calendar event objects, invoice objects, student and instructor objects, fire alarm objects, etc. Some objects don't represent anything but themselves: button and other GUI objects, audio and video clip objects, data record objects, etc.

Objects have *properties* (also called attributes, fields, instance variables, and other names). Each property has a name and a value. For example a *student* object might have properties with names such as "Name", "ID number", "Home address", "Date enrolled", "Transcript", etc. (Notice how some properties can also be objects!) A button object on a GUI display might have properties of height, width, position, color, caption, font, etc.

Objects also have *actions*, or behaviors. A fire alarm object might have actions of "turn on", "turn off".

Many OOP languages use similar syntax: *object.method*() to invoked *method* on some *object*, and *object.property* to get or set the value of *object's property*.

A typical program may have many objects (dozens to thousands). Many objects are similar; how many "menu" objects do you see on a web browser? All similar objects have the same set of properties and actions; they just have different values for them. All similar objects are of the same *class*. (E.g., class button, class Student, class Bicycle, ...)

In the object oriented design method (OOD) the software developer decides what objects are needed, with what properties and actions. In object oriented programming (OOP) you write programs containing object descriptions, or classes.

One benefit of OOP is that you can often reuse existing classes. For example there is no need to create a class for GUI menus. When you do need to create your own class, you need to define the properties and actions. The individual actions are designed using structured programming (which works very well for actions).

The classes whose objects represent real-world things are called *abstractions*. Real world objects have a nearly infinite number of properties. But for a given program only some of them are important. Do "Person" objects need to represent a person's height, weight, or shoe size? Not for an HCC student registration system but maybe for a doctor's patient system. This abstraction process is one reason why you can't find a Person class pre-built you can use; a general Person class suitable to many different applications would have too many properties and actions to be used easily.

Once all the classes are done, a small amount of initialization code creates all the objects from the classes. For example you might create three Button objects from the class Button, each at a different position on the screen, with a different caption, and each can do a different action when clicked on.)

Now the program simply waits for the user to click or type something, or for data to arrive (say stock quotes from the Internet). These events trigger some object's action, which might invoke other objects' actions.

The benefits of OOD/OOP are huge and today all large software projects are designed and written this way. Some of the benefits include the reuse of classes, the ability to take an existing class (such as a class representing people) and specialize them easily (to represent a student). Classes are much more independent than task-based modules, so bugs tend to be localized to a single object and are more easily found and fixed. It is easier to reuse a class for other programs when they represent things. Updates and enhancements can be made to individual classes without affecting the rest of the software.

One problem for students is that trivial tasks are harder with OOP; you don't really see the benefit until you work on larger projects. But what teacher will assign their students a 100,000 line program for homework? You will just have to trust me! **Don't be tempted to design and program in a sloppy way (and make the program "pretty for teacher" at the last minute) just because it is easier for the short programs you will write for class. You need to practice programming the right way so you don't have to unlearn bad habits later.**

What other types of programming are there?

Actually, there are many. ***Declarative programming*** is when you write your code in such a way that it describes what you want to do, and not how you want to do it. It is left up to the compiler or runtime system to figure out the how. Examples of declarative programming languages are SQL, Prolog, HTML and CSS, and regular expressions. In contrast, Java, C#, JavaScript, and so on, where the programmer specifies a series of algorithmic steps to take, is known as ***imperative programming***.

Functional programming is popular in the telecommunications industry. It is a type of declarative programming in which you define functions, or expressions, that compute something from their arguments only (so the same arguments cause the same result every time.) A functional program just evaluates an expression, composed of sub-expressions. It may seem strange that useful software could be developed without variables or stored data, but many types of problems lend themselves to this style. Examples include Haskell, Closure (which is compatible with Java and also compiles to Java byte code), Erlang, and others. Note that support for some functional programming may be present in otherwise imperative languages, such as JavaScript and C#.

Generic programming is another style, supported by Ada in 1983, and later by C++, Java, C#, and other languages. Briefly, generic functions and *classes* are written with the types of variables determined later. The basic idea is to capture the procedure in code, and have the system be able to apply the code to different types of data.

***GPU* (graphic processing unit)** *programming* means to craft software designed to run on the type of CPU used in graphics adaptor. Such GPUs need to be able to handle hundreds (to thousands) of individual threads of instruction in order to render smooth graphics. It turns out that many other problems can benefit from GPU's design, such as crypto analysis. Such software is typically written in a subset of C or C++, along with the OpenCL standard library.

