Unit 1 Introduction to EECS

All About EECS

EECS is only 100 years old.

EECS is a discipline that integrates many other disciplines, such as physics, chemistry, mathematics, computers, electronics, optics, electromagnetics communications, analysis of signal and images, remote sensing.

Our School

The School of Electronics and Information Engineering was established in May, 2001 with the following specialties: Software Technology, Electronics and Information Engineering, Animation Design and Production, Computing Networks, Multimedia Technology, E-Commerce, and Applied Computing Technology.

The School focuses on technical education, all-round development education, creative education, and practical education, and aims at training specialized technical professionals to meet the needs of social development.

Faculty

For our current faculty list, please visit our web site http://cie.lnc.edu.cn/

Course Outlines

Specialty: E-Commerce
Training Objectives: To train students for mastery of e-commerce technologies, modern management skills, and networking techniques.

Core Subjects: Practical E-Commerce, Network Marketing, E-Payment and Security, Customer Relations Management, E-Commerce Project Management, Database Systems, E-Commerce Web Design and Management, Management Information Systems.

Specialty: Electronic Information Engineering

Training Objectives: To train students for mastery of modern electronic technology, electronic system design, and basic capabilities of development and applications of integrated electronic equipment.

Core Subjects: Electronic Technology and Project Training, Electric Techniques, EDA Technology, DSP Technology, Put-In Syste3ms, Intelligent Robot Design and

Production, Automobile Electronic Control Technology, Smart Card Technology.

Specialty: Animation Design and Production

Training Objectives: To train students for professional knowledge and skills of animation design and production, through practice work with local animation companies.

Core Subjects: Graphic Processing Technology, Animation Techniques, Drawing Techniques, Animation Production, 3D Animation, Promotional Design, Animation Marketing Analysis.

Specialty: Computing and Multimedia Technology

Training Objectives: To train students for professional knowledge and skills of multimedia design and production, through practice work with local multimedia production companies.

Core Subjects: Graphic Processing Technology, Graphic Design, Animation Techniques, Drawing Techniques, Animation Production, 3D Animation, Advertising Design, Video Editing Techniques.

Specialty: Computing Network Technology

Training Objectives: To train students for professional knowledge and skills in computing network technology and applications, meeting the development needs of the local computing network industry.

Core Subjects: Computing Network Operating Systems, Network Equipment and Enterprise Networks, Network Construction and Management, Network Security Technology, Website Planning and Construction.

Specialty: Applied Computing Technology

Training Objectives: To train students for professional knowledge and skills in computing applications and IT management, meeting the growing needs of IT applications for social and business development.

Core Subjects: Computer Fault Testing and Repair, English for IT Profession, Database Techniques, Website Design, Engineering Drawing, Network Equipment Maintenance, Linux System Maintenance, Data Restoration Techniques.

Specialty: Software Technology

Training Objectives: To train students for professional knowledge and skills in software development and applications, through practice work with local software companies.

Core Subjects: Program Logic, JSP Web Techniques, C# Sales and Marketing Systems, .Net Web Techniques, SQL Server Management, Software Project Management, Game Programming. Three Streams: Software Development; Software Engineering; Game Software Development.

Unit 2 How to Study Computer Science

“What school teaches computer science right?”, someone asked on programming.reddit. My university is one of the best here in Germany according to the CHE rating. We’re officially elite, yet, this doesn’t guarantee a good education. It is possible get a degree and have no clue about programming, math and science at the end.

The question was primarily about the programming languages used, i.e. do the lectures use languages other than Java/C++. The right thing, as implied by the author and the community, would be to teach cool languages like Ruby, Haskell, Erlang and Scheme.

Does the choice of programming language influence the quality of education? No. Learning Java and C++ is a good thing, since most of the software is written with those, but a teacher, who uses Java/C++ for everything, even when it is inappropriate, shows a lack of wisdom for me. Trying to explain Functional Programming with Java is funny at best.

Unfortunately it is unlikely that you learn any cool language in any school. As with most things, you have to do that on your own. A university can only give you resources and opportunities. Everybody knows that lectures and pen&paper tests are the worst way to educate.

What you do with your time is your responsibility. Here is my list (mostly scooped from w-g), what you should do or learn:

1. Programming paradigms – imperative, object-oriented and functional programming. Try to learn a language for every category. Example: C (procedural), Java (object-oriented) and Haskell (functional). If you know every style, you will program much better, especially when you use multi-paradigm languages, like Python, C++ or Scala. Addons are Assembler, Prolog and Forth.

2. Algorithms and data structures – Complexity theory, optimizations, searching, sorting, cryptography, AI, graphics. Dig deep to the hardware level and learn about instruction sets, caches and architectures. Compilers and operating systems are the high end topics here.

3. Abstract background knowledge – quite theoretical. Lots of math, formal

verification aka model checking, Turing machines, lambda calculus and type theory.

4. The human side – project management, design patterns, UI design, psychology. You will have to deal with people, who think very different than you. Managers, customers and co-wokers may not be “into computers” like you are.

5. Practice – with the Open Source movement, you have a wealth of projects, who would like to get a hand or two. You’ll learn to deal with people and foreign source code. You’ll also gain familiarity with some tools like version control, IDEs and wikis.

6. Creativity – don’t forget to do other things. Music, design, drama or whatever you like. Your brain has two sides and not both are for logic. Creativity and outside the box thinking can be learned and nurtured.

7. Personality feats – university is a good place to develop yourself. Learn to work with discipline, to stay healthy and fit and to be social! Train good habits and starve the bad ones! Read a lot!

The choice of the university is mostly important for the piece of paper you get at the end. The education you get depends on you.

Unit 3 Introdution to Programming

What is programming?
Given the general nature of that question, and the fact that programming is seen as a hideously complex subject, you're probably expecting a highly convoluted and technical answer. But you're not going to get one (sorry about that). In truth, it's quite easy to say what programming is, so I will:

- Programming is breaking a task down into small steps.

That's just about the most honest and accurate answer I can give. It also has the added benefit of being concise, and sounding very much like something you'd read in an official book on the topic, thus adding to my credibility.

You're perhaps wondering what exactly I mean by breaking a task down into small steps, so I'll explain the point in more detail. Let me start by giving you a fact about programmers that you'll find very easy to believe:

- Programmers think in an unnatural way.

This refers just to the fact that programming is breaking a task into small steps, and that's not the usual way that your mind works. An example will help you to understand what I mean. Here's a task for you to do:

Put these words in alphabetical order: apple, zebra, abacus

I'm going to assume that you managed to put them in alphabetical order, and ended up with abacus, then apple, then zebra. If you didn't manage that, reading this article may soon become one of the worst experiences of your life so far.

Now think about exactly how you performed that task; what steps you took to put the words in alphabetical order, and what you required to know in order to do so. The most obvious thing you needed to know was the alphabet; the desired order of the letters.

Then, if you're like me, you probably did something like this:
1. Look through the words for one beginning with "A".

2. If you found a word beginning with "A", put that word at the beginning of the list (in your mind).

3. Look for another word beginning with "A".

4. If there's another word beginning with "A", compare its second letter with the second letter of our first "A" word.

5. If the second letters are different, put the two words in alphabetical order by their second letter. If the second letters are the same, proceed to the third letter, and so on.

6. Repeat this whole process for "B" and each other letter in alphabetical order, until all the words have been moved to the appropriate place.

Your method may differ slightly, but probably not by much. The thing to notice (which I noticed particularly, since I'm having to type all this) is how much time it took to explain a process which happens without any real conscious thought. When you saw that you had to put some words in alphabetical order, you certainly didn't first sit down and draw up a plan of what you were going to do, detailing all the steps I listed above. Your mind doesn't need to; you learned to do it once when you were a child, and now it just happens. You have a kind of built-in shortcut to that sequence of steps.

Now let's try another simple example:

Count the number of words in this sentence: "Programming really can be fun."
Hopefully you decided that there were five words. But how did you come to that decision? First you had to decide what a word is, naturally. Let's assume for the moment that a word is a sequence of letters which is separated from other words by a space.

Using that rule, you do indeed get five words when looking at the sentence above. But what about this sentence:

"Sir Cecil Hetherington-Smythe would make an excellent treasurer,minister."
Notice my deliberate mistake: I didn't leave a space after the comma. You might also feel it's a mistake to name anyone Cecil Hetherington-Smythe, but that's a debate for another time. Using our rule about sequences of letters which are separated from other sequences by spaces, you would decide that there were eight words in the new sentence. However, I think we can agree that there are in fact ten words, so our rule clearly isn't working. Perhaps if we revised our rule to say that words can be separated by spaces, commas or dashes, instead of just spaces. Using that new rule, you'd indeed find ten words. Now let's try another sentence:

"Some people just love to type really short lines, instead of using the full width of the

page."

Although it might not be obvious, there is no space after "really", nor is there a space after "full".

Instead, I took a new line by pressing the return key. So, using our newest rule, how many words would you find in that last sentence? I'll tell you: you'd get sixteen, when in fact there are eighteen.

This means that we need to revise our rule yet again, to include returns as valid word-separators. And so on, until another sentence trips up our rule, and we need to revise it yet again.

You might wonder why we're doing this at all, because after all, we all know what we mean when we say "count the number of words". You can do it properly without thinking about any rules or valid word-separators or any such thing. So can I. So can just about anyone. What this example has shown us is that we take for granted something which is actually a pretty sophisticated "program" in our minds. In fact, our own built-in word counting "program" is so sophisticated that you'd probably have a lot of trouble describing all the actual little rules it uses. So why bother?

The answer comes in the form of an exceptionally important truth which you must learn. It's to do with computers (even your own computer that you're using to read this). Here it is:

- Computers are very, very stupid.

To some people, that statement is almost sacrilegious. You can understand that, because computers are really expensive. If you've just bought a Ferrari, you probably don't want your neighbour to come along and say that it's ugly and slow. If you're a Mac user (as I am, so don't send abusive email about this - I'm typing this article on my beloved PowerBook), you might actually kill a person who said your computer was stupid. Nevertheless, it's true - computers are desperately stupid. Your computer will sit there and do whatever mindless task you tell it to, for days, weeks, months or years on end, without any complaints or any slacking-off. That's not the typical behaviour of something that's even slightly clever. It will also happily erase it's own hard disk (which is a bit like you deleting all your memories then pulling parts of your brain out), so we're clearly not dealing with an intimidatingly intelligent item.

In fact, computers are so painfully stupid that they require to be told, in minute detail,

how to do even the most laughably simple of tasks. It's quite pathetic, when you think about it (or perhaps we're pathetic, since we're willing to pay ridiculous amounts of money to own them). In fact, just about the only positive thing about computers is that they're completely obedient. No matter how crazed your instructions might be, your computer will carry them out precisely.

By now, hopefully you can see how this is all tying together. Programmers tell computers what to do.

Computers require these instructions to be precise and complete in every way. Humans aren't usually good at giving precise and complete instructions since we have this incredible brain which lets us give vague commands and still get the correct answer. Thus, programmers have to learn to think in an unnatural way: they have to learn to take a description of a task (like "count the number of words in a sentence"), and break it down into the fundamental steps which a computer needs to know in order to perform that task.

You may be feeling slightly uneasy at this point. I've admitted that programming is, in a way, unnatural. I've warned you about the spectacular stupidity of computers, so you're probably getting a small idea of the amount of task-description you'd need to do in order to make your computer do anything even vaguely impressive. But don't worry; there are some excellent reasons to become a programmer:

- Programmers make lots of money.
- Programming really is fun.
- Programming is very intellectually rewarding.
- Programming makes you feel superior to other people.
- Programming gives you complete control over an innocent, vulnerable machine, which will do your evil bidding with a loyalty not even your pet dog can rival.

At least some if not all of these points will instantly appeal to you as a human being, and it's none of my business which ones you find most attractive. Additionally, people have been programming for many years, and so have already written many task-descriptions ("programs") for a lot of common tasks, so you can use all their hard work to save yourself some trouble. Which is also a very attractive prospect, I'm sure.

If you're still not sure whether programming is for you, perhaps the next section will help you decide.

Who can be a programmer?

You might be wondering just who can become a programmer (probably because you've

read the title of this section, and it got you thinking). The clichéd thing to do would be to say "anyone!", but I won't do that, because it's just not true. There are a few traits which might indicate that the person would be a good programmer:

- Logical
- Patient
- Perceptive
- At least moderately intelligent
- Enjoys an intellectual challenge
- Star Trek fan
- Female *

* Actually, males and females make equally good programmers. I just said that to address the gender disparity which exists in the programming world; it's true that there are currently more male programmers than female, which is strange given that one of the first ever programmers (Ada Lovelace) was female. Besides, male programmers are especially eager to encourage more females to take up the profession, since they (the male programmers) traditionally have no social lives whatsoever. Or perhaps that's an unfair generalisation, and it's just me who has no social life.

As you can see, it does take a certain type of person to be a successful programmer (in my humble opinion). If you're not logical, you'll have trouble organising and constructing complex programs. If you're not fairly intelligent, you won't be able to break a task down into the necessary individual steps.

If you're not patient, you'll deliberately destroy your computer long before you manage to finish all but the most simple programs. If you don't enjoy intellectual challenges, you'll hate every minute of programming. If you're not a Star Trek fan, you'll miss out on a large percentage of the conversation your fellow programmers will have (I'm not saying that's a bad thing). Finally, if you're not perceptive, you won't notice that the twenty consecutive hours you've spent in front of your computer have depleted your body's energy resources to a near-fatal level, and you'll die whilst designing an icon for your new application program (tip: experienced programmers avoid this problem by designing the icon before writing the program).

In other words, if you get annoyed waiting for the five beeps after your microwave finishes cooking a Pot Noodle, and you fly into a rage trying to solve the "coffee break" crossword in the tabloid newspapers, you may want to consider another line of work. That's not to say that you can't be a programmer; you just might find it more difficult than it needs to be.

By now you should at least have an inkling of whether or not programming could be a serious proposition for you. If you're interested in finding out what's actually involved in programming (the actual process of writing programs), read on. If not, then it's clear that you should instead leave the programming to others, and then buy their excellent software (hint hint).

What's actually involved in programming?

In this section I'm going to introduce some technical terms. It's unavoidable. The jargon will always get you in the end (if you choose to become a programmer, soon you'll actually be talking in jargon).

However, I'll explain each term as we come to it, and there's a list of jargon in the following section (with explanations, of course).

So, what's actually involved in programming - the actual process of writing programs? I'll answer that myself even though I asked the question, since I'm assuming that you don't know yet. Here's a quick overview of the process:

1. Write a program.
2. Compile the program.
3. Run the program.
4. Debug the program.
5. Repeat the whole process until the program is finished.

Let's discuss those steps one by one, shall we? Yes, we shall. And here goes.

1. Write a program.

I have a small amount of bad news for you: you can't write programs in English. It would be nice indeed to be able to type "count the number of words in a sentence" into your computer and have it actually understand, but that's not going to happen for a while (unless someone writes a program to make a computer do that, of course). Instead, you have to learn a programming language.

You're now probably having nightmarish flashbacks to French class in high school, sweating over which nouns took the verb "avoir" and which didn't. Thankfully, learning a programming language is nothing like that - for one thing, much of a programming language is indeed in English. Programming languages commonly use words like "if", "repeat", "end" and such. Also, they use the familiar mathematical operators like "+" and "=". It's just a matter of learning the "grammar" of the language; how to say things properly. Since there are many possible languages you could choose to learn (see the jargon section for a bit more on some of the languages), I won't go into any specific

one here. I just wanted to make you aware of the fact that you won't be typing your programs in English.

So, we said "Write a program". This means: write the steps needed to perform the task, using the programming language you know. You'll do the typing in a programming environment (an application program which lets you write programs, which is an interesting thought in itself). A common programming environment is CodeWarrior, and another common one is InterDev, but you don't need to worry about those just yet. Some programming environments are free, and some you have to buy just like any other application program. Commercial (non-free) programming environments cost anything from $50 to $500+, and you'll almost always get a huge discount if you're a student or teacher of some kind. Again, see the jargon section for where to find some programming environments.

Incidentally, the stuff you type to create a program is usually called source code, or just code.

Programmers also sometimes call programming coding. We think it sounds slightly more cool.

2. Compile the program.
In order to use a program, you usually have to compile it first. When you write a program (in a programming language, using a programming environment, as we mentioned a moment ago), it's not yet in a form that the computer can use. This isn't hard to understand, given that computers actually only understand lots of 1s and 0s in long streams. You can't very well write programs using only vast amounts of 1s and 0s, so you write it in a more easily-understood form (a programming language), then you convert it to a form that the computer can actually use. This conversion process is called compiling, or compilation. Not surprisingly, a program called a compiler does the compiling (that's why it's called a compiler and not, for example, a banana).