There are many other types of programming, designed for specific applications and/or hardware. These include scientific computing, smart phone apps, *declarative programming* (for example, in HTML), and database programming (in SQL).

What does event-driven mean?

Most applications and server programs today are *event driven*. The basic design is the same for all such programs: Create objects, display the UI, and do the following forever: wait for something (an "event") to happen, trigger some object's action. No matter what the language used, event driven programming is often the best design. Examples include games, servers, and network software.

What does client-server mean?

Before the Internet, software was confined to a single computer. Such programs are called *stand-alone*. Once networking became common it was possible to split a program into parts: the ***client*** that has a user interface and can do some computations, and the ***server*** that responds to requests from (authorized) clients for data and services. A single server can handle requests from many clients. For example an airline reservation system uses a server to book reservations on flights; the client software is available at travel agencies. Stock brokers use client software to buy and sell stocks, using a server that tracks the number of shares available and their price. Salesmen use client software on laptops to place orders. And so on.

The amount of work done by the client and server can vary with the design. A ***thin client*** is just a user interface; all the processing is done on the server. A ***thick client*** has a user interface and does most or all of the processing; the server just provides information from a database. Thick clients allow many clients to run processor-intensive applications, since each client runs on a different computer. But think clients must be distributed ("deployed") to the customers, and updated regularly. Using a thin client keeps most of the code on the server, away from your competitors, safe from viruses, and (for web programs) no deployment is needed. For some applications a thin client can run just as fast as a thick one. To increase the capacity a company can use a ***cluster*** of application servers. (That's what Google, Amazon, E-Bay, etc. all do!)

Web browsers make possible really thin clients. The user interface is just a web page form. When the user clicks the submit button, the data on the web page is sent to a web server (via the HTTP or HTTPS protocol), which does one thing or another depending on the design. Each task handed by the server has a unique URL (for example, https://FlyByNightAir.com/bookflight or paypal.com/billcustomer). Such web applications ("web programming") works very well since no client code ever needs to be shipped to customers; all the code is on the company's web server (both the client web pages and the server part of the program).

What is web programming?

You can't program without an understanding of the Web. Web applications are everywhere. From simple "dumb" documents ("web pages") written with HTML, the World Wide Web has evolved into a complex, dynamic environment that uses many different technologies. The current crop of popular technologies is sometimes called "web 2.0". Web programming includes HTML, and other programming languages such as **JavaScript** (which is unrelated to Java), CSS, Flash, PHP and/or ASP. Many of these technologies are evolving quickly; currently new versions of HTML, JavaScript, and CSS are being developed (and web browsers already support some of these new features).

In addition to scripting up web pages a Web Developer needs to understand multimedia (how to deploy it) and database ("SQL") technology such as MySQL, MariaDB, or PostgreSQL, as well as general deployment issues. The HTML and database (and other servers) are often tied together with the PHP programming language. Today, complex websites use one or another framework to manage the collection of web pages (a *web site*): WordPress, Drupal, Plone, etc. (Some of those are written in PHP.) To avoid the headache of writing web applications, partly in PHP and partly in JavaScript, many web sites are tuning to *Node.js*, a JavaScript framework for the server that can be used instead of PHP.

Employers today want programmers who understand modern programming principles, including object-oriented design, distributed programming techniques, client/server software design, and who understand the process of producing software: design, implementation, testing, debugging, and maintenance. Along with all kinds of development methodologies, employers will expect some familiarity with related environments or toolsets (that they've spend money on already).

> Many, perhaps most programming job interviews will require the applicant to be able to code a simple program quickly, on a whiteboard (no computer or Internet to help).

What programming languages are used?

While it is true that hundreds of high-level languages have been developed over the years, only a few have seen wide acceptance. Most are *procedural* languages. Some early languages still in use today include:

- **COBOL** - Designed for business applications in the 1950s and adapted by financial institutions. While COBOL is an ugly language, there is such a large investment in it, and so much of it out there, that it doesn't make sense to scrape the software. The programs are hideous to update, which is why COBOL programmers get paid a lot.

- **FORTRAN** - Designed for scientific computing (number crunching), it is still used on super-computers.

- **BASIC** - Designed in the 1960s as an interpreted language to teach programming.

- **Pascal** - Designed in the 1970s by a professor to teach a more modern style of programming than BASIC supports ("structured programming"), it was so popular that nearly every college used it exclusively. This lead to an unfortunate "four-year effect": students graduated only knowing Pascal and tried to use it for real-world software. (The Mac OS 1.0 was written in Pascal.) The language is not suitable for that, but it inspired many other languages. It isn't used today.

- **Ada** - Designed by a committee, Ada was mandated by the U.S. DoD. Before Ada the military computer systems were a large collection of programs written in a hodge-podge of languages, making it difficult to hire programmers, or to have different systems communicate. Ada didn't catch on, as it was not easy to write useful programs.

- **C** - Developed in the late 1960's it remains one of the most popular languages to this day. Designed to support "structured programming" like Pascal but included the features needed to write compact, efficient code. Most system programming is done in C (that is, Unix, Mac OS, Windows, Oracle, and compilers such as Visual Basic .net).

- **C++** - Developed in the 1980s to add "object orientation" to C. Whatever system's programming isn't done in C is almost always done in C++ today, especially in the gaming industry. Embedded systems (such as the computer in an ATM machine, an airplane, an industrial robot, or the smart cash registers used at McDonald's) also use C or C++. (The name comes from an operator in the C language; "++" means to increment.) In addition to OOP, C++ supports *generic programming*, not discussed here.

- **Visual Basic** - Developed to create applications for Windows platforms. Before "VB", creating GUI applications was very difficult especially for Windows platforms. VB is not the same as BASIC; like most modern languages it was inspired by Pascal's structured programming, and later was modified to support OOP. VB also includes an *integrated development environment* (or "IDE") called ***Visual Studio*** that allows you to develop GUI programs using a *WYSIWYG* interface, allowing for very *rapid application development* (or RAD). The language includes a very large library of objects (sometimes VB programmers call these "controls"). Although applications can be created in other languages, and VB has a lot of historical "baggage", it remains very popular.

  > Visual Basic is also the language used to write macros for Word, Excel, and other applications.

- **Java** - Developed in the 1990s by Sun Microsystems (now Oracle) to develop a "better" C++, suitable for non-systems programming. Unlike all the above languages, Java included statements

for creating *graphic user interfaces* (GUI), for networking, and many other features not found in other languages. Java also supports a "write once, run anywhere" approach.

A Java source program is compiled into machine code for a non-existent platform! This is a "virtual machine", and the compiled program is called "byte code". Every platform that supports Java **must include a "Java Runtime Environment" or JRE**, which includes an interpreter for byte code to machine code (the Java Virtual Machine or JVM) and a large standard library of pre-written program fragments you can use.

Today Java dominates the market for server programs. And except for the iPhone from Apple, Java is the language used to develop "apps" for cell phones, blackberries, and PDAs. (The iPhone uses a variant of C for its apps.) Java "applets" can be embedded in web pages as "active media", although today Adobe's "flash" is more common.

- **Visual Basic .net** - VB.net is an updated version of Visual Basic that takes advantage of Microsoft's ".net" framework (discussed below). Like Java, all the .net languages compile into an intermediate language and require a .net runtime.

- **Haskell, Scala, Closure, and Erlang** - Popular functional programming languages.

- **C# .net** - Microsoft's answer to Java. It is a recent language and takes advantage of the lessons learned with C++ and Java to create a "better" language for programmers than Visual Basic, at the same time retaining the rapid development possible with Visual Studio IDE. C# also uses the ".net" runtime (discussed below).

  > A Java (or .net) program need not be interpreted. It can be compiled to machine code for some platform. Of course if you do that, the program won't run on other platforms (you'd need to compile it for each platform you want to support), but then doesn't need the JRE (or .net) framework installed.

- **SQL** - The standard language of relational database programming. SQL powers most e-commerce sites and much of the financial industry.

- **HTML** - This is not a programming language like the others listed here. It has no imperative statements: you can't write HTML code that does something. There are no variables, data, math, loops, etc. in HTML. One problem with creating web pages manually is that while they might look fine on a big monitor, they might look awful on a tiny cell phone screen, or may not be accessible to vision-impaired users.