It's worth mentioning that if your program has problems which the compiler can't deal with, it won't be able to compile your program (in this situation, some programmers say that the compiler puked or barfed - which I'm sure you'll agree is just delightful).
You'll be pleased to hear that your programming environment will include a suitable compiler (or maybe more than one compiler: each different programming language your programming environment allows you to use requires its own compiler). Compilers are just fancy programs, so they too are written by programmers. Programmers who write compilers are a bit like gods; they make it possible for everyone else to program. So if anyone ever tells you that they're a compiler programmer, be sure to buy them a cup of

coffee (they will definitely like coffee, you can be sure).

3. Run the program.
Now that you've compiled the program into a form that the computer can use, you want to see if it works: you want to make the computer perform the steps that you specified. This is called running the program, or sometimes executing it (I'm aware of the potential irony of that term). Just the same as how a car isn't much use if you don't drive it, a program isn't much use if you don't run it. Your programming environment will allow you to run your program too (as you can see, programming environments do rather a lot for you).

4. Debug the program.
You've probably heard the term "debug" before (it's pronounced just as you might expect: "dee-bug").

It refers to fixing errors and problems with your program. As I'm sure you know, the term came about because the earliest computers were huge building-sized contraptions, and actual real-life insects sometimes flew into the machinery and caused havoc with the circuits and valves. Hence, those first computer engineers had to physically "debug" the computers - they had to scrape the toasted remains of various kinds of flying insect out of the inner workings of their machines. The term became used to describe any kind of problem-solving process in relation to computers, and we use it today to refer purely to fixing errors in our code. I've never had an insect fly inside my computer (though I hear that cats love to lie on top of them, so be sure to de-cat your computer before debugging your programs).

You may also have heard the phrase "it's not a bug, it's a feature". Programmers sometimes say this when someone points out a problem with their programs; they're saying that it's not a bug, but rather a deliberate design choice (which is almost always a lie). This is rather like accidentally spilling coffee all over yourself whilst simultaneously falling down some stairs, then getting up and saying "I meant to do that".

Once again, your programming environment will help you to debug your programs (indeed, you'll often find the picture of an insect shown in your programming environment to indicate debugging).

You usually debug your program by stepping through it. This means just what it sounds like: you go through your program one step at a time, watching how things are going and what's happening.

Sooner or later (usually later), you'll see what's going wrong, and slap yourself upside the head at the ridiculously obvious error you've made. Ahem.

5. Repeat the whole process until the program is finished.
And then you repeat the whole process until you're happy with the program. This is more tricky than it might sound, since programmers are never happy with their programs. You see, programmers are perfectionists - never satisfied until absolutely everything is complete and elegant and powerful and just gorgeous (though this pursuit of perfection doesn't always extend to their choice of girlfriends/boyfriends, by necessity). Programmers will commonly release a new version of their program every day for a couple of weeks after the initial release (just ask my friends about this).

As you can imagine, enjoying an intellectual challenge is an important trait to have when you're going back to correct and enhance your code many times over. You'll actually find that you can't wait to get back into your program and fix the bugs, make improvements, and refine the existing code. No, really you will!

And that's the basic process of programming. Note that most programming environments will make a lot of it much easier for you, by doing such things as:

- Warning you about common errors
- Taking you to the specific bit of code which is causing the compiler to puke
- Letting you quickly look up documentation on the programming languange you're using
- Letting you just choose to run the program, and compiling it automatically first
- Colouring parts of your code to make it easier to read (for example, making numbers a different colour from other text)
- And many other things

So, don't worry too much about the specifics of compiling then running then debugging or whatever.

The purpose of this section was mostly to make you aware of the cyclical nature of programming: you write code, test it, fix it, write more, test it, fix, and so on.
Now, we digress for a moment (or however long it takes you read the next section), to talk briefly about something called scripting.

A brief word about scripting

You've perhaps heard about something called scripting, or maybe you've heard of languages like JavaScript, AppleScript, Tcl and others (those languages are called scripting languages). You may thus be wondering if scripting is the same as

programming, and/or what the differences are, and so on.

People get quite passionate about this question, so I'm just going to cover it briefly and technically.

Here are some facts:

- Scripting is essentially a type of programming
- Scripting languages have a few minor technical differences which aren't important to discuss at this point
- Scripting languages tend to be interpreted rather than compiled, which means that you don't need to compile them - they're compiled "on the fly" (i.e. when necessary, right before they're run). This can make it faster to program in them (since you always have the source code, and don't need to take the deliberate extra step of compiling)
- The fact that scripting languages are interpreted generally makes them slower than programming languages for intensive operations (like complex calculations)
- Scripting languages are often easier to learn than programming languages, but usually aren't as powerful or flexible
- For programming things like applications for personal computers, you'll need to use a programming language rather than a scripting language

Scripting languages can be excellent for beginners: they tend to be easier to learn, and they insulate you from some of the technical aspects of programming (compiling, for one). However, if you're serious about programming, you won't be able to stay with a scripting language forever - you will move on to a programming language at some point. I'd say that it's good to know a scripting language or two, and even to start with a scripting language rather than a programming language. However, there's a point of view which says that, by protecting and "hand-holding" too much, scripting languages don't properly prepare you for "serious" programming, and set you up for a bit of a learning curve when you move on to a programming language. I can't really tell you whether that's true one way or the other; all I can say is that I started with scripting languages like JavaScript and AppleScript, and moved on to programming languages. I'd advise that you try a programming language, and if you find it really hard going, try a scripting language to ease you into thinking like a programmer.

We now move on to the "where to start" section, which is funny considering how much we've talked about already. Such is life.

Where to start

The question of exactly where to start learning to program is a big one, and it's

something that everyone has an opinion about (even your postman may have a strongly-held opinion on this subject).

I'm going to try to be as practical as possible about answering this, but just be aware that personal preferences (in terms of programming languages, books, programming environments, and so on) will always creep in somewhere. I do however promise to be as impartial as possible (which isn't too difficult when you're as unquestionably great as I am).

You may think that the obvious first step would be to choose a programming language, probably by weighing up the various pros and cons of each major language. However, when someone begins a sentence with "you may think that", there's a fair chance that they're going to tell you that you were wrong to think whatever it was. I'm not exactly going to do that; instead I'm going to point out something that you must realise before going any further:

- Programming is programming

You're probably thinking "you don't say" (or something less kind), but there really is a valid point in there. You see, there are fundamental similarities between all programming languages, so as a beginner it's really not so important which language you begin with; the important thing is to just get some experience with programming.

So, you're really free to choose a language, up to a point. Just be aware that there's no "wrong" language to start with. Having said that, there are of course some other considerations you might want to take into account, for example these:

- Do you want to pay for a programming environment, or use a free one?
- Do you want to learn with a language you'll still be able to use professionally?

You see, each programming environment will only support certain languages. If you're going to use a free programming environment and compiler at first, you may have a limited choice of languages.

Also, you may (understandably) want to learn with a language which you'll still be able to use professionally. That's a normal feeling, and it's probably a good idea too, but just don't feel that you have to learn with a "commercially viable" language - that's nonsense.

As I said, just get started programming, whatever the language may be - your skills will be entirely transferrable to other languages, and you'll find each new language much (much) easier after learning your first one.

Let's assume that you want to use a commercial language (meaning one which is used by professional programmers in the current mainstream software industry), and you're willing to pay for a programming environment if need be. The main choices basically boil down to these, in no particular order:

- C
- C++
- Java

(Remember that you can find out more about these languages in the jargon section, later in this article). For now, let's talk about each language very briefly.

• C

This is probably the most widely-used, and definitely the oldest, of the three languages I mentioned.

It's been in use since the 70s, and is a very compact (i.e. not much "vocabulary" to learn), powerful and well-understood language (by "well understood", I mean that there's a huge body of literature and example source code related to C). It would be an excellent choice. I'd recommend it to you as a starting language if you asked me for my own opinion (but you didn't ask me, so I won't say anything).

• C++

This language is a superset of C (that just means that it's C with more stuff added; it's more than C, and includes pretty much all of C). Its main benefit over C is that it's object oriented. Don't worry about what that means right now, but feel free to check the jargon section for more about it. The key point is that object oriented languages are more modern, and using object oriented languages is the way things are done now in the programming world.

However, don't take that the wrong way. You'll notice that I said C++ included pretty much all of C (in case you're wondering; there was no "C+"). This means that C is at the core of C++, and you need to know C in order to use C++. I also said that C++ is essentially an object oriented version of C, but it's not the only enhanced version of C which is object oriented. You've got basic C, and you've got a few languages which are object oriented, enhanced versions of C, and C++ is only one of those. What I'm trying to convey by saying this is that it's a better idea to learn C, then you can move to whatever object oriented version of C you want to. If you dive in and learn C++ first, you may find it harder to switch to other C-based object oriented languages since you'll have to "forget" parts of C++. I hope that makes sense.

Let me just balance that (which was my own opinion) by saying that C++ is most definitely the second-most used language after C, and may soon become the most used language. I'd say that it's certainly the language of choice for most commercial application software development on Windows and Macintosh, which says a lot. I just feel that you can learn C and not suffer (you can still go on to C++ afterwards, very quickly indeed since you'll already know the vast majority of C++ by virtue of knowing C), and still also leave your options open.

• Java
You've almost certainly heard of Java; it's been hyped all the way to the moon and back. Java has a benefit which other programming languages lack: it's cross-platform. So what exactly does that mean? Well, it means that it runs on more than one platform without needing to be recompiled. A platform is just a particular type of computer system, like Windows or Mac OS or Linux. Normally, if you wanted to use the same program on a different platform from the one it was written on, you'd have to recompile it - you'd have to compile a different version for each different platform.

Sometimes, you'd also need to change some of your code to suit the new platform. This probably isn't surprising, since the different platforms work differently, and look different (an alert window on Windows looks different than an alert window on Mac OS, for example, and they both use different code).

Anyway, Java can run on more than one platform without needing to be recompiled. You can understand why that would be a serious advantage. However, Java also has a disadvantage which is
almost as serious: it's slow. Java achieves its cross-platform trick by putting what is essentially a big program on your computer which pretends to be a little computer all of its own. The Java runs inside this "virtual machine", which runs on whatever platform you're using (like Windows or Mac OS).

Because of this extra layer between the program and the computer's processor chip, Java is slower than a program written and compiled natively for the same platform. Hopefully you can roughly understand why that might be. It's for the same reason that it's quicker to speak French to a French person directly than to speak English to an interpreter, and have the interpreter repeat what you said in French to the French person. It's exactly like that with Java and its Virtual Machine. In fact, I'm rather proud of that analogy (so don't steal it).

Having said all that, remember that I also mentioned how much Java has been marketed and hyped.

Because of this, Java has become very popular. Another reason for its popularity is that it runs inside web browsers, letting programmers create little applications which can run on web sites. This is obviously a big attraction, given the explosion of the internet over the past several years. Due to this popularity, there are a lot of jobs out there for Java programmers at the moment. I'm just making you aware of this fact as it's naturally going to be one of your considerations. I wouldn't say that there's more demand for Java programmers than for C or C++ programmers, but there will probably be more demand for Java programmers than for programmers who use languages I've not yet mentioned.

Now let's talk about books, and more generally about reading material, help and tutorials. One of the first things you should do after thinking about a language is to research it a bit. Do a web search, and you'll doubtless find many thousands of relevant web sites. Take a quick look around a few of them, and you'll notice that there are plenty of mailing lists (email lists which you can send mail to, and everyone who's on the list receives it and can reply to it) about programming languages, so those are an excellent place to get help (but do learn at least some of your programming language first). You'll quickly see how popular a language is by how much material you find about it on the internet. You'll also be able to find programming environments, both free and commercial, in the same way. When you're comfortable that you've chosen the right language for you, it's time to get a book.

Don't let anyone tell you otherwise: there's no substitute for being able to hold a book. I'm not claiming that they'll never be replaced with Star Trek-like electronic pads; I'm just saying that they won't be replaced until the pads offer the same light weight, sharpness of text, and portability. At the moment, you can only get that in a good old-fashioned book (and don't say that laptops offer all that; they don't, and I know because I'm typing this article on a very nice one, which is nevertheless not as enjoyable to read as a printed book). But I digress.

You'll want to find a suitable beginner's book on your chosen programming language, and you'll be pleased to know that there will be plenty to choose from (particularly so if you chose one of the three languages I mentioned earlier). Generally speaking, go by the opinions of others, not the blurb on the books themselves. An excellent way to do this is to read the customer reviews at places like amazon.com (and amazon.co.uk, of course). Some of the comments are inevitably semi-literate nonsense, but you'll get an excellent general impression of the book's quality and relevance. Just one tip about that: when weighing up the reviews, discard the top and bottom 5-10% (in terms of ratings).

There will always be overly negative people, and there will most certainly always be

those who lavish excessive praise on anything they lay hands upon (perhaps surprisingly, you'll always find more of them than of the overly negative people). But that's just common sense. I can't give a specific suggestion regarding Java since I haven't read many Java books (though I will say that the book I have here now is Beginning Java 2 by Ivor Horton, published by Wrox). Regarding the other two languages, I would suggest titles as follows:

•C

The C Programming Language by Brian W Kernighan and Dennis M Ritchie, published by Prentice Hall. Make sure you get the 2nd Edition. Let me just say that this was written by the people who created the C language, and is seen as the classic programming text. It's commonly referred to as "K&R", after its authors. However, it's quite challenging, and I wouldn't tend to recommend it for beginners.

For beginners I would tend to recommend you also get:

The Absolute Beginner's Guide to C by Greg Perry, published by Sams. This is great because it gives a good introduction to programming in general as well as giving you a good grounding in C. It also talks about how to use various popular programming environments, which will be useful if you've never used one before.

• C++

The C++ Programming Language by Bjarne Stroustrup, published by Addison-Wesley. This was written by the programmer who created C++, so it's certainly authoritative. I don't have this one myself, but I would presume it will be similar (in terms of its target audience) to The C Programming Language, detailed above. Make sure you look for a suitable beginner's C++ book also.

You might feel that this section is a bit dismissive, but you really will benefit hugely from getting hold of an appropriate book and working through it. Let me just finish this section with two tips for when you're going through your new books:

1. Take your time, and don't expect to go too fast.

It may take you an hour to go through a single page and really understand everything (you may particularly find this with The C Programming Language, which is focused almost to the point of terseness), but that's fine. The point isn't to get through the book; it's to understand the subject. A programming textbook is like any other textbook; you don't read it, you work through it (and you keep going back to it). It will be an extremely enjoyable and enlightening experience, so enjoy every moment of it.

2. Do the exercises.

That sounds like a really obvious thing to say, but you'd be amazed how many people don't do the exercises - they don't type in the code they see in the book, they don't run it,

and then they come away from the book when they're only about a third of the way through, having found that they suddenly ran up against an impossible-to-understand section after they thought they'd been doing so well. It's an old story indeed (I know, because I have first-hand experience - I never used to do the exercises, and only when I finally tried doing it "by the book" did I suddenly realise how important they were).

Just do it to humour me, or as thanks for writing this huge article. I'll go so far as to say this: if you don't do the exercises, you may still understand every bit of the book, and get through it all, but you won't have learned even half of what you might have. Also, you won't be as good a programmer – I guarantee it. There are so many things you miss by not doing the exercises:

- You miss typing in the code and making typing errors which make the compiler puke
- You miss having to debug your program because you mistyped something in it
- You miss the immersion you get from editing the code yourself
- You miss the feeling of control you get from typing the code in yourself
- You deprive your brain of the chance to assimilate the code line by line

Really, it's that much of a loss. You might think that the first two points are reasons not to do the exercises, but you'd be wrong - we learn much more from those kinds of mistakes than from reading some code in a book. If you learn nothing else from this article, just learn that you should always do the exercises. I promise you'll thank me later (or at least you would if you remembered to).

Now, at last, it's time to move on to the dreaded jargon section, where grown men and women have been known to go completely insane after only moments of exposure (though that might just be hearsay). It's not required reading, but it might really help to give you a leg up on some of the terms which you will definitely run into at some point. You can also be fairly sure that nowhere else will you find them explained as sensibly and clearly as they are here (you'd be shocked at the bizarre and convoluted definitions and explanations some people come up with for programming jargon – those people often go on to write documentation for major application programs).

Programming jargon

This section lists various terms related to programming (and computers in general, but only those which are in some way relevant to programming), in alphabetical order. The terms aren't split into categories or topics, because it's presumed that in most cases you wouldn't be sure which category to look at in the first place (people really do categorise glossaries; those people should probably be forced to take part in Star Trek trivia competitions with expert programmers).

Note that terms in bold are defined in this section, so if you see a bold term you're not familiar with, just look in this section for an explanation. Now, without further ado, here we go.

• Ada
A programming language, sometimes with a number after the name (e.g. Ada95). Used commonly for defense applications (particularly in the UK). A fairly common choice as a first language in university computer science courses (again, particularly in the UK). Named after Ada Lovelace, a very clever lady who was one of the first ever programmers (I told you that gender had nothing to do with programming ability).

• API
An Application Programming Interface. An API is a kind of predefined standard for how you should write certain types of programs, and commonly refers to a standard for how to write programs which interact with and extend other programs. For example, you probably know that your web browser can use plug-ins to allow it to display more kinds of content. You might have a plug-in that lets you view PDF files, or a plug-in that lets you view Quicktime movies. The reason people can write these plug- ins is because the browser has a "plug-in API" - a predefined set of rules that programmers should follow if they want their plug-ins to work with the web browser.