- **JavaScript** - Used for creating software ("web applications") that are designed to run in a web browser. The language has many good features, as well as some best avoided in quality programs. Many have tried to replace JavaScript with a "better" language, such as Google's Dart, but so far, none have had the widespread support of JavaScript. Today, JavaScript is also used for server-side code, usually using Node.js.

- **PHP** - This popular language is typically used to generate dynamic HTML on a server. (An alternative is Microsoft's ASP, but that is rarely used on non-Microsoft web servers.)

What is ".net"?

The .net framework is not just for VB. It is similar in nature to Java's JRE. .net includes a virtual machine and class library called the ***common language runtime*** (or **CLR**). The virtual machine includes an interpreter for ***Microsoft Intermediate Language*** (or **MSIL**, analogous to Java's byte-code). All .net languages including VB and C# compile the source code into MSIL. The CLR also includes the .net *framework class library* (or FCL).

In an unusual move for Microsoft they have published the specifications for the CLR and MSIL (the common language specification, or CLS). This has enabled .net frameworks to be written for non-

windows platforms! Already there is an open source .net framework available for Linux called "mono".

The .net framework includes support for web services (web programming), not just for web browsers running on a traditional PC, but also web programming for cell phones, PDA's, netbooks, and other devices. The VB.net program can create the proper HTML for you, no matter what the client. (Note Java supports all that too.)

What is the job of a programmer?

Programmers write the code that tells computers what to do. Not surprisingly, computer programmers spend much of their workday in front of a computer in an office setting. Technological advances also allow programmers to telecommute from home or remote locations. Important responsibilities of a computer programmer include: Writing, testing, designing and maintaining the programs that allow computers to function properly; Updating, repairing and modifying existing computer programs to make them operate as efficiently as possible; and Converting software designs into conventional programming languages such as COBOL, Java, C++ C#, PHP/HTML/JavaScript, etc.

Computer programmers should be comfortable both working as a member of a team and individually. Entry-level programmers may get their start in the industry by working alone on simple projects, or working together with a team of more experienced programmers.

There are different types of programming. *System code* tells a computer how to interact with its hardware; *applications code* tells a computer how to accomplish a specific task, such as word processing or spreadsheet calculating. A computer programmer writes programs usually relying on specifications obtained by someone else.

Systems programmers must be familiar with hardware specifications, design, memory management, and structure, while ***applications programmers*** must know standard user interface protocols, data structure, program architecture, and response speed. Most programmers specialize in one of the two areas. Of course there are many areas of specialization available, including framework programming (e.g., .NET, JRE, Eclipse, Mozilla, etc.; someone has to write the pieces of these, and others have to write code that uses these), network programming, game programming, database programming, enterprise programming, and web programming.

Nowadays ERP (*Enterprise Resource Planning*), CRM (*Customer Relationship Management*), HRM (*Human Resource Management*) and other systems are emerging. These systems are packages that are ready made and implemented in almost all organizations that can afford it. Sometimes programmers must learn these packages and how to deploy and customize them, and create add-ons (or extensions or plug-ins) for these systems. (Most of the time those jobs are assigned to people of the appropriate field e.g. accountant, business admin graduate.)

Currently the top five most demanded programming languages are Java, C, C++, PHP, and Visual Basic. Since the popularity of programming languages is constantly changing, **employers look for programmers who have not only subject matter expertise in at least one of these major programming languages, but also a broad knowledge and understanding of multiple other languages** such as Python, JavaScript, Ruby on Rails, C#, and so forth.

A simple Google search can put you in the know as to what programming languages are currently "hot". **Programmers are expected to understand all of the nuances related to the programming language(s) they know.** This means understanding how to architect, design, implement, test, debug, and maintain both simple and complex systems.

A *Software Engineer* is an experienced programmer who analyzes requirements and designs software systems. These designs are then implemented by programmers.

It is rare for a programmer to have total responsibility for some system. Instead programmers work in teams on new projects, or are assigned specific parts of a system to code, test, or improve. Programmers usually hold *code reviews* (and *design reviews*) in groups, to help each other (nobody can proof-read their own code as well as another).

At the start of projects, applications programmers meet with the designers, artists, and financiers in order to understand the expected scope and capabilities of the intended final product. Next, they map out a strategy for the program, finding the most potentially difficult features and working out ways to avoid troublesome areas. Programmers present different methods to the producer of the project (the software engineer who developed the design, the program manager, their peers, or all of these), who chooses one direction. Then the programmer writes the code.