APIs are everywhere. The reason all Windows applications look much the same (the same style of buttons and checkboxes and menus) is that there's a Windows programming API. Any time you're defining a standard way of writing code to work with something, you're creating an API. If someone asks about the API for something, they're asking for information on how to properly write programs which work with whatever the thing is.

• Apple Computer
A computer software and hardware company based in Cupertino, California. Apple are the creators of the Macintosh (sometimes just called Mac) line of computers, and of the Mac OS operating system. Apple invented many of the technologies which all computers use today. People who own Macs are typically intensely fond of their computers, and passionately evangelise them. Many surveys have found that Mac users are amongst the most loyal computer users in the whole computer industry.

• application
A program designed to let you accomplish a specific task. For example, a word processor (like Microsoft Word, or WordPerfect) is an application. So is a graphics

program like Photoshop or Paint.

• application framework
A sort of template to help you make applications. With some programming languages, there's quite a lot of work involved in creating a full application for a GUI operating system, so it helps to have the basics already done for you. A popular application framework is PowerPlant, which is included with the programming environment called CodeWarrior.

• ASCII
American Standard Code for Information Interchange. A way of specifying alphabetical letters, digits, punctuation marks and so on as numbers. Computers deal with numbers by design, so it's easier and more efficient for them to represent things as numbers rather than as anything else. The ASCII system lets you represent letters and other characters as numbers; for example, a capital "A" is represented by the number 65. Sometimes, people use the term "plain text" to refer to text which is encoded as ASCII (notably in email programs).

• barf
Also sometimes puke. To run into a programming error; usually used to refer to a compiler, in this form: "I'd left out a semicolon, and the compiler just barfed."

• BASIC
A programming language, or more accurately a family of programming languages since there are many versions of BASIC. The term stands for Beginner's All-purpose Symbolic Instruction Code, which sounds technical but really isn't: it's for beginners, it's all-purpose rather than being designed for a specific type of programming (quite a few languages are designed for just one type of programming, like business programming or scientific programming), and it's a "symbolic instruction code" which just essentially means that it's a programming language. As its name implies, any version of BASIC tends to be good for those who are just starting to program. For that same reason, BASIC tends to have a bad reputation amongst "serious" programmers, for being "too simple" or inflexible. Of course, the real reason they're annoyed is that BASIC makes it even easier for people to take up programming, so there's more competition for all the fabulous software products programmers create.

• binary
A way of counting which involves just two digits: 0 and 1. Our normal way of counting (with ten digits) is called decimal. Computers use binary since processors really have millions of little switches, and switches can only be in one of two states: on or off. So, it

was sensible to use binary since you can represent off with 0 and on with 1. Sometimes you'll also hear programmers referring to "a binary". This means the same as object code; it's the compiled version of a program.

• Borland
A software company which makes programming environments, notably for C++ and Java programmers.

•C
A very popular programming language, created by Dennis M. Ritchie. C is a very compact and quite easy to learn language, and is an excellent choice for those intending to make a career out of programming. It is an especially good first language, since you can then move on to any of various object oriented versions of C, or other fairly C-like languages such as Java.

• C++
A programming langauge which is an object oriented version of C, and which was created by Bjarne Stroustrup. The modern language of choice for Windows and Mac OS application development.

• Carbon
An API for the operating system called Mac OS X, which also can be used with Mac OS versions 8- 9. Carbon is an updated version of the Macintosh Toolbox.

• COBOL
COmmon Business-Oriented Language; a programming language. As its name implies, COBOL was designed to be used for writing business programs. It was very popular during the 1960s and 70s, but is now seen as an essentially dead language.

• Cocoa
A very tasty hot drink. In computing, it also refers to an API for the operating system called Mac OS X, which runs on Macintosh computers. Cocoa is a very advanced object oriented API. Cocoa used to be called NEXTSTEP, back when it was owned by Next Computer Inc.

• code
The actual text of the programs you write, before you compile them. Sometimes called source code, or just source.

• CodeWarrior

A programming environment, including an editor, compiler, linker, debugger, application frameworks and RAD tools. It's called CodeWarrior because some programmers like to call themselves "code warriors", because it's a better title than "pasty-faced geeks". CodeWarrior is available in a full version and a starter version, and both have academic discounts available. You can get versions of CodeWarrior for Mac OS, Windows, Linux and any number of other platforms.

• compile
To convert source code into object code. Sometimes also used as a noun; you might refer to your compiled program as "a compile", or you might say you "did a compile" when you've compiled your program.

• compile
A special kind of program which compiles (converts) source code (which is easy for humans to read and write) into object code (which is what computers need). Compilers are almost always included with your programming environment.

• cross-platform
Able to run on multiple different types of computers and operating systems without needing to be re- compiled first. Java is cross-platform.

• debug
To find and correct errors and problems in your programs. You will probably use a debugger to help you debug. The term comes from the fact that the earliest computers were huge mechanical devices, and sometimes insects would fly into them and cause damage, thus the computer engineers had to physically remove dead bugs from the internal workings of the computers.

• debugger
A special program which lets you run your programs one line at a time, and keep track of everything that's going on in them, to help you identify and correct errors and problems. Debugging is an integral, if sometimes unpleasant, part of programming.

• editor
A program (much like a simple word processor) which you use to write your programs. It can be any text editor, or part of a programming environment.

• FOLDOC
The Free OnLine Dictionary Of Computing. A web site at http://www.foldoc.org/ where you can find definitions of pretty much any computing term. Use as a next step after

reading this list of common jargon.

• FORTRAN
Also "Fortran" (not all in capital letters). FORmula TRANslation; a programming language. FORTRAN is a programming language designed to be used for writing scientific and numerical programs. Still in use, although not really a mainstream language.

• GUI
A Graphical User Interface. The idea of having little pictures (icons) of things instead of just text. You'll be familiar with a GUI, since you're almost certainly using one right now. The concept of having a mouse pointer, menubars, buttons, checkboxes, windows, folders, document icons and so on is all part of a GUI.

• hack
A program which was written very quickly and/or carelessly, or a program which does something unconventional or frivolous. Can also be used as a verb, meaning simply to write computer programs.

• hacker
Traditionally, simply a computer programmer - and this meaning is still in use. More recently, the term also refers to a person who maliciously compromises or damages computer systems (often over the internet).

• hardware
Any physical piece of machinery or electronics. A computer is hardware, as is a mouse or keyboard or printer. A tangible, physical piece of equipment. Compare with software.

• InterDev
A programming environment from Microsoft. It allows you to program in various different programming languages.

• Interface Builder
A RAD and object oriented programming tool included with Mac OS X. Used in conjunction with Project Builder to create applications for Carbon, Cocoa and Java.
• Java
A programming language which is object oriented and can run on many different computers without needing to be re-compiled for each one. Very popular at the moment, but can be slow in comparison to other languages.

• JavaScript
A scripting language used within web pages to add basic interactivity and so on. A completely different language from Java, despite the similar name. Often confused with Java by those who don't know the difference.

• linker
Usually included with a compiler. A special program which links together all the bits of object code produced by a compiler. You'll rarely, if ever, have to explicitly use a linker; it's always automatically taken care of by your programming environment after your program is compiled.

• Linux
An open source, Unix-like operating system. Created by a person called Linus Torvalds, hence it's "Linus' Unix", or Linux.

• Macintosh
Also called Mac. A line of computers created by Apple Computer. The first mainstream GUI computers (though the first ever GUI computer was created by Xerox Corporation).

• Macintosh Toolbox
The API for Macintosh computers running the Mac OS operating system up to version 9. Now updated for Mac OS X and called Carbon.

• Mac OS
The operating system which runs on Macintosh computers. Used to refer to versions up to and including 9 (not including Mac OS X).

• Mac OS X
A radically new operating system for recent Macintosh computers. It combines the power of Unix with the friendliness of the Macintosh GUI.

• object code
A program which has been compiled into a form suitable for the computer to use. Also sometimes called a binary.

• object orientation
Also called OO. A modern programming concept where the programmer creates "objects" like real- life objects, with both properties and abilities. In traditional programming, the program was very separate from the information it acted upon. That's not very much like real life; objects in real life have both properties (like the colour of

your hair, or your height) and abilities (like your ability to read this article, or your ability to tie your shoelaces). Object oriented programming essentially tries to allow programmers to think (and program) in a more natural and familiar way. Popular modern object oriented programming languages include Java and C++ (and my own favourite, Objective-C).

• Objective-C
Sometimes also Obj-C or ObjC. A programming language which is an object oriented version of C. Used as the language of choice for developing programs to run in the Cocoa environment on Mac OS X. A very easy to learn and powerful language. My own language of choice.

• OO
Another way of saying object orientation (or object oriented).

• open source
The idea that the source code of a program should be available to everyone, as well as the compiled version. This allows any programmer to modify and enhance the program as they see fit, and it allows new programmers to see exactly how the program was written. Programs which are open source are free, since anyone can get the source and compile it themselves.

• operating system
The special and very important program which makes your computer work. It takes care of things like talking to the screen, printer, keyboard and all the other hardware. Without an operating system, your computer would just sit there staring blankly into the distance, like you do after you've had a few beers.

• Pascal
A programming language designed for teaching programming; made to be as simple as possible. However, this simplicity is also a weakness, and has lead to Pascal being denounced as a "toy" language, just for hobbyists and rudimentary teaching purposes. There are many variations of Pascal available.

• Perl
A programming language originally created for Unix computers. Very powerful for manipulating text, and very popular for writing programs to help run web sites.

• programming
The art of writing computer programs (and it is indeed an art). What this article is all

about, so you shouldn't need this definition.

• programming language
Any of countless special languages used to write programs. Usually not difficult to learn, and including several English words like "end" and "repeat" and "if".

• programming environment
A special program (or set of programs) which help people to write other programs. Just like you use a word processor program to write letters to your Aunt Mary, you use a programming environment to write programs. Commonly includes a compiler, linker, debugger and sometimes other things too.

• Project Builder
An excellent programming environment included with Mac OS X. Used in conjunction with Interface Builder to create programs for Carbon, Cocoa and Java.

• PROLOG
PROgramming in LOGic; a programming language. PROLOG was the first of many languages designed to use so-called "logical programming" techniques. The basic idea is that you define a set of facts, then define a goal, and the computer tries to get to the goal using the facts you've defined.

• puke
Another way of saying barf.

• RAD
Rapid Application Development. A concept where you create application programs very quickly by visually dragging controls (like buttons and checkboxes and so forth) into windows, to create the user interface for your application. Many programming environments now include RAD tools.

• REALbasic
Sometimes also called RB. An object oriented version of BASIC which includes a programming environment with an editor, compiler, linker, debugger, and RAD tools. REALbasic's programming environment runs on Mac OS computers, but can compile applications for Windows and Carbon too. Probably a very good choice for Macintosh users who are just starting into programming.

• run
To make a computer perform the steps you've written in your program; to make the

program do whatever it does. Sometimes also called "executing" a program.

• SDK
A Software Development Kit. Typically a package of sample code, documentation and other items to help you create certain kinds of programs. For example, if you wanted to create a plug-in for Adobe Photoshop (that's a program which lets you edit pictures), you'd probably want to get hold of the Photoshop plug-ins SDK. Sort of like a toolkit for programmers.

• scripting
A type of programming using a scripting language.

• scripting language
A special kind of programming language which isn't compiled before you run it; it is compiled automatically as needed, right before it's run. Often easier than traditional programming for a few very technical reasons which we won't go into. Might be a good place to start if you find normal programming languages too difficult.

• software
Any computer program. It's called software because it's not tangible; not "hard". Compare with hardware.

• source
Another name for code.

• source code
Another name for code.

• string
Any piece of text. Computers refer to any amount of alphabetical letters or digits or punctuation marks etc as a "string".

• Unix
A very general group of operating systems, known for their power, stability and reliability, but also acknowledged to be more difficult to learn and master than a GUI operating system. Extremely popular operating systems for running web servers.

• user interface
The appearance of your program to the person using it. The windows, menus, buttons and so on are all collectively called the user interface. Sometimes just called UI.

• virtual machine
A program which emulates (pretends to be) an entire little computer all of its own. Java uses a virtual machine, which allows Java code itself to be cross-platform, at the expensive of some speed.

• virtual memory
A technique whereby a computer uses part of its hard disk as temporary memory for a program. A way to use more memory than your computer actually has, but at the expense of considerable speed.

• VM
Either virtual machine or virtual memory.

• wetware
What programmers sometimes humorously call the human brain; the ultimate computing device. See also hardware and software.

• Windows
One of several operating systems from the company called Microsoft. Collectively they are the most commonly used operating systems in the world. Each version of Windows tends to have a year after it instead of a version number (for example, Windows 98). Windows is a graphical operating system (or GUI operating system). This brings us to the end of the list of jargon (you can wake up now). For a much more extensive list of computer related terms, be sure to look at the FOLDOC definition, in the list above - it will give you the address of a web site you can visit to find the meaning of any computer term ever used by anyone, anywhere (probably).

Further information
This is the final section of the main article, so congratulations (and heartfelt thanks) if you've made it this far. This section gives a brief list of some web sites you might want to visit to find out more about certain topics we've covered previously. I've split the sites into categories, and mentioned which terms or items they're relevant to. So here goes.

• All programming topics
http://www.programmersheaven.com/
Packed full of resources for every kind of programmer (even me).
http://www.programmingjobs.com/
A place for US programmers to post their resumés, or find a programming job. As indeed you might expect from the name.

http://www.seriousprogramming.com/
Lots of information, including tutorials on C and C++, information on how to use a compiler, and more. Covers lots of programming and scripting languages too. Don't be put off by the "serious" name; there's lots of material for beginners.

• Books
http://www.amazon.com/ and http://www.amazon.co.uk/
Go and read all those customer reviews of appropriate books for your chosen programming language.

• C/C++
http://www.cprogramming.com/
A good site for information and tutorials on C and C++.

• CodeWarrior
http://www.metrowerks.com/
The company which makes CodeWarrior, and other programming tools.

• Geeks
http://www.thinkgeek.com/
Essential clothing, beverages, books and toys for the discerning geek. Highly recommended.
http://www.startrek.com/
Assemble an away team and take a look around whilst your ship is docked for repairs. Damn those Borg.

• Java
http://java.sun.com/
The Java site from Sun, the company which created Java. Authoritative, as you might expect.

• Objective-C / Project Builder / Interface Builder / Cocoa
http://www.stepwise.com/
Excellent tutorials and articles on Objective-C and Cocoa (which used to be called NEXTSTEP, hence the name "Stepwise").
http://developer.apple.com/
The developer site from the company who brought us Project Builder, Interface Builder and Cocoa.
Lots of good material here, including sample code and technical articles.
http://www.cocoadev.com/

An excellent community-built site for those new to Cocoa and Objective-C. I've contributed material there on a few occasions, and there are some excellent programmers ready to help you out. This site is also particularly interesting since every page of it is editable by anyone who visits. You can actually add your own pages and edit existing pages just using your web browser, without needing to log in or register first. So play nice, please. http://www.cocoadevcentral.com/
Another great Cocoa site, with an emphasis on new users and question-and-answer type articles.

• REALbasic
http://www.realsoftware.com/
The home of REALbasic. Download a free trial version here, or read all the documentation. Plenty of links to source code and such also.

One final thing

Traditionally, this is the place where the author gives a final, rousing inspirational speech, then goes merrily on his way feeling like a fabulously generous and wise person. I intend to follow that tradition to the letter.

If you're read this far and understood at least most of what I've said, you have sufficient intelligence to pursue a career in programming. It doesn't have to be expensive (there are lots of free programming environments, compilers and so on, and even the commercial ones have starter versions and academic discounts), and it really can be a huge amount of fun. Despite my somewhat deadpan tone, you can tell by the fact that I wrote all this that I'm passionate about programming, and would dearly like to see even more people taking it up (so that I can steal their code later).

Programming is an extremely rewarding activity. The thrill of thinking your way around a complex problem, and the satisfaction of seeing your program perform the desired task perfectly, are hard to describe. I've always felt that writing computer programs is very like composing music; you take an initial fully-formed idea, and break it down and analyse it until you properly understand how it's formed, then you go about actually building it, piece by piece. So much art and elegance goes into writing programs, and there's considerable room for interpretation and personal style to show through.

Programming is seen as an exceptionally technical and boring profession, yet in all the years I've been doing it I've never tired of it. There really is a genuine thrill about finding a new problem, and engaging your mind and all your knowledge and experience to solve it. Programming is about creating; about making things possible and bringing ideas to life. In that respect it's no different from painting or sculpture or music, but it's in many ways even more rewarding because your creation actually does something in itself.

Write a program, and watch it go. I'll never tire of seeing my programs go off and solve people's problems and make their lives a little easier. There's something very powerful in the ability to create something which can perform a function, and interact, and process and contribute - to whatever extent you've granted it the ability to. You'll find that you become very fond of, and proud of, your programs. And of course you may well make vast quantities of money. Either way, there's no shortage of satisfaction.