Systems programmers may be hired on a Monday, handed the technical specifications to a piece of hardware, then told to write an interface, or a patch, or some small, discrete project that takes only a few hours. Then on Tuesday, they might be moved to a different project, working on code inherited from previous projects. Systems programmers must prove themselves technically fluent: "If you can't code, get off the keyboard and make room for someone who can," wrote one.

Application programmers may be assigned work by their manager or by meeting as a group with other programmers. Often a *trouble-ticketing* system is used and a programmer is assigned some issue to resolve, some error in the software or some feature request.

The final stages of a software project are usually marked by intense (and often isolated) coding and extensive error-checking and testing for quality control. The programmer is expected to address all issues that arise during this testing. The resulting code often must then be presented in a *code review* to other programmers.

Both arenas accommodate a wide range of work styles, but communication skills, technical expertise and the ability to work with others are important in general. Programmers work together respectfully; they help each other when they want to. But there are no significant professional organizations which might turn this group of people into a community. The best features of this profession are the creative outlet it provides for curious and technical minds, the pay, which can skyrocket if a product you coded is a major success, and the continuing education. A few programmers we surveyed indicated that an aesthetic sensibility emerges at the highest levels of the profession, saying that "Reading good code is like reading a well-written book. You're left with wonder and admiration for the person who wrote it."

What else is required?

Programmers need to understand the principles and practices common to production software development. Production considerations include such things as version control, code libraries, source control systems, documentation, code reviews, testing tools and methodology, and software release management. Whether you're a one-man shop writing custom applications, or a member of a 100-developer team, understanding these principles will make your life as a programmer much easier.

Many companies now use specific methodologies for managing the software development lifecycle, and programmers should be familiar with those methodologies and how they relate to the production cycle. Some common methodologies include Agile, Lean, Scrum, Spiral, and Waterfall. Certifications and workshops are available for many of these methodologies.

Personal Skills

Success doesn't lie in technical skills or education alone, although those certainly help. Programmers don't exist in a vacuum; they generally interact with teams. Successful programmers must also possess the following personal (soft) skills:

- **Ability to solve problems.** A savvy programmer must be a solution-oriented, out-of-the-box thinker who frequently comes up with new and innovative ways of doing old (and new) things.
- **Good communication skills.** This means communicating with humans. These skills should encompass all facets of communication, including both written and verbal contact.
- **Self-discipline.** While programmers may interact with groups, writing code is a solitary process that requires self-discipline. The world of software development is frequently fraught with deadlines and last-minute changes that translate into long days and lots of hard work.

Academic Requirements

Academic requirements are important for entry-level positions in the field of programming. Coursework should include basic and advanced programming, some technical computer science courses, and some logic or systems architecture classes. The complexity of what first-time programmers are asked to code is growing, as is the variety of applications. Long hours and a variety of programming languages including PERL, FORTRAN, COBOL, C, C++, and C# can make the student's life a whirlwind of numbers, terms and variables, so those who are not comfortable working in many modes at once may find it difficult to complete tasks. The programmer must remain detail-oriented in this maelstrom of acronyms. For mobility within the field, programmers should concentrate on developing a portfolio of working programs that show competence, style, and ability.

Computer Science covers the core concepts and technologies involved with how to make a computer do something. Learning to program a computer by writing software is essential, and computer programming is used in most computer science courses. You will learn details about how computers and networks work, but with an emphasis on how software and programming languages work. You will learn how to make them do very sophisticated things (e.g. graphics, robotics, databases, and operating systems). You will also learn about the theory behind how and why computers and software work.

Software Engineering focuses on how to design and build software in teams. You will take many of the same courses as you would in computer science, but you will take additional courses that teach you about topics like requirements engineering, software architecture, software testing, and software deployment. You will learn about working with people (communication, management, and working with non-technical customers), processes for developing software, and how to measure and analyze the software product and the software process.

Other educational programs include computer engineering, MIS, and other such titles.