Since you've read all this, you're at least thinking seriously about what programming is, and you may even be considering looking into it further. I urge you to do so. Ever notice how programmers make no apologies about the fact that they're programmers? That's because it really is good to be a programmer, as you'll hopefully be finding out.

Well, that's it; the lecture is over. Naturally I'll now go off and feel like a particularly wonderful person (a justifiable position, I'm sure you'll agree), and hopefully you'll go and start down the road to becoming a programmer. Before you go, or even afterwards, I'd love to hear any comments or relevant anecdotes you might want to share. I'll especially welcome gushingly positive reviews of this article, and free money. But I'll certainly settle for just hearing from you. Send me an email via my web site, which is listed at the very start of the article.

And good luck.

Unit 4 How to become a hacker

What Is a Hacker?

The Jargon File contains a bunch of definitions of the term 'hacker', most having to do with technical adeptness and a delight in solving problems and overcoming limits. If you want to know how to become a hacker, though, only two are really relevant.

There is a community, a shared culture, of expert programmers and networking wizards that traces its history back through decades to the first time-sharing minicomputers and the earliest ARPAnet experiments. The members of this culture originated the term 'hacker'. Hackers built the Internet. Hackers made the Unix operating system what it is today. Hackers run Usenet. Hackers make the World Wide Web work. If you are part of this culture, if you have contributed to it and other people in it know who you are and call you a hacker, you're a hacker.

The hacker mind-set is not confined to this software-hacker culture. There are people who apply the hacker attitude to other things, like electronics or music - actually, you can find it at the highest levels of any science or art. Software hackers recognize these kindred spirits elsewhere and may call them 'hackers' too - and some claim that the hacker nature is really independent of the particular medium the hacker works in. But in the rest of this document we will focus on the skills and attitudes of software hackers, and the traditions of the shared culture that originated the term 'hacker'.

There is another group of people who loudly call themselves hackers, but aren't. These are people (mainly adolescent males) who get a kick out of breaking into computers and phreaking the phone system. Real hackers call these people 'crackers' and want nothing to do with them. Real hackers mostly think crackers are lazy, irresponsible, and not very bright, and object that being able to break security doesn't make you a hacker any more than being able to hotwire cars makes you an automotive engineer. Unfortunately, many journalists and writers have been fooled into using the word 'hacker' to describe crackers; this irritates real hackers no end.

The basic difference is this: hackers build things, crackers break them.

If you want to be a hacker, keep reading. If you want to be a cracker, go read the alt.2600 newsgroup and get ready to do five to ten in the slammer after finding out you

aren't as smart as you think you are. And that's all I'm going to say about crackers.

The Hacker Attitude

1. The world is full of fascinating problems waiting to be solved.
2. No problem should ever have to be solved twice.
3. Boredom and drudgery are evil.
4. Freedom is good.
5. Attitude is no substitute for competence.

Hackers solve problems and build things, and they believe in freedom and voluntary mutual help. To be accepted as a hacker, you have to behave as though you have this kind of attitude yourself. And to behave as though you have the attitude, you have to really believe the attitude.

But if you think of cultivating hacker attitudes as just a way to gain acceptance in the culture, you'll miss the point. Becoming the kind of person who believes these things is important for you - for helping you learn and keeping you motivated. As with all creative arts, the most effective way to become a master is to imitate the mind-set of masters - not just intellectually but emotionally as well.

Or, as the following modern Zen poem has it:

> To follow the path:
> look to the master,
> follow the master,
> walk with the master,
> see through the master,
> become the master.

So, if you want to be a hacker, repeat the following things until you believe them:

1. The world is full of fascinating problems waiting to be solved.

Being a hacker is lots of fun, but it's a kind of fun that takes lots of effort. The effort takes motivation. Successful athletes get their motivation from a kind of physical delight in making their bodies perform, in pushing themselves past their own physical limits. Similarly, to be a hacker you have to get a basic thrill from solving problems, sharpening your skills, and exercising your intelligence.

If you aren't the kind of person that feels this way naturally, you'll need to become one in order to make it as a hacker. Otherwise you'll find your hacking energy is sapped by distractions like sex, money, and social approval. (You also have to develop a kind of faith in your own learning capacity - a belief that even though you may not know all of what you need to solve a problem, if you tackle just a piece of it and learn from that, you'll learn enough to solve the next piece - and so on, until you're done.)

2. No problem should ever have to be solved twice.

Creative brains are a valuable, limited resource. They shouldn't be wasted on re-inventing the wheel when there are so many fascinating new problems waiting out there.

To behave like a hacker, you have to believe that the thinking time of other hackers is precious - so much so that it's almost a moral duty for you to share information, solve problems and then give the solutions away just so other hackers can solve new problems instead of having to perpetually re-address old ones.

Note, however, that "No problem should ever have to be solved twice." does not imply that you have to consider all existing solutions sacred, or that there is only one right solution to any given problem. Often, we learn a lot about the problem that we didn't know before by studying the first cut at a solution. It's OK, and often necessary, to decide that we can do better. What's not OK is artificial technical, legal, or institutional barriers (like closed-source code) that prevent a good solution from being re-used and force people to re-invent wheels. (You don't have to believe that you're obligated to give all your creative product away, though the hackers that do are the ones that get most respect from other hackers. It's consistent with hacker values to sell enough of it to keep you in food and rent and computers. It's fine to use your hacking skills to support a family or even get rich, as long as you don't forget your loyalty to your art and your fellow hackers while doing it.)

3. Boredom and drudgery are evil.

Hackers (and creative people in general) should never be bored or have to drudge at stupid repetitive work, because when this happens it means they aren't doing what only they can do - solve new problems. This wastefulness hurts everybody. Therefore boredom and drudgery are not just unpleasant but actually evil.

To behave like a hacker, you have to believe this enough to want to automate away the boring bits as much as possible, not just for yourself but for everybody else (especially other hackers).

(There is one apparent exception to this. Hackers will sometimes do things that may seem repetitive or boring to an observer as a mind-clearing exercise, or in order to acquire a skill or have some particular kind of experience you can't have otherwise. But this is by choice - nobody who can think should ever be forced into a situation that bores them.)

4. Freedom is good.

Hackers are naturally anti-authoritarian. Anyone who can give you orders can stop you from solving whatever problem you're being fascinated by - and, given the way authoritarian minds work, will generally find some appallingly stupid reason to do so. So the authoritarian attitude has to be fought wherever you find it, lest it smother you and other hackers. (This isn't the same as fighting all authority. Children need to be guided and criminals restrained. A hacker may agree to accept some kinds of authority in order to get something he wants more than the time he spends following orders. But that's a limited, conscious bargain; the kind of personal surrender authoritarians want is not on offer.)

Authoritarians thrive on censorship and secrecy. And they distrust voluntary cooperation and information-sharing â€” they only like 'cooperation' that they control. So to behave like a hacker, you have to develop an instinctive hostility to censorship, secrecy, and the use of force or deception to compel responsible adults. And you have to be willing to act on that belief.

5. Attitude is no substitute for competence.

To be a hacker, you have to develop some of these attitudes. But copping an attitude alone won't make you a hacker, any more than it will make you a champion athlete or a rock star. Becoming a hacker will take intelligence, practice, dedication, and hard work.

Therefore, you have to learn to distrust attitude and respect competence of every kind. Hackers won't let posers waste their time, but they worship competence - especially competence at hacking, but competence at anything is valued. Competence at demanding skills that few can master is especially good, and competence at demanding skills that involve mental acuteness, craft, and concentration is best.

If you revere competence, you'll enjoy developing it in yourself â€” the hard work and dedication will become a kind of intense play rather than drudgery. That attitude is vital to becoming a hacker.

Basic Hacking Skills

1. Learn how to program.
2. Get one of the open-source Unixes and learn to use and run it.
3. Learn how to use the World Wide Web and write HTML.
4. If you don't have functional English, learn it.

The hacker attitude is vital, but skills are even more vital. Attitude is no substitute for competence, and there's a certain basic toolkit of skills which you have to have before any hacker will dream of calling you one.

This toolkit changes slowly over time as technology creates new skills and makes old ones obsolete. For example, it used to include programming in machine language, and didn't until recently involve HTML. But right now it pretty clearly includes the following:

1. Learn how to program.

This, of course, is the fundamental hacking skill. If you don't know any computer languages, I recommend starting with Python. It is cleanly designed, well documented, and relatively kind to beginners. Despite being a good first language, it is not just a toy; it is very powerful and flexible and well suited for large projects. I have written a more detailed evaluation of Python. Good tutorials are available at the Python web site.

I used to recommend Java as a good language to learn early, but this critique has

changed my mind (search for “The Pitfalls of Java as a First Programming Language” within it). A hacker cannot, as they devastatingly put it “approach problem-solving like a plumber in a hardware store”; you have to know what the components actually do. Now I think it is probably best to learn C and Lisp first, then Java.

There is perhaps a more general point here. If a language does too much for you, it may be simultaneously a good tool for production and a bad one for learning. It's not only languages that have this problem; web application frameworks like RubyOnRails, CakePHP, Django may make it too easy to reach a superficial sort of understanding that will leave you without resources when you have to tackle a hard problem, or even just debug the solution to an easy one.

If you get into serious programming, you will have to learn C, the core language of Unix. C++ is very closely related to C; if you know one, learning the other will not be difficult. Neither language is a good one to try learning as your first, however. And, actually, the more you can avoid programming in C the more productive you will be.

C is very efficient, and very sparing of your machine's resources. Unfortunately, C gets that efficiency by requiring you to do a lot of low-level management of resources (like memory) by hand. All that low-level code is complex and bug-prone, and will soak up huge amounts of your time on debugging. With today's machines as powerful as they are, this is usually a bad tradeoff - it's smarter to use a language that uses the machine's time less efficiently, but your time much more efficiently. Thus, Python.

Other languages of particular importance to hackers include Perl and LISP. Perl is worth learning for practical reasons; it's very widely used for active web pages and system administration, so that even if you never write Perl you should learn to read it. Many people use Perl in the way I suggest you should use Python, to avoid C programming on jobs that don't require C's machine efficiency. You will need to be able to understand their code.

LISP is worth learning for a different reason - the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use LISP itself a lot. (You can get some beginning experience with LISP fairly easily by writing and modifying editing modes for the Emacs text editor, or Script-Fu plugins for the GIMP.)

It's best, actually, to learn all five of Python, C/C++, Java, Perl, and LISP. Besides being the most important hacking languages, they represent very different approaches to programming, and each will educate you in valuable ways.

But be aware that you won't reach the skill level of a hacker or even merely a programmer simply by accumulating languages - you need to learn how to think about programming problems in a general way, independent of any one language. To be a real hacker, you need to get to the point where you can learn a new language in days by relating what's in the manual to what you already know. This means you should learn several very different languages.

I can't give complete instructions on how to learn to program here - it's a complex skill. But I can tell you that books and courses won't do it - many, maybe most of the best hackers are self-taught. You can learn language features - bits of knowledge - from books, but the mind-set that makes that knowledge into living skill can be learned only by practice and apprenticeship. What will do it is (a) reading code and (b) writing code.

Peter Norvig, who is one of Google's top hackers and the co-author of the most widely used textbook on AI, has written an excellent essay called Teach Yourself Programming in Ten Years. His "recipe for programming success" is worth careful attention.

Learning to program is like learning to write good natural language. The best way to do it is to read some stuff written by masters of the form, write some things yourself, read a lot more, write a little more, read a lot more, write some more ... and repeat until your writing begins to develop the kind of strength and economy you see in your models.

Finding good code to read used to be hard, because there were few large programs available in source for fledgeling hackers to read and tinker with. This has changed dramatically; open-source software, programming tools, and operating systems (all built by hackers) are now widely available. Which brings me neatly to our next topic...

2. Get one of the open-source Unixes and learn to use and run it.

I'll assume you have a personal computer or can get access to one. (Take a moment to appreciate how much that means. The hacker culture originally evolved back when computers were so expensive that individuals could not own them.) The single most important step any newbie can take toward acquiring hacker skills is to get a copy of Linux or one of the BSD-Unixes or OpenSolaris, install it on a personal machine, and

run it.

Yes, there are other operating systems in the world besides Unix. But they're distributed in binary â€” you can't read the code, and you can't modify it. Trying to learn to hack on a Microsoft Windows machine or under any other closed-source system is like trying to learn to dance while wearing a body cast.

Under Mac OS X it's possible, but only part of the system is open source - you're likely to hit a lot of walls, and you have to be careful not to develop the bad habit of depending on Apple's proprietary code. If you concentrate on the Unix under the hood you can learn some useful things.

Unix is the operating system of the Internet. While you can learn to use the Internet without knowing Unix, you can't be an Internet hacker without understanding Unix. For this reason, the hacker culture today is pretty strongly Unix-centered. (This wasn't always true, and some old-time hackers still aren't happy about it, but the symbiosis between Unix and the Internet has become strong enough that even Microsoft's muscle doesn't seem able to seriously dent it.)

So, bring up a Unix - I like Linux myself but there are other ways (and yes, you can run both Linux and Microsoft Windows on the same machine). Learn it. Run it. Tinker with it. Talk to the Internet with it. Read the code. Modify the code. You'll get better programming tools (including C, LISP, Python, and Perl) than any Microsoft operating system can dream of hosting, you'll have fun, and you'll soak up more knowledge than you realize you're learning until you look back on it as a master hacker.

For more about learning Unix, see The Loginataka. You might also want to have a look at The Art Of Unix Programming.

To get your hands on a Linux, see the Linux Online! site; you can download from there or (better idea) find a local Linux user group to help you with installation.

During the first ten years of this HOWTO's life, I reported that from a new user's point of view, all Linux distributions are almost equivalent. But in 2006-2007, an actual best choice emerged: Ubuntu. While other distros have their own areas of strength, Ubuntu is

far and away the most accessible to Linux newbies.

You can find BSD Unix help and resources at www.bsd.org.

A good way to dip your toes in the water is to boot up what Linux fans call a live CD, a distribution that runs entirely off a CD without having to modify your hard disk. This will be slow, because CDs are slow, but it's a way to get a look at the possibilities without having to do anything drastic.

I have written a primer on the basics of Unix and the Internet.

I used to recommend against installing either Linux or BSD as a solo project if you're a newbie. Nowadays the installers have gotten good enough that doing it entirely on your own is possible, even for a newbie. Nevertheless, I still recommend making contact with your local Linux user's group and asking for help. It can't hurt, and may smooth the process.

3. Learn how to use the World Wide Web and write HTML.

Most of the things the hacker culture has built do their work out of sight, helping run factories and offices and universities without any obvious impact on how non-hackers live. The Web is the one big exception, the huge shiny hacker toy that even politicians admit has changed the world. For this reason alone (and a lot of other good ones as well) you need to learn how to work the Web.

This doesn't just mean learning how to drive a browser (anyone can do that), but learning how to write HTML, the Web's markup language. If you don't know how to program, writing HTML will teach you some mental habits that will help you learn. So build a home page. Try to stick to XHTML, which is a cleaner language than classic HTML. (There are good beginner tutorials on the Web; here's one.)

But just having a home page isn't anywhere near good enough to make you a hacker. The Web is full of home pages. Most of them are pointless, zero-content sludge - very snazzy-looking sludge, mind you, but sludge all the same (for more on this see The HTML Hell Page).

To be worthwhile, your page must have content - it must be interesting and/or useful to other hackers. And that brings us to the next topic...

4. If you don't have functional English, learn it.

As an American and native English-speaker myself, I have previously been reluctant to suggest this, lest it be taken as a sort of cultural imperialism. But several native speakers of other languages have urged me to point out that English is the working language of the hacker culture and the Internet, and that you will need to know it to function in the hacker community.

Back around 1991 I learned that many hackers who have English as a second language use it in technical discussions even when they share a birth tongue; it was reported to me at the time that English has a richer technical vocabulary than any other language and is therefore simply a better tool for the job. For similar reasons, translations of technical books written in English are often unsatisfactory (when they get done at all).

Linus Torvalds, a Finn, comments his code in English (it apparently never occurred to him to do otherwise). His fluency in English has been an important factor in his ability to recruit a worldwide community of developers for Linux. It's an example worth following.

Being a native English-speaker does not guarantee that you have language skills good enough to function as a hacker. If your writing is semi-literate, ungrammatical, and riddled with misspellings, many hackers (including myself) will tend to ignore you. While sloppy writing does not invariably mean sloppy thinking, we've generally found the correlation to be strong - and we have no use for sloppy thinkers. If you can't yet write competently, learn to.

Status in the Hacker Culture

1. Write open-source software
2. Help test and debug open-source software
3. Publish useful information
4. Help keep the infrastructure working

5. Serve the hacker culture itself

Like most cultures without a money economy, hackerdom runs on reputation. You're trying to solve interesting problems, but how interesting they are, and whether your solutions are really good, is something that only your technical peers or superiors are normally equipped to judge.