Careers

A number of programmers take on additional duties to become systems architects, software producers, or technical writers. Some specialize in systems programming, web programming, enterprise applications, games, or embedded systems. Others take their programming expertise to a related profession, such as graphic designer or animator. Those who go into government work can become computer security consultants, encryption specialists, or federal agents specializing in computer science. A few who enter the business world become Management Information Systems Specialists (MISSs) who analyze, improve, and maintain corporate information systems for (usually) large, multinational corporations. The software engineers analyze new systems from clients, do feasibility studies, work on project management tasks such as costing and planning etc.

Programmers often enjoy the technical challenges of the work and prefer a career path that stays out of management. Others grow into software engineering, then project management. To succeed you will likely need a 4 year degree and some certifications. Then you can expect to earn between $45,000 and $70,000 annually with limited experience, and about $1,000 to $2,000 more for each year of experience. Additionally, the U.S. Department of Labor has forecast that information technology (IT) careers will be among those with the highest demand and growth for quite some time to come.

> **Resources** (*for the job and career information above, read on 7/2010*):
>
> [A Day in the life of a Computer Operator/Programmer](#)
>
> [What is a working day in the life of a programmer/software engineer like?](#)
>
> [Job descriptions: Day in the life of a computer programmer](#)
>
> [What is the difference between computer science and software engineering?](#)
>
> [Computer Programmer: Career Information](#)

Summary

- Professional societies and certification programs can help you find jobs by improving your qualifications, and help with careers (planning, skill updating, etc.). Some well-known professional societies include the ACM and the IEEE. Microsoft defines certifications such as MCPD.
- A computer is a machine that uses software (programs) to define the tasks it will do. Computers are composed of CPUs, ALUs, Memory, storage and other peripheral devices, connected via buses. Computers have a *word size*; today 32 bit computers are common.
- Binary numbers are used to provide high reliability for low cost, but are harder for humans to use directly.
- A single program called the *operating system* (OS) controls the hardware. All other programs are run under the control of the OS, and don't access hardware directly. An OS provides other features such as security and multi-processing.
- Programs must be written for a particular OS and CPU type (a *platform*).
- A programmer determines the exact sequence of instructions to make a computer perform some task. The set of instructions forms a *program*.
- Programs can be written in different languages. The CPU only understands *machine language*. Any programs written in other languages must be translated into machine language before they can be executed. Other languages include assembly and high-level languages.
- A program written in a high-level language is called source code. It is translated by a compiler or interpreter. A file containing machine code is called a binary.
- Structured programming breaks up large programs into smaller modules, each focused on some task.
- Object oriented programming breaks of large programs into smaller modules, each focused on some "thing". The modules are called objects.
- Objects have properties and actions. Similar objects belong to the same *class*.
- OO software design means to decide on the objects you will need for some program. OO programming means to write (or reuse) classes for the different types of objects needed.
- A (usually short) initialization action creates all the required objects from the classes, sets their properties and actions. Then the program just waits for events to occur, and runs the statements for the associated actions.

- Client server programming splits a program into two parts. The client-side program has the user interface and makes requests to the server-side program. With *thin* clients most of the processing is done server-side. With *thick* clients most of the processing is done client-side.
- With web programming each task performed on the server-side has its own URL. Clients use HTTP and JavaScript to pass data to and from the server and to request services, often using PHP.
- Some popular computer languages (out of the hundreds that have been developed) include COBOL, FORTRAN, C, Visual Basic ("VB"), C++, Java, VB.net, and C#. HTML is not really a traditional programming language, however JavaScript can be embedded in a web page.
- The .net framework include the CLR (common language runtime). The CLR contains the virtual machine interpreter for MSIL (Microsoft intermediate language) and the FCL (framework class library). .net is designed to support client-server programming, including web services for different types of devices (PCs, cell phones, etc.)
- The JRE includes the JVM (Java virtual machine) and the standard Java class library. Java is designed to support desktop applications, web page embedded "applets", client-server computing, and web services for "micro" devices such as cell phones.
- Professional programmers develop, improve, test, and debug software. They will at times work in teams and individually.
- Entry level programmers rarely design software; software engineers do this.
- Professional programmers will need continuous education, which often is provided from professional associations. In today's job market, a 4 year degree helps get the better jobs. With either a 2 year or 4 year degree, you will likely need some certifications to land entry level positions.
- Even in today's job market, programmers and software engineers can expect to earn over $70,000 per year, with some experience. The U.S. Dept. of Labor has forecast (2010) that I.T. career grow with be among the highest of any career, over the next decade or so.