Accordingly, when you play the hacker game, you learn to keep score primarily by what other hackers think of your skill (this is why you aren't really a hacker until other hackers consistently call you one). This fact is obscured by the image of hacking as solitary work; also by a hacker-cultural taboo (gradually decaying since the late 1990s but still potent) against admitting that ego or external validation are involved in one's motivation at all.

Specifically, hackerdom is what anthropologists call a gift culture. You gain status and reputation in it not by dominating other people, nor by being beautiful, nor by having things other people want, but rather by giving things away. Specifically, by giving away your time, your creativity, and the results of your skill.

There are basically five kinds of things you can do to be respected by hackers:

1. Write open-source software

The first (the most central and most traditional) is to write programs that other hackers think are fun or useful, and give the program sources away to the whole hacker culture to use.

(We used to call these works “free software”, but this confused too many people who weren't sure exactly what “free” was supposed to mean. Most of us now prefer the term “open-source” software).

Hackerdom's most revered demigods are people who have written large, capable programs that met a widespread need and given them away, so that now everyone uses them.

But there's a bit of a fine historical point here. While hackers have always looked up to the open-source developers among them as our community's hardest core, before the mid-1990s most hackers most of the time worked on closed source. This was still true when I wrote the first version of this HOWTO in 1996; it took the mainstreaming of open-source software after 1997 to change things. Today, "the hacker community" and "open-source developers" are two descriptions for what is essentially the same culture and population - but it is worth remembering that this was not always so. (For more on this, see the section called “Historical Note: Hacking, Open Source, and Free Software”.)

2. Help test and debug open-source software

They also serve who stand and debug open-source software. In this imperfect world, we will inevitably spend most of our software development time in the debugging phase. That's why any open-source author who's thinking will tell you that good beta-testers (who know how to describe symptoms clearly, localize problems well, can tolerate bugs in a quickie release, and are willing to apply a few simple diagnostic routines) are worth their weight in rubies. Even one of these can make the difference between a debugging phase that's a protracted, exhausting nightmare and one that's merely a salutary nuisance.

If you're a newbie, try to find a program under development that you're interested in and be a good beta-tester. There's a natural progression from helping test programs to helping debug them to helping modify them. You'll learn a lot this way, and generate good karma with people who will help you later on.

3. Publish useful information

Another good thing is to collect and filter useful and interesting information into web pages or documents like Frequently Asked Questions (FAQ) lists, and make those generally available.

Maintainers of major technical FAQs get almost as much respect as open-source authors.

4. Help keep the infrastructure working

The hacker culture (and the engineering development of the Internet, for that matter) is run by volunteers. There's a lot of necessary but unglamorous work that needs done to

keep it going - administering mailing lists, moderating newsgroups, maintaining large software archive sites, developing RFCs and other technical standards.

People who do this sort of thing well get a lot of respect, because everybody knows these jobs are huge time sinks and not as much fun as playing with code. Doing them shows dedication.

5. Serve the hacker culture itself

Finally, you can serve and propagate the culture itself (by, for example, writing an accurate primer on how to become a hacker :-)). This is not something you'll be positioned to do until you've been around for while and become well-known for one of the first four things.

The hacker culture doesn't have leaders, exactly, but it does have culture heroes and tribal elders and historians and spokespeople. When you've been in the trenches long enough, you may grow into one of these. Beware: hackers distrust blatant ego in their tribal elders, so visibly reaching for this kind of fame is dangerous. Rather than striving for it, you have to sort of position yourself so it drops in your lap, and then be modest and gracious about your status.

The Hacker/Nerd Connection

Contrary to popular myth, you don't have to be a nerd to be a hacker. It does help, however, and many hackers are in fact nerds. Being something of a social outcast helps you stay concentrated on the really important things, like thinking and hacking.

For this reason, many hackers have adopted the label 'geek' as a badge of pride - it's a way of declaring their independence from normal social expectations (as well as a fondness for other things like science fiction and strategy games that often go with being a hacker). The term 'nerd' used to be used this way back in the 1990s, back when 'nerd' was a mild pejorative and 'geek' a rather harsher one; sometime after 2000 they switched places, at least in U.S. popular culture, and there is now even a significant geek-pride culture among people who aren't techies.

If you can manage to concentrate enough on hacking to be good at it and still have a life, that's fine. This is a lot easier today than it was when I was a newbie in the 1970s; mainstream culture is much friendlier to techno-nerds now. There are even growing numbers of people who realize that hackers are often high-quality lover and spouse material.

If you're attracted to hacking because you don't have a life, that's OK too - at least you won't have trouble concentrating. Maybe you'll get a life later on.

Points For Style

Again, to be a hacker, you have to enter the hacker mindset. There are some things you can do when you're not at a computer that seem to help. They're not substitutes for hacking (nothing is) but many hackers do them, and feel that they connect in some basic way with the essence of hacking.

- Learn to write your native language well. Though it's a common stereotype that programmers can't write, a surprising number of hackers (including all the most accomplished ones I know of) are very able writers.
- Read science fiction. Go to science fiction conventions (a good way to meet hackers and proto-hackers).
- Train in a martial-arts form. The kind of mental discipline required for martial arts seems to be similar in important ways to what hackers do. The most popular forms among hackers are definitely Asian empty-hand arts such as Tae Kwon Do, various forms of Karate, Kung Fu, Aikido, or Ju Jitsu. Western fencing and Asian sword arts also have visible followings. In places where it's legal, pistol shooting has been rising in popularity since the late 1990s. The most hackerly martial arts are those which emphasize mental discipline, relaxed awareness, and control, rather than raw strength, athleticism, or physical toughness.
- Study an actual meditation discipline. The perennial favorite among hackers is Zen (importantly, it is possible to benefit from Zen without acquiring a religion or discarding one you already have). Other styles may work as well, but be careful to choose one that doesn't require you to believe crazy things.
- Develop an analytical ear for music. Learn to appreciate peculiar kinds of music. Learn to play some musical instrument well, or how to sing.

- Develop your appreciation of puns and wordplay.

The more of these things you already do, the more likely it is that you are natural hacker material. Why these things in particular is not completely clear, but they're connected with a mix of left- and right-brain skills that seems to be important; hackers need to be able to both reason logically and step outside the apparent logic of a problem at a moment's notice.

Work as intensely as you play and play as intensely as you work. For true hackers, the boundaries between "play", "work", "science" and "art" all tend to disappear, or to merge into a high-level creative playfulness. Also, don't be content with a narrow range of skills. Though most hackers self-describe as programmers, they are very likely to be more than competent in several related skills - system administration, web design, and PC hardware troubleshooting are common ones. A hacker who's a system administrator, on the other hand, is likely to be quite skilled at script programming and web design. Hackers don't do things by halves; if they invest in a skill at all, they tend to get very good at it.

Finally, a few things not to do.

- Don't use a silly, grandiose user ID or screen name.
- Don't get in flame wars on Usenet (or anywhere else).
- Don't call yourself a 'cyberpunk', and don't waste your time on anybody who does.
- Don't post or email writing that's full of spelling errors and bad grammar.

The only reputation you'll make doing any of these things is as a twit. Hackers have long memories - it could take you years to live your early blunders down enough to be accepted.

The problem with screen names or handles deserves some amplification. Concealing your identity behind a handle is a juvenile and silly behavior characteristic of crackers, warez d00dz, and other lower life forms. Hackers don't do this; they're proud of what they do and want it associated with their real names. So if you have a handle, drop it. In the hacker culture it will only mark you as a loser.

Historical Note: Hacking, Open Source, and Free Software

When I originally wrote this how-to in late 1996, some of the conditions around it were very different from the way they look today. A few words about these changes may help clarify matters for people who are confused about the relationship of open source, free software, and Linux to the hacker community. If you are not curious about this, you can skip straight to the FAQ and bibliography from here.

The hacker ethos and community as I have described it here long predates the emergence of Linux after 1990; I first became involved with it around 1976, and, its roots are readily traceable back to the early 1960s. But before Linux, most hacking was done on either proprietary operating systems or a handful of quasi-experimental homegrown systems like MIT's ITS that were never deployed outside of their original academic niches. While there had been some earlier (pre-Linux) attempts to change this situation, their impact was at best very marginal and confined to communities of dedicated true believers which were tiny minorities even within the hacker community, let alone with respect to the larger world of software in general.

What is now called "open source" goes back as far as the hacker community does, but until 1985 it was an unnamed folk practice rather than a conscious movement with theories and manifestos attached to it. This prehistory ended when, in 1985, arch-hacker Richard Stallman ("RMS") tried to give it a name - "free software". But his act of naming was also an act of claiming; he attached ideological baggage to the "free software" label which much of the existing hacker community never accepted. As a result, the "free software" label was loudly rejected by a substantial minority of the hacker community (especially among those associated with BSD Unix), and used with serious but silent reservations by a majority of the remainder (including myself).

Despite these reservations, RMS's claim to define and lead the hacker community under the "free software" banner broadly held until the mid-1990s. It was seriously challenged only by the rise of Linux. Linux gave open-source development a natural home. Many projects issued under terms we would now call open-source migrated from proprietary Unixes to Linux. The community around Linux grew explosively, becoming far larger and more heterogenous than the pre-Linux hacker culture. RMS determinedly attempted to co-opt all this activity into his "free software" movement, but was thwarted by both the exploding diversity of the Linux community and the public skepticism of its founder, Linus Torvalds. Torvalds continued to use the term "free software" for lack of any alternative, but publicly rejected RMS's ideological baggage. Many younger hackers

followed suit.

In 1996, when I first published this Hacker HOWTO, the hacker community was rapidly reorganizing around Linux and a handful of other open-source operating systems (notably those descended from BSD Unix). Community memory of the fact that most of us had spent decades developing closed-source software on closed-source operating systems had not yet begun to fade, but that fact was already beginning to seem like part of a dead past; hackers were, increasingly, defining themselves as hackers by their attachments to open-source projects such as Linux or Apache.

The term "open source", however, had not yet emerged; it would not do so until early 1998. When it did, most of hacker community adopted it within the following six months; the exceptions were a minority ideologically attached to the term "free software". Since 1998, and especially after about 2003, the identification of 'hacking' with 'open-source (and free software) development' has become extremely close. Today there is little point in attempting to distinguish between these categories, and it seems unlikely that will change in the future.

It is worth remembering, however, that this was not always so.

Other Resources

Paul Graham has written an essay called Great Hackers, and another on Undergraduation, in which he speaks much wisdom.

There is a document called How To Be A Programmer that is an excellent complement to this one. It has valuable advice not just about coding and skillsets, but about how to function on a programming team.

I have also written A Brief History Of Hackerdom.

I have written a paper, The Cathedral and the Bazaar, which explains a lot about how the Linux and open-source cultures work. I have addressed this topic even more directly in its sequel Homesteading the Noosphere.

Rick Moen has written an excellent document on how to run a Linux user group.

Rick Moen and I have collaborated on another document on How To Ask Smart Questions. This will help you seek assistance in a way that makes it more likely that you will actually get it.

If you need instruction in the basics of how personal computers, Unix, and the Internet work, see The Unix and Internet Fundamentals HOWTO.

When you release software or write patches for software, try to follow the guidelines in the Software Release Practice HOWTO.

If you enjoyed the Zen poem, you might also like Rootless Root: The Unix Koans of Master Foo.

Unit 5 Introduction to Open Source Movement

The open source movement is a broad-reaching movement comprised both officially and unofficially of individuals who feel that software should be produced altruistically[citation needed]. Open source software is made available to anybody to use, or to modify as its source code is made available. The freedom of the software is subject only to the stipulation that the user of the software makes any enhancements or changes as freely available to the public. Open source software promotes learning and understanding through the dissemination of understanding.

The main difference between open source and traditional proprietary software is in the terms of user and property rights. These rights reside in the conditions of use that are imposed on the user by the license of the software, as opposed to a difference in the programming code. With open source software, users are granted the right to both the program's functionality and methodology. With proprietary software programs, such as Microsoft Office, users only have the rights to functionality.[1]

Libraries are using open source software to develop information as well as library services. The purpose of open source is to provide a software that is cheaper, reliable and has better quality. The one feature that makes this software so sought after is that it is free. Libraries in particular benefit from this movement because of the resources it provides. They also promote the same ideas of learning and understanding new information through the resources of other people. Open source allows a sense of community. It is an invitation for anyone to provide information about various topics. The open source tools even allow libraries to create web-based catalogs. According to the IT source there are various library programs that benefit from this. [2]

Programmers who support the open source movement philosophy contribute to the open source community by voluntarily writing and exchanging programming code for software development.[3] The term “open source” requires that no one can discriminate against a group in not sharing the edited code or hinder others from editing their already edited work. This approach to software development allows anyone to obtain and modify open source code. These modifications are distributed back to the developers within the open source community of people who are working with the software. In this way, the identities of all individuals participating in code modification are disclosed and the transformation of the code is documented over time.[4] This method makes it difficult to establish ownership of a particular bit of code but is in keeping with the open

source movement philosophy. These goals promote the production of “high quality programs” as well as “working cooperatively with other similarly minded people” to improve open source technologies.[3]

Brief History

In the late 1970s and early 1980s, two different groups were establishing the roots of the current open source software movement. On the east coast, Richard Stallman, formerly of the MIT AI lab, created the GNU project and the Free Software Foundation.[5] The GNU was aimed to create a free operating system. The GNU General Public License (GPL) was one of the open source licenses that served as a prohibitory of control over software codes. This specific license allowed users to not only modify, but also redistribute people’s own versions of the software. This not only allows, but also requires that anyone operating under the Linux GPL agree to the terms of the original kernel and makes the edit available to everyone. On the US West coast, the Computer Science Research Group (CSRG) of the University of California at Berkeley was improving the Unix system, and developing many applications which quickly became "BSD Unix". These efforts were funded mainly by DARPA contracts, and a dense network of Unix hackers around the world helped to debug, maintain and improve the system.[6] During 1991-1992, the whole landscape of open source software, and of software development in general, was ready to change. Two very exciting events were taking place, although in different communities: In California, Bill Jolitz was implementing the missing portions to complete the Net/2 distribution, until it was ready to run on i386-class machines. Net/2 was the result of the effort of the CSRG to make an unencumbered version of BSD Unix (free of AT&T copyrighted code). He called his work 386BSD, and it quickly became appreciated within the BSD and Unix communities. It included not only a kernel, but also many utilities, making a complete operating system.[6] In Finland, Linus Torvalds, a student of computer science, unhappy with Tanenbaum's Minix, was implementing the first versions of the Linux kernel. Soon, many people were collaborating to make that kernel more and more usable, and adding many utilities to complete GNU/Linux, a real operating system. The Linux kernel, and the GNU applications used on top of it are covered by GPL.[6] In 1993, both GNU/Linux and 386BSD were reasonably stable platforms. Since then, 386BSD has evolved into a family of BSD based operating systems (NetBSD, FreeBSD, and OpenBSD), while the Linux kernel is healthy evolving and being used in many GNU/Linux distributions (Slackware, Debian, Red Hat, Suse, Mandrake, and many more).[6] Stallman coined the term “copyleft” for these types of licenses. It is the copyright of the GPL and rather than taking away freedom, gives the freedom to change the software.

Evolution

Any technological advance needs a reason to be introduced into society. In the beginning, a difference between hardware and software did not exist. The user and programmer of a computer were one and the same. When the first commercial electronic computer was introduced by IBM in 1952, the machine was hard to maintain and expensive. Putting the price of the machine aside it was the software that caused the problem when owning one of these computers. Then in 1952, a collaboration of all the owners of the computer got together and created a set of tools. The collaboration of people were in a group called PACT (The Project for the Advancement of Coding techniques). After passing this hurdle, in 1956, the Eisenhower administration decided to put restrictions on the types of sales AT&T could make. This did not stop the inventors from developing new ideas of how to bring the computer to the mass population. The next step was making the computer more affordable which slowly developed through different companies. Then they had to develop a software which would host multiple users. MIT computation center developed one of the first systems, CTSS (Compatible Time-Sharing System). This lay the foundation for many more systems to come and what we now call the Open Source Movement.

The Open Source Movement is branched from the free software movement which began in the late 80s with the launching of the GNU/Linux project by Richard Stallman.[4] Stallman is regarded within the open source community as sharing a key role in the conceptualization of freely shared source code for software development.[4] The term “free software” in the free software movement is meant to imply freedom of software exchange and modification. The term does not refer to any monetary freedom.[4] Both the free software movement and the open source movement share this view of free exchange of programming code, and this is often why both of the movements are sometimes referenced in literature as part of the FOSS or “Free and Open Software” or FLOSS “Free/Libre Open Source” communities.

These movements share fundamental differences in the view on open software. The main, factionalizing difference between the groups is the relationship between open source and propriety software. Often makers of proprietary software, such as Microsoft, may make efforts to support open source software to remain competitive.[8] Members of the open source community are willing to coexist with the makers of propriety software[4] and feel that the issue of whether software is open source is a matter of practicality.[4]

In contrast, members of the free software community maintain the vision that all software is a part of freedom of speech[4] and that proprietary software is unethical and unjust.[4] The free software movement openly champions this belief through talks that denounce propriety software. As a whole the community refuses to support propriety software. It also is suggested there are external motivations exist for these developers. One motivation is when a programmer fixes a bug or makes a program it benefits others in an open source environment. Another motivation is that a programmer can work on multiple projects at the same time doing something they enjoy. Also, programming in the open source world can lead to commercial job offers or entrance into the venture capital community. These are just a few reasons why open source programmers continue to create and advance.[9]

While cognizant of the fact that both it and the open source movement share similarities in practical recommendations regarding open source, the free software movement fervently continues to distinguish themselves from the open source movement entirely.[4] The free software movement maintains that it has fundamentally different attitudes towards the relationship between open source and propriety software. The free software community does not view the open source community as their target grievance, however. Their target grievance is propriety software itself.[4]

Legal Issues

The Open Source Movement has faced a number of legal challenges. Companies that manage open source products have some difficulty securing their trademarks. For example, the scope of “implied license” conjecture remains unclear and can compromise an enterprise’s ability to patent productions made with open source software. Another example is the case of companies offering add-ons for purchase; licensees who make additions to the open-source code that are similar to those for purchase may have immunity from patent suits.

In the court case "Jacobsen v Katzer", the plaintiff sued the defendant for failing to put the required attribution notices in his modified version of the software, thereby violating license. The defendant claimed Artistic License in not adhering to the conditions of the software’s use, but the wording of the attribution notice decided that this was not the case. "Jacobsen v Katzer" established open source software’s equality to for-profit software in the eyes of the law.

In a court case accusing Microsoft of being a monopoly, Linux and open source software

was introduced in court to prove that Microsoft had valid competitors and was grouped in with Apple.

There are resources available for those involved open source projects in need of legal advice. The Software Freedom Legal Center features a primer on open source legal issues. International Free and Open Source Software Law Review offers peer-reviewed information for lawyers on free software issues.

Formalization

In February 1998 the open source movement was adopted, formalized, and spearheaded by the Open Source Initiative (OSI), an organization formed to market software “as something more amenable to commercial business use”[4] The OSI owns the trademark “Open Source”[3] The main tool they adopted for this was the Open Source Definition[10]

Overall, the software developments that have come out of the open source movement have not been unique to the computer science field, but they have been successful in developing alternatives to propriety software. Members of the open source community improve upon code and write programs that can rival much of the propriety software that is already available.[4]

Examples of software that have come out of the Open Source Movement

- Linux – a Unix-Based operating system used predominantly in servers[3] Linux was created by a student in 1991 along with other developers around the world.[11]
- Apache — a leading server software and scripting language on the web[12]
- MySQL — a database management system[13]
- PHP — a widely used open source general-purpose scripting language[14]
- Blender — a 3D graphics and animation software[13]
- OpenOffice.org – an office suite software with word processor, spreadsheet, and presentation capabilities[13]
- Mozilla — a web browser and e-mail client[13]
- Perl — a programming/scripting language[13]
- Wikipedia — Online encyclopedia open for anyone to update and revise content.[11]

Strengths

- The collaborative nature of the open source community creates software that can offer customizability and, as a result, promotes the adoption of its products.[15]
- The open source community promotes the creation of software that is not proprietary, thus resulting in lower costs.[15]
- The development of open source software within the community is motivated by the individual who has expressed interest in the code and software creation. This differs from proprietary software that is often motivated via monetary means.[15]
- An open source tool puts the system administrator in control of the level of risk assumed in deploying the tool.[16]
- Open source provides a flexibility not available in closed products. The hope is that If you make improvements to an open tool you'll offer them back to the original developer and community at large. The give-and-take of the gift economy benefits everyone.[16]

Members of the Open Source Movement stress the importance of differentiating between “open source” software and “free software”. Although the two issues are related, they are quite different. Although the groups agree on the overall practical recommendations, they disagree on the basic principles. A major advantage to open source code is the ability for a variety of different people to edit and fix problems and errors that have occurred. Naturally because there are more people who can edit the material there are more people who can help make the information more credible and reliable. The open source mission statement promises better quality, higher reliability, more flexibility, lower cost, and an end to predatory vendor lock-in. They stress the importance of maintaining the Open Source Definition. This trademark creates a trusted group that connects all users and developers together.[17] To fully understand the Open Source Definition, one must understand certain terms: Free redistribution means that there is no restriction on any party to sell or give away the software to third parties. Source Code means that the program must efficiently publicize the means of obtaining the source code. Derived works means that the program must allow certain works to be distributed under the same terms. There must be a promise of no discriminating against any certain persons or groups. All of these factors allow for the open source movement to become available to all and easy to access, which is there overall mission. The latest updates from the Open Source Institution took place on January 19, 2011: The OSI collaborated with the Free Software Foundation and together they updated a version of the request that they have sent to the US Department of Justice.[18]

Drawbacks

- The structure of the open source community requires that individuals have programming expertise in order to engage in open code modification and exchange. Individuals interested in supporting the open source movement may lack this skill set.[4]
- Programmers and developers comprise a large percentage of the open source community and sought-out technical support may not be useful or clear to open source software lay-users.[15]
- The structure of the open source community is one which involves contributions of multiple developers and programmers; software produced in this fashion may lack standardization and be incompatible with various computer applications and capabilities.[15]
- Production can be very limited. Programmers that create open source software often can turn their attention elsewhere very quickly. This opens the door for many bug filled programs and applications out there. Because no one is being paid to create it, many projects do not get finished.[19]
- The quality of the software in an open source industry is decided by the user. A user has to learn the skills of the software on their own and then make the determination.[20]
- Librarians may not be equip to take on this new responsibility of technologies.[21]
- Beyond the obvious detriments towards the theoretical success of open source software, there are several factors that contribute to the lack of long-term success in open source projects. One of the most obvious drawback is that without pay or royalty licensing, there is little financial incentive for a programmer to become involved with a project in the first place, or to continue development and support once the initial product is released. This leads to innumerable examples of well-anticipated software being forever condemned to beta versions and unsupported early model products. With donations as the only source of income for a truly open source (and GPL licensed) project, there is almost no certainty in the future of the project simply because of developer abandonment, making it a poor choice for any sort of application in which future versions, support and a long-term plan would be essential, as is the case for most business software.[22]

Evidence of Open Source Adoption

The following are events and applications that have been developed via the open source community as and echo the ideologies of the open source movement.[4]

OpenCourseWare Consortium — organization composed various colleges that supported

the open source software. This organization was headed by Massachusetts Institute of Technology and was established to aid in the exchange of open source educational materials.[4]

Wikipedia — user generated online encyclopedia that had branched into many other academic areas such as Wikiversity — a community dedicated to the creation and exchange of learning materials[4]

Project Gutenberg — prior to the exist of Google Scholar Beta, this was the first supplier of electronic books and the very first free library project[4]

Google — this search engine has led the way in transformation of Web-based applications, such as books, scholarly journals, that are based primarily on open source software.[4] Google continues to make applications based on open software. Recently, in November 2009, Google announced that it would be “enabling people everywhere to find, and read full text legal opinions from U.S. federal and state districts, appellate and supreme courts using Google Scholar”[8]

Government agencies and infrastructure software — Government Agencies are utilizing open source infrastructure software, like the Linux operating system and the Apache Web-server into software, to manage information.[8]

Synthetic Biology- Synthetic Biology is considered the feasibility of the open source movement. This new technology is important and exciting because it promises to enable cheap, lifesaving new drugs as well as helping to yield biofuels that may help to solve our energy problem. Although synthetic biology has not yet come out of its "lab" stage, it has great potential to become industrialized in the near future. In order to industrialize open source science, there are some scientists who are trying to build their own brand of it.[23]

Open Source Movement in the Military- Open source movement has potential to help in the military. The open source software allows anyone to make changes that will improve it. This is a form of invitation for people to put their minds together to grow a software in a cost efficient manner. The reason the military is so interested is because it is possible that this software can increase speed and flexibility. Although there are security setbacks to this idea due to the fact that anyone has access to change the software, the

advantages can outweigh the disadvantages. The fact that the open- source programs can be modified quickly is crucial. A support group was formed to test these theories. It was called The Military Open Source Software Working Group, was organized in 2009 and held over 120 military members. Their purpose was to bring together software developers and contractors from the military to discover new ideas for reuse and collaboration. Overall, open-source software in the military is an intriguing idea that has potential drawbacks but they are not enough to offset the advantages.[24]

Open Source in Education- Colleges and organizations use software predominantly online to educate their students. Open source technology is being adopted by many institutions because it can save these institutions from paying companies to provide them with these administrative software systems. One of the first major colleges to adopt an open source system was Colorado State University in 2009 with many others following after that. Colorado State Universities system was produced by the Kuali Foundation who has become a major player in open source administrative systems. The Kuali Foundation defines itself as a group of organizations that aims to "build and sustain open source software for higher education, by higher education." There are many other examples of open source instruments being used in education other than the Kuali Foundation as well.[25]

Ideologically Related Movements

The Open access movement is a movement that is similar in ideology to the open source movement. Members of this movement maintain that academic material should be readily available to provide help with “future research, assist in teaching and aid in academic purposes.” The Open access movement aims to eliminate subscription fees and licensing restrictions of academic materials[8]

The Free Culture Movement is a movement that seeks to achieve a culture in that engages in collective freedom via freedom of expression, free public access to knowledge and information, full demonstration of creativity and innovation in various arenas and promotion of citizen liberties.[26]

References

1. ^ Bradley, D.A. (2005). The divergent anarcho-utopian discourses of the open source software movement. Canadian Journal of Communication, 30, 585-611.
2. ^ Poynder, Richard. (2001). The Open Source Movement. Information Today, volume 8, (issue 9). Retrieved from

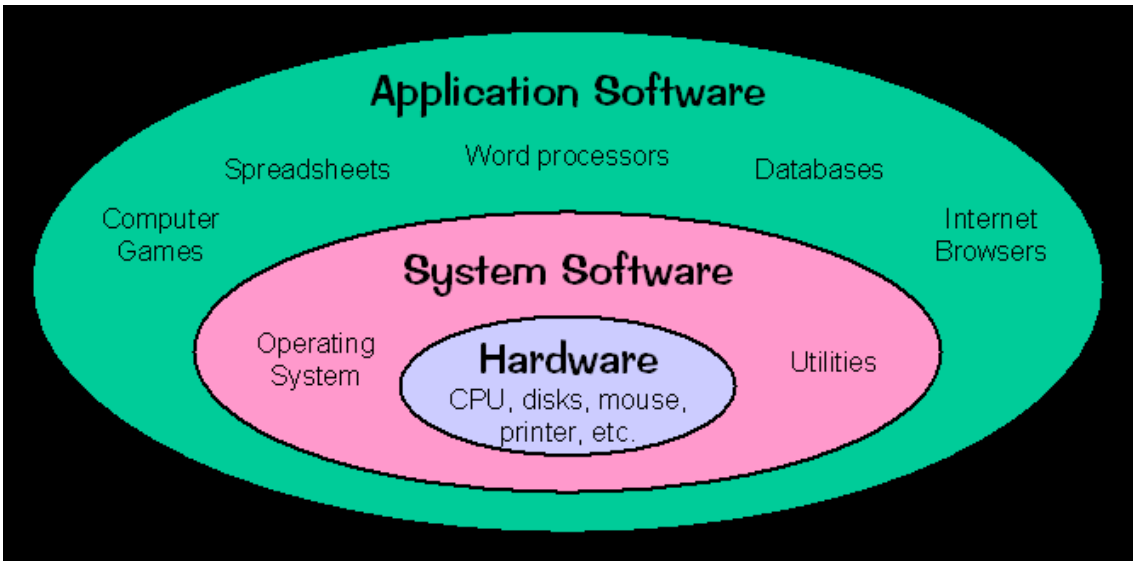
http://www.infotoday.com/it/oct01/poynder.htm
3. ^ a b c d Wyllys, R.E. (2000). Overview of the Open-Source Movement. Retrieved November 22, 2009, from The University of Texas at Austin Graduate School of Library & Information Science: http://www.ischool.utexas.edu/~l38613dw/readings/Open
4. ^ a b c d e f g h i j k l m n o p q r Warger, T. (2002). The Open Source Movement. Retrieved November 22, 2009, from Education Resources Information Center: http://net.educause.edu/ir/library/pdf/eqm0233.pdf
5. ^ Richard Stallman. The GNU Project. In Chris DiBona, Sam Ockman, and Mark Stone, editors, Open Sources. Voices from the Open Source Revolution. O'Reilly & Associates, 1999
6. ^ a b c d http://eu.conecta.it/paper/brief_history_open_source.html
7. ^ Weber, Steven. The Success of Open Source. The President and Fellows of Harvard College. 2004. Print pg.20-28. This whole paragraph is referenced to Steven Weber
8. ^ a b c d Taft, D. K. (2009, November 3). Microsoft Recommits to $100k Apache Contribution at ApacheCon. Retrieved November 22, 2009, from eWeek: http://www.eweek.com/c/a/Linux-and-Open-Source/Microsoft-Recommits-100K-Apache-Contribution-at-
9. ^ Lerner, John and Tirole, Jean. "The simple Economics of Open Source". National Bureau of Economic Research. Cambridge, MA. March 2000: [1]
10. ^ http://www.opensource.org/docs/osd
11. ^ a b Unknown. (2007). How the Open Source Movement has Changed Education: 10 Success stories. Online Education Database. Retrieved from http://oedb.org/library/features/how-the-open-source-movement-has-changed-education-10-success-stories
12. ^ Langley, N. (2007). Apache is the big chief in the world of web servers. Computer Weekly, 34. Retrieved from Business Source Premier database.
13. ^ a b c d e Metcalfe, R. (200p, October 13). Examples of Open Source Software. Retrieved November 22, 2009, from OSS Watch:
14. ^ The PHP Group. (2009, November 20). What is PHP? Retrieved November 22, 2009, from PHP: http://php.net/manual/en/intro-whatis.php
15. ^ a b c d e Webb, M. (2001, July 18). Going With Open Source Software. Retrieved November 22, 2009, from techsoup: http://www.techsoup.org/learningcenter/software/archives/page9905.cfm
16. ^ a b http://www.albion.com/security/intro-7.html
17. ^ Poynder, R. (n.d.). IT Feature: The Open Source Movement. Information Today, Inc.. Retrieved January 25, 2011, from http://www.infotoday.com/it/oct01/
18. ^ Wyllys, R. (n.d.). Overview of the Open-Source Movement. UT School of

Information - Home Page. Retrieved January 25, 2011, from http://www.ischool.utexas.edu/~l38613dw/readings/OpenSource
19.^ http://www.softwarecompany.org/advantages-open-source-software.html
20.^ Golden, Bernard. Succeeding with Open Source. Pearson Education. 2005
21.^ Poynder, Richard. (2001). The Open Source Movement. Information Today, volume 8, (issue 9). Retrieved from http://www.infotoday.com/it/oct01/poynder.htm
22.^ http://www.techsoup.org/learningcenter/software/archives/page9905.cfm
23.^ Wilson Center.(2009). Synthetic Biology: Feasibility of the Open Source Movement. Wislson On Demand Center. Retrieved from http://www.wilsoncenter.org/ondemand/index.cfm?fuseaction=home.play&mediaid=09AE937D-D3F4-4501-4BBEF8D8F8EED0CD
24.^ Toon, John. (2009). Open Source Movement May Accelerate Military Software Development. Georgia Tech Research Institute. Retrieved from http://www.gtri.gatech.edu/casestudy/open-source-mil-oss-military-software
25.^ http://www.kuali.org/about/
26.^ Students For Free Cultue. (2009). Main Page. Retrieved November 22, 2009, from free culture.org: http://freeculture.org/

Unit 6 Introduction to Operating Systems

Computer software can be divided into two main categories: application software and system software. According to Brookshear [1997], "application software consists of the programs for performing tasks particular to the machine's utilization. Examples of application software include spreadsheets, database systems, desktop publishing systems, program development software, and games." Application software is generally what we think of when someone speaks of computer programs. This software is designed to solve a particular problem for users.

On the other hand, system software is more transparent and less noticed by the typical computer user. This software "provides a general programming environment in which programmers can create specific applications to suit their needs. This environment provides new functions that are not available at the hardware level and performs tasks related to executing the application program" [Nutt 1997]. System software acts as an interface between the hardware of the computer and the application software that users need to run on the computer. The diagram below illustrates the relationship between application software and system software.



The most important type of system software is the operating system. According to Webopedia [2000], an operating system has three main responsibilities:

1. Perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.

2. Ensure that different programs and users running at the same time do not interfere with each other.

3. Provide a software platform on top of which other programs (i.e., application software) can run.

The first two responsibilities address the need for managing the computer hardware and the application programs that use the hardware. The third responsibility focuses on providing an interface between application software and hardware so that application software can be efficiently developed. Since the operating system is already responsible for managing the hardware, it should provide a programming interface for application developers.

Nutt [1997] identifies four common types of operating system strategies on which modern operating systems are built: batch, timesharing, personal computing, and dedicated. According to Nutt, "the favored strategy for any given computer depends on how the computer is to be used, the cost-effectiveness of the strategy implementation in the application environment, and the general state of the technology at the time the operating system is developed." The table below summarizes the characteristics of each operating system strategy as described by Nutt [1997].

Batch: This strategy involves reading a series of jobs (called a batch) into the machine and then executing the programs for each job in the batch. This approach does not allow users to interact with programs while they operate.

Timesharing: This strategy supports multiple interactive users. Rather than preparing a job for execution ahead of time, users establish an interactive session with the computer and then provide commands, programs and data as they are needed during the session.

Personal Computing: This strategy supports a single user running multiple programs on a dedicated machine. Since only one person is using the machine, more attention is given to establishing predictable response times from the system. This strategy is quite common today because of the popularity of personal computers.

Dedicated: This strategy supports real-time and process control systems. These are the

types of systems which control satellites, robots, and air-traffic control. The dedicated strategy must guarantee certain response times for particular computing tasks or the application is useless.

By the end of this section, you should be able to do the following:

- Understand the purpose of the operating system,
- Distinguish between a resource, a program, and a process,
- Recognize critical resources and explain the behavior of semaphores,
- Describe various memory page replacement algorithms, and
- Describe how files are stored in secondary storage.

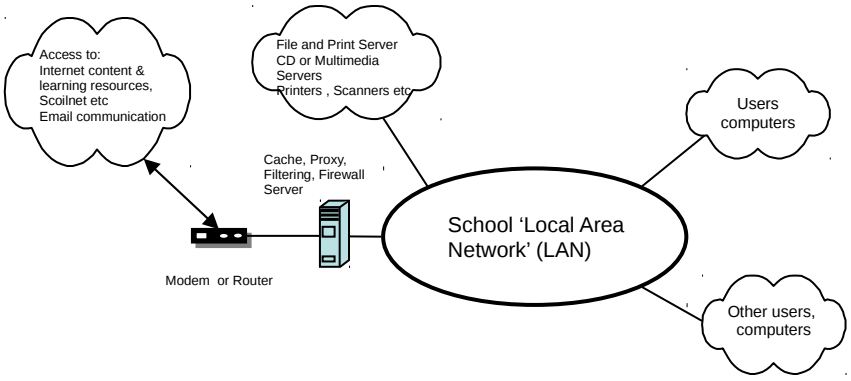
References

- Brookshear, J. G. (1997), Computer Science: An Overview, Fifth Edition, Addison-Wesley, Reading, MA.
- Nutt, G. (1997), Operating Systems: A Modern Perspective, First Edition, Addison-Wesley, Reading, MA.
- Webopedia (2000), "Online Computer Dictionary," http://webopedia.internet.com/TERM/o/operating_system.html.

Unit 7 Introduction to Computer Networking

Basic of Networking

A computer network consists of a collection of computers, printers and other equipment that is connected together so that they can communicate with each other (see Advice Sheet 17 on the ICT Planning for schools pack). Fig 1 gives an example of a network in a school comprising of a local area network or LAN connecting computers with each other, the internet, and various servers.



Broadly speaking, there are two types of network configuration, peer-to-peer networks and client/server networks.

Peer-to-peer networks are more commonly implemented where less then ten computers are involved and where strict security is not necessary. All computers have the same status, hence the term 'peer', and they communicate with each other on an equal footing. Files, such as word processing or spreadsheet documents, can be shared across the network and all the computers on the network can share devices, such as printers or scanners, which are connected to any one computer.

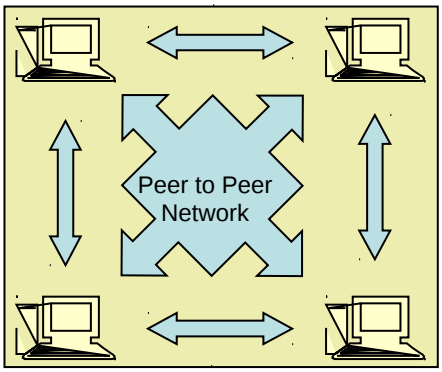


Fig 2: Peer to Peer Networking

Client/server networks are more suitable for larger networks. A central computer, or 'server', acts as the storage location for files and applications shared on the network. Usually the server is a higher than average performance computer. The server also controls the network access of the other computers which are referred to as the 'client' computers. Typically, teachers and students in a school will use the client computers for their work and only the network administrator (usually a designated staff member) will have access rights to the server.

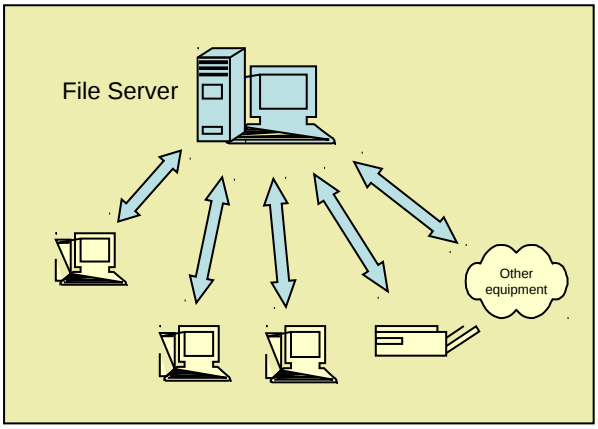


Fig 3: Client – Server Networking

Table 1 provides a summary comparison between Peer-to-Peer and Client/Server Networks.

Peer-to-Peer Networks vs Client/Server Networks	
Peer-to-Peer Networks	Client/Server Networks
· Easy to set up	· More difficult to set up
· Less expensive to install	· More expensive to install

· Can be implemented on a wide range of operating systems	· A variety of operating systems can be supported on the client computers, but the server needs to run an operating system that supports networking
· More time consuming to maintain the software being used (as computers must be managed individually)	· Less time consuming to maintain the software being used (as most of the maintenance is managed from the server)
· Very low levels of security supported or none at all. These can be very cumbersome to set up, depending on the operating system being used	· High levels of security are supported, all of which are controlled from the server. Such measures prevent the deletion of essential system files or the changing of settings
· Ideal for networks with less than 10 computers	· No limit to the number of computers that can be supported by the network
· Does not require a server	· Requires a server running a server operating system
· Demands a moderate level of skill to administer the network	· Demands that the network administrator has a high level of IT skills with a good working knowledge of a server operating system

Table 1: Peer-to-Peer Networks vs Client/Server Networks

Components of a Network

A computer network comprises the following components:

- A minimum of at least 2 computers
- Cables that connect the computers to each other, although wireless communication is becoming more common (see Advice Sheet 20 for more information)
- A network interface device on each computer (this is called a network interface card or NIC)
- A ‘Switch’ used to switch the data from one point to another. Hubs are outdated and are little used for new installations.
- Network operating system software

Structured Cabling

The two most popular types of structured network cabling are twisted-pair (also known as 10BaseT) and thin coax (also known as 10Base2). 10BaseT cabling looks like ordinary telephone wire, except that it has 8 wires inside instead of 4. Thin coax looks like the copper coaxial cabling that's often used to connect a Video Recorder to a TV.

10BaseT Cabling

When 10BaseT cabling is used, a strand of cabling is inserted between each computer and a hub. If you have 5 computers, you'll need 5 cables. Each cable cannot exceed 325 feet in length. Because the cables from all of the PCs converge at a common point, a 10BaseT network forms a star configuration.

Fig 4a shows a Cat5e cable, with a standard connector, known as an RJ-45 connector.

Fig 4b shows a standard Cat5e Wall Outlet socket which the cables are connected to.

Fig 4c shows a standard Cat5e Patch Panel Wall Outlet socket which is used to terminate the cables from various points in the school bank to a central point.

Fig 4d shows a wall mounted cabinet used to house and protect patch panel cables and connectors.

Fig 4a: Cat5e Cable and a close up of RJ-45 connector

Fig 4b: Cat5e Wall Outlets

Fig 4c: Cat5e Patch Panel

Fig4d: Wall Mounted Cabinet

10BaseT cabling is available in different grades or categories. Some grades, or "cats", are required for Fast Ethernet networks, while others are perfectly acceptable for standard 10Mbps networks--and less expensive, too. All new networks use a minimum of standard unshielded twisted-pair (UTP) Category 5e 10BaseT cabling because it offers a performance advantage over lower grades.

Network Interface Card (NIC)

A NIC (pronounced 'nick') is also known as a network card. It connects the computer to the cabling, which in turn links all of the computers on the network together. Each computer on a network must have a network card. Most modern network cards are 10/100 NICs and can operate at either 10Mbps or 100Mbps.

Only NICs supporting a minimum of 100Mbps should be used in new installations schools.

Computers with a wireless connection to a network also use a network card (see Advice Sheet 20 for more information on wireless networking).

Fig 5: Network Interface Cards (NICs)

Hub and Switch

A hub is a device used to connect a PC to the network. The function of a hub is to direct information around the network, facilitating communication between all connected devices. However in new installations switches should be used instead of hubs as they are more effective and provide better performance. A switch, which is often termed a 'smart hub'.

Switches and hubs are technologies or ‘boxes’ to which computers, printers, and other networking devices are connected. Switches are the more recent technology and the accepted way of building today's networks. With switching, each connection gets "dedicated bandwidth" and can operate at full speed. In contrast, a hub shares bandwidth

across multiple connections such that activity from one PC or server can slow down the effective speed of other connections on the hub.

Now more affordable than ever, Dual-speed 10/100 autosensing switches are recommended for all school networks. Schools may want to consider upgrading any hub based networks with switches to improve network performance – ie speed of data on the network.

Fig 6a: An 8 port Hub

Fig 6b: 2 Examples of 24 port Switches

Wireless Networks

The term 'wireless network' refers to two or more computers communicating using standard network rules or protocols, but without the use of cabling to connect the computers together. Instead, the computers use wireless radio signals to send information from one to the other. A wireless local area network (WLAN) consists of two key components: an access point (also called a base station) and a wireless card. Information can be transmitted between these two components as long as they are fairly close together (up to 100 metres indoors or 350 metres outdoors).

Fig 7a: Wireless Access point or Wireless Basestation

Suppliers would need to visit the schools and conduct a site survey. This will determine the number of base stations you need and the best place(s) to locate them. A site survey will also enable each supplier to provide you with a detailed quote. It is important to contact a number of different suppliers as prices, equipment and opinions may vary. When the term 'wireless network' is used today, it usually refers to a wireless local area network or WLAN. A WLAN can be installed as the sole network in a school or building. However, it can also be used to extend an existing wired network to areas where wiring would be too difficult or too expensive to implement, or to areas located away from the main network or main building. Wireless networks can be configured to provide the same network functionality as wired networks, ranging from simple peer-to-peer configurations to large-scale networks accommodating hundreds of users.

Fig 7b: Desktop PC Wireless LAN card

Fig 7c: Laptop PC Wireless LAN card

What are the advantages and disadvantages of a Wireless LAN?

Wireless LANs have advantages and disadvantages when compared with wired LANs. A wireless LAN will make it simple to add or move workstations, and to install access points to provide connectivity in areas where it is difficult to lay cable. Temporary or semi-permanent buildings that are in range of an access point can be wirelessly connected to a LAN to give these buildings connectivity. Where computer labs are used in schools, the computers (laptops) could be put on a mobile cart and wheeled from classroom to classroom, providing they are in range of access points. Wired network points would be needed for each of the access points.

A WLAN has some specific advantages:

- It is easier to add or move workstations
- It is easier to provide connectivity in areas where it is difficult to lay cable
- Installation can be fast and easy and can eliminate the need to pull cable through walls and ceilings
- Access to the network can be from anywhere in the school within range of an access point

- Portable or semi-permanent buildings can be connected using a wireless LAN
- Where laptops are used, the ‘computer suite’ can be moved from classroom to classroom on mobile carts
- While the initial investment required for wireless LAN hardware can be similar to the cost of wired LAN hardware, installation expenses can be significantly lower
- Where a school is located on more than one site (such as on two sides of a road), it is possible with directional antennae, to avoid digging trenches under roads to connect the sites
- In historic buildings where traditional cabling would compromise the façade, a wireless LAN can avoid drilling holes in walls
- Long-term cost benefits can be found in dynamic environments requiring frequent moves and changes
- They allows the possibility of individual pupil allocation of wireless devices that move around the school with the pupil.

WLANs also have some disadvantages:

- As the number of computers using the network increases, the data transfer rate to each computer will decrease accordingly
- As standards change, it may be necessary to replace wireless cards and/or access points
- Lower wireless bandwidth means some applications such as video streaming will be more effective on a wired LAN
- Security is more difficult to guarantee, and requires configuration
- Devices will only operate at a limited distance from an access point, with the distance determined by the standard used and buildings and other obstacles between the access point and the user
- A wired LAN is most likely to be required to provide a backbone to the wireless LAN; a wireless LAN should be a supplement to a wired LAN and not a complete solution
- Long-term cost benefits are harder to achieve in static environments that require few moves and changes
- It is easier to make a wired network ‘future proof’ for high data transfer.

Wireless Network Components

There are certain parallels between the equipment used to build a WLAN and that used in a traditional wired LAN. Both networks require network interface cards or network adapter cards. A wireless LAN PC card, which contains an in-built antenna, is used to

connect notebook computers to a wireless network. Usually, this is inserted into the relevant slot in the side of the notebook, but some may be internal to the notebook. Desktop computers can also connect to a wireless network if a wireless network card is inserted into one of its internal PCI slots.

In a wireless network, an 'access point' has a similar function to the hub in wired networks. It broadcasts and receives signals to and from the surrounding computers via their adapter card. It is also the point where a wireless network can be connected into an existing wired network.

The most obvious difference between wireless and wired networks, however, is that the latter uses some form of cable to connect computers together. A wireless network does not need cable to form a physical connection between computers.

Wireless Network Configurations

Wireless networks can be configured in an ad hoc/peer-to-peer arrangement or as a local area network.

Ad Hoc/Peer-to-Peer Configuration

This is the most basic wireless network configuration. It relies on the wireless network adapters installed in the computers that are communicating with each other. A computer within range of the transmitting computer can connect to it. However, if a number of computers are networked in this way, they must remain within range of each other. Even though this configuration has no real administration overhead, it should only be a consideration for very small installations.

Benefits and Educational Uses

The installation of cables is time consuming and expensive. The advantages of not doing so are apparent: the amount of work required and the time taken to complete it are significantly reduced the network is accessible in places where wiring would have been difficult or impossible with no cables linking computers together, cable-related faults and network downtime are minimised.

Where a wireless network is in place, teachers or students can have continuous access to the network, even as they move with their equipment from class to class.

The space over which a wireless network operates is not planar but spherical. Therefore, in a multi-level site, network access is available in rooms above or below the access point, without the need for additional infrastructure.

In a location within a school where network access is required occasionally, desktop computers fitted with wireless network cards can be placed on trolleys and moved from location to location. They can also be located in areas where group work is taking place. As they are connected to the network, documents and files can be shared, and access to the Internet is available, enhancing group project work.

As the range of the wireless network extends outside the building, students and teachers can use wireless devices to gather and record data outside, e.g., as part of a science experiment or individual performance data as part of a PE class.

Technical and Purchasing Considerations

Network interface cards for wireless networks are more expensive than their wired counterparts. The cost of the access points has also to be considered.

Wireless networks work at up top 54Mbps, whereas wired networks normally work at 100Mbps (Fast Ethernet). This data transmission rate is dependant on the number of users, the distance from the access point and the fabric of the building (metal structures in walls may have an impact). A wireless network will be noticeably slow when a group of users are transferring large files. This should be considered if multimedia applications are to be delivered over the network to a significant number of users.

As the range of the network may extend beyond the walls of the building, it can be accessed from outside. Consideration should be given to what security features the equipment provides to ensure that only valid users have access to the network and that data is protected.

Unit 8 Teach Yourself Programming in Ten Years

Why is everyone in such a rush?

Walk into any bookstore, and you'll see how to Teach Yourself Java in 7 Days alongside endless variations offering to teach Visual Basic, Windows, the Internet, and so on in a few days or hours. I did the following power search at Amazon.com:

> pubdate: after 1992 and title: days and
> (title: learn or title: teach yourself)

and got back 248 hits. The first 78 were computer books (number 79 was Learn Bengali in 30 days). I replaced "days" with "hours" and got remarkably similar results: 253 more books, with 77 computer books followed by Teach Yourself Grammar and Style in 24 Hours at number 78. Out of the top 200 total, 96% were computer books.

The conclusion is that either people are in a big rush to learn about computers, or that computers are somehow fabulously easier to learn than anything else. There are no books on how to learn Beethoven, or Quantum Physics, or even Dog Grooming in a few days. Felleisen et al. give a nod to this trend in their book , when they say "Bad programming is easy. Idiots can learn it in 21 days, even if they are dummies.

Let's analyze what a title like Learn C++ in Three Days could mean:

- Learn: In 3 days you won't have time to write several significant programs, and learn from your successes and failures with them. You won't have time to work with an experienced programmer and understand what it is like to live in a C++ environment. In short, you won't have time to learn much. So the book can only be talking about a superficial familiarity, not a deep understanding. As Alexander Pope said, a little learning is a dangerous thing.
- C++: In 3 days you might be able to learn some of the syntax of C++ (if you already know another language), but you couldn't learn much about how to use the language. In short, if you were, say, a Basic programmer, you could learn to write programs in the style of Basic using C++ syntax, but you couldn't learn what C++ is actually good (and bad) for. So what's the point? Alan Perlis once said: "A language that doesn't affect the way you think about programming, is not worth knowing". One possible point is that you have to learn a tiny bit of C++ (or more likely, something like JavaScript or Flash's Flex) because you need to interface with an existing tool to accomplish a specific task. But then you're not learning

how to program; you're learning to accomplish that task.

- in Three Days: Unfortunately, this is not enough, as the next section shows.

Teach Yourself Programming in Ten Years

Researchers (Bloom (1985), Bryan & Harter (1899), Hayes (1989), Simmon & Chase (1973)) have shown it takes about ten years to develop expertise in any of a wide variety of areas, including chess playing, music composition, telegraph operation, painting, piano playing, swimming, tennis, and research in neuropsychology and topology. The key is deliberative practice: not just doing it again and again, but challenging yourself with a task that is just beyond your current ability, trying it, analyzing your performance while and after doing it, and correcting any mistakes. Then repeat. And repeat again. There appear to be no real shortcuts: even Mozart, who was a musical prodigy at age 4, took 13 more years before he began to produce world-class music. In another genre, the Beatles seemed to burst onto the scene with a string of #1 hits and an appearance on the Ed Sullivan show in 1964. But they had been playing small clubs in Liverpool and Hamburg since 1957, and while they had mass appeal early on, their first great critical success, Sgt. Peppers, was released in 1967. Malcolm Gladwell reports that a study of students at the Berlin Academy of Music compared the top, middle, and bottom third of the class and asked them how much they had practiced:

> Everyone, from all three groups, started playing at roughly the same time - around the age of five. In those first few years, everyone practised roughly the same amount - about two or three hours a week. But around the age of eight real differences started to emerge. The students who would end up as the best in their class began to practise more than everyone else: six hours a week by age nine, eight by age 12, 16 a week by age 14, and up and up, until by the age of 20 they were practising well over 30 hours a week. By the age of 20, the elite performers had all totalled 10,000 hours of practice over the course of their lives. The merely good students had totalled, by contrast, 8,000 hours, and the future music teachers just over 4,000 hours.

So it may be that 10,000 hours, not 10 years, is the magic number. (Henri Cartier-Bresson (1908-2004) said "Your first 10,000 photographs are your worst," but he shot more than one an hour.) Samuel Johnson (1709-1784) thought it took even longer: "Excellence in any department can be attained only by the labor of a lifetime; it is not to be purchased at a lesser price." And Chaucer (1340-1400) complained "the lyf so short, the craft so long to lerne." Hippocrates (c. 400BC) is known for the excerpt "ars longa, vita brevis", which is part of the longer quotation "Ars longa, vita brevis, occasio praeceps, experimentum periculosum, iudicium difficile", which in English renders as "Life is short, [the] craft long, opportunity fleeting, experiment treacherous, judgment

difficult." Although in Latin, ars can mean either art or craft, in the original Greek the word "techne" can only mean "skill", not "art".

Here's my recipe for programming success:

- Get interested in programming, and do some because it is fun. Make sure that it keeps being enough fun so that you will be willing to put in ten years.
- Talk to other programmers; read other programs. This is more important than any book or training course.
- Program. The best kind of learning is learning by doing. To put it more technically, "the maximal level of performance for individuals in a given domain is not attained automatically as a function of extended experience, but the level of performance can be increased even by highly experienced individuals as a result of deliberate efforts to improve." (p. 366) and "the most effective learning requires a well-defined task with an appropriate difficulty level for the particular individual, informative feedback, and opportunities for repetition and corrections of errors." (p. 20-21) The book Cognition in Practice: Mind, Mathematics, and Culture in Everyday Life is an interesting reference for this viewpoint.
- If you want, put in four years at a college (or more at a graduate school). This will give you access to some jobs that require credentials, and it will give you a deeper understanding of the field, but if you don't enjoy school, you can (with some dedication) get similar experience on the job. In any case, book learning alone won't be enough. "Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter" says Eric Raymond, author of The New Hacker's Dictionary. One of the best programmers I ever hired had only a High School degree; he's produced a lot of great software, has his own news group, and made enough in stock options to buy his own nightclub.
- Work on projects with other programmers. Be the best programmer on some projects; be the worst on some others. When you're the best, you get to test your abilities to lead a project, and to inspire others with your vision. When you're the worst, you learn what the masters do, and you learn what they don't like to do (because they make you do it for them).
- Work on projects after other programmers. Be involved in understanding a program written by someone else. See what it takes to understand and fix it when the original programmers are not around. Think about how to design your programs to make it easier for those who will maintain it after you.
- Learn at least a half dozen programming languages. Include one language that supports class abstractions (like Java or C++), one that supports functional

abstraction (like Lisp or ML), one that supports syntactic abstraction (like Lisp), one that supports declarative specifications (like Prolog or C++ templates), one that supports coroutines (like Icon or Scheme), and one that supports parallelism (like Sisal).

- Remember that there is a "computer" in "computer science". Know how long it takes your computer to execute an instruction, fetch a word from memory (with and without a cache miss), read consecutive words from disk, and seek to a new location on disk. (Answers here.)
- Get involved in a language standardization effort. It could be the ANSI C++ committee, or it could be deciding if your local coding style will have 2 or 4 space indentation levels. Either way, you learn about what other people like in a language, how deeply they feel so, and perhaps even a little about why they feel so.
- Have the good sense to get off the language standardization effort as quickly as possible.

With all that in mind, its questionable how far you can get just by book learning. Before my first child was born, I read all the How To books, and still felt like a clueless novice. 30 Months later, when my second child was due, did I go back to the books for a refresher? No. Instead, I relied on my personal experience, which turned out to be far more useful and reassuring to me than the thousands of pages written by experts.

Fred Brooks, in his essay No Silver Bullet identified a three-part plan for finding great software designers:

1. Systematically identify top designers as early as possible.
2. Assign a career mentor to be responsible for the development of the prospect and carefully keep a career file.
3. Provide opportunities for growing designers to interact and stimulate each other.

This assumes that some people already have the qualities necessary for being a great designer; the job is to properly coax them along. Alan Perlis put it more succinctly: "Everyone can be taught to sculpt: Michelangelo would have had to be taught how not to. So it is with the great programmers".

So go ahead and buy that Java book; you'll probably get some use out of it. But you

won't change your life, or your real overall expertise as a programmer in 24 hours, days, or even months.

References

Bloom, Benjamin (ed.) Developing Talent in Young People, Ballantine, 1985.

Brooks, Fred, No Silver Bullets, IEEE Computer, vol. 20, no. 4, 1987, p. 10-19.

Bryan, W.L. & Harter, N. "Studies on the telegraphic language: The acquisition of a hierarchy of habits. Psychology Review, 1899, 8, 345-375

Hayes, John R., Complete Problem Solver Lawrence Erlbaum, 1989.

Chase, William G. & Simon, Herbert A. "Perception in Chess" Cognitive Psychology, 1973, 4, 55-81.

Lave, Jean, Cognition in Practice: Mind, Mathematics, and Culture in Everyday Life, Cambridge University Press, 1988.

Unit 9 Introduction to Database Systems

What's wrong with a file system (and also what's right)

The file system that comes with your computer is a very primitive kind of database management system. Whether your computer came with the Unix file system, NTFS, or the Macintosh file system, the basic idea is the same. Data are kept in big unstructured named clumps called files. The great thing about the file system is its invisibility. You probably didn't purchase it separately, you might not be aware of its existence, you won't have to run an ad in the newspaper for a file system administrator with 5+ years of experience, and it will pretty much work as advertised. All you need to do with a file system is back it up to tape every day or two.
Despite its unobtrusiveness, the file system on a Macintosh, Unix, or Windows machine is capable of storing any data that may be represented in digital form. For example, suppose that you are storing a mailing list in a file system file. If you accept the limitation that no e-mail address or person's name can contain a newline character, you can store one entry per line. Then you could decide that no e-mail address or name may contain a vertical bar. That lets you separate e-mail address and name fields with the vertical bar character.

So far, everything is great. As long as you are careful never to try storing a newline or vertical bar, you can keep your data in this "flat file." Searching can be slow and expensive, though. What if you want to see if "philg@mit.edu" is on the mailing list? You computer must read through the entire file to check.

Let's say that you write a program to process "insert new person" requests. It works by appending a line to the flat file with the new information. Suppose, however, that several users are simultaneously using your Web site. Two of them ask to be added to the mailing list at exactly the same time. Depending on how you wrote your program, the particular kind of file system that you have, and luck, you could get any of the following behaviors:

- Both inserts succeed.
- One of the inserts is lost.
- Information from the two inserts is mixed together so that both are corrupted.

In the last case, the programs you've written to use the data in the flat file may no longer work.

So what? Emacs may be ancient but it is still the best text editor in the world. You love using it so you might as well spend your weekends and evenings manually fixing up your flat file databases with Emacs. Who needs concurrency control?

It all depends on what kind of stove you have.

Yes, that's right, your stove. Suppose that you buy a $268,500 condo in Harvard Square. You think to yourself, "Now my friends will really be impressed with me" and invite them over for brunch. Not because you like them, but just to make them envious of your large lifestyle. Imagine your horror when all they can say is "What's this old range doing here? Don't you have a Viking stove?"

A Viking stove?!? They cost $5000. The only way you are going to come up with this kind of cash is to join the growing ranks of on-line entrepreneurs. So you open an Internet bank. An experienced Perl script/flat-file wizard by now, you confidently build a system in which all the checking account balances are stored in one file, checking.text, and all the savings balances are stored in another file, savings.text.

A few days later, an unlucky combination of events occurs. Joe User is transferring $10,000 from his savings to his checking account. Judy User is simultaneously depositing $5 into her savings account. One of your Perl scripts successfully writes the checking account flat file with Joe's new, $10,000 higher, balance. It also writes the savings account file with Joe's new, $10,000 lower, savings balance. However, the script that is processing Judy's deposit started at about the same time and began with the version of the savings file that had Joe's original balance. It eventually finishes and writes Judy's $5 higher balance but also overwrites Joe's new lower balance with the old high balance. Where does that leave you? $10,000 poorer, cooking on an old GE range, and wishing you had Concurrency Control.

After a few months of programming and reading operating systems theory books from the 1960s that deal with mutual exclusion, you've solved your concurrency problems. Congratulations. However, like any good Internet entrepreneur, you're running this business out of your house and you're getting a little sleepy. So you heat up some coffee in the microwave and simultaneously toast a bagel in the toaster oven. The circuit breaker trips. This is the time when you are going to regret having bought that set of Calphalon pots to go with your Viking stove rather than investing in an uninterruptible

power supply for your server. You hear the sickening sound of disks spinning down. You scramble to get your server back up and don't really have time to look at the logs and notice that Joe User was back transferring $25,000 from savings to checking. What happened to Joe's transaction?

The good news for Joe is that your Perl script had just finished crediting his checking account with $25,000. The bad news for you is that it hadn't really gotten started on debiting his savings account. You're so busy preparing the public offering for your on-line business that you fail to notice the loss. But your underwriters eventually do and your plans to sell the bank to the public go down the toilet.

Where does that leave you? Cooking on an old GE range and wishing you'd left the implementation of transactions to professionals.

What Do You Need for Transaction Processing?

Data processing folks like to talk about the "ACID test" when deciding whether or not a database management system is adequate for handling transactions. An adequate system has the following properties:

Atomicity

> Results of a transaction's execution are either all committed or all rolled back. All changes take effect, or none do. That means, for Joe User's money transfer, that both his savings and checking balances are adjusted or neither are. For a Web content management example, suppose that a user is editing a comment. A Web script tells the database to "copy the old comment value to an audit table and update the live table with the new text". If the hard drive fills up after the copy but before the update, the audit table insertion will be rolled back.

Consistency

The database is transformed from one valid state to another valid state. This defines a transaction as legal only if it obeys user-defined integrity constraints. Illegal transactions aren't allowed and, if an integrity constraint can't be satisfied then the transaction is rolled back. For example, suppose that you define a rule that postings in a discussion forum table must be tied to a valid user ID. Then you hire Joe Novice to write some admin pages. Joe writes a delete-user page that doesn't bother to check whether or not the deletion will result in an orphaned discussion forum posting. The

DBMS will check, though, and abort any transaction that would result in you having a discussion forum posting by a deleted user.

Isolation
: The results of a transaction are invisible to other transactions until the transaction is complete. For example, if you are running an accounting report at the same time that Joe is transferring money, the accounting report program will either see the balances before Joe transferred the money or after, but never the intermediate state where checking has been credited but savings not yet debited.

Durability
: Once committed (completed), the results of a transaction are permanent and survive future system and media failures. If the airline reservation system computer gives you seat 22A and crashes a millisecond later, it won't have forgotten that you are sitting in 22A and also give it to someone else. Furthermore, if a programmer spills coffee into a disk drive, it will be possible to install a new disk and recover the transactions up to the coffee spill, showing that you had seat 22A.

That doesn't sound too tough to implement, does it? And, after all, one of the most refreshing things about the Web is how it encourages people without formal computer science backgrounds to program. So why not build your Internet bank on a transaction system implemented by an English major who has just discovered Perl?

Because you still need indexing.

Finding Your Data (and Fast)

One facet of a database management system is processing inserts, updates, and deletes. This all has to do with putting information into the database. Sometimes it is also nice, though, to be able to get data out. And with popular sites getting 100 hits per second, it pays to be conscious of speed.

Flat files work okay if they are very small. A Perl script can read the whole file into memory in a split second and then look through it to pull out the information requested. But suppose that your on-line bank grows to have 250,000 accounts. A user types his account number into a Web page and asks for his most recent deposits. You've got a chronological financial transactions file with 25 million entries. Crunch, crunch, crunch. Your server laboriously works through all 25 million to find the ones with an account

number that matches the user's. While it is crunching, 25 other users come to the Web site and ask for the same information about their accounts.

You have two choices: (1) buy a 64-processor Sun E10000 server with 64 GB of RAM, or (2) build an index file. If you build an index file that maps account numbers to sequential transaction numbers, your server won't have to search all 25 million records anymore. However, you have to modify all of your programs that insert, update, or delete from the database to also keep the index current.

This works great until two years later when a brand new MBA arrives from Harvard. She asks your English major cum Perl hacker for "a report of all customers who have more than $5,000 in checking or live in Oklahoma and have withdrawn more than $100 from savings in the last 17 days." It turns out that you didn't anticipate this query so your indexing scheme doesn't speed things up. Your server has to grind through all the data over and over again.

Enter the Relational Database

You are building a cutting-edge Web service. You need the latest and greatest in computer technology. That's why you use, uh, Unix. Yeah. Anyway, even if your operating system was developed in 1969, you definitely can't live without the most modern database management system available. Maybe this guy E.F. Codd can help:

> "Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). ... Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.
>
> "Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on n-ary relations, a normal form for data base relations, and the concept of a universal data sublanguage are introduced. In Section 2, certain operations on relations (other than logical inference) are discussed and applied to the problems of redundancy and consistency in the user's model."

Sounds pretty spiffy, doesn't it? Just like what you need. That's the abstract to "A

Relational Model of Data for Large Shared Data Banks", a paper Codd wrote while working at IBM's San Jose research lab. It was published in the Communications of the ACM in June, 1970.

Yes, that's right, 1970. What you need to do is move your Web site into the '70s with one of these newfangled relational database management systems (RDBMS). Actually, as Codd notes in his paper, most of the problems we've encountered so far in this chapter were solved in the 1960s by off-the-shelf mainframe software sold by IBM and the "seven dwarves" (as IBM's competitors were known). By the early 1960s, businesses had gotten tired of losing important transactions and manually uncorrupting databases. They began to think that their applications programmers shouldn't be implementing transactions and indexing on an ad hoc basis for each new project. Companies began to buy database management software from computer vendors like IBM. These products worked fairly well but resulted in brittle data models. If you got your data representation correct the first time and your business needs never changed then a 1967-style hierarchical database was great. Unfortunately, if you put a system in place and subsequently needed new indices or a new data format then you might have to rewrite all of your application programs.

From an application programmer's point of view, the biggest innovation in the relational database is that one uses a declarative query language, SQL (an acronym for Structured Query Language and pronounced "ess-cue-el" or "sequel"). Most computer languages are procedural. The programmer tells the computer what to do, step by step, specifying a procedure. In SQL, the programmer says "I want data that meet the following criteria" and the RDBMS query planner figures out how to get it. There are two advantages to using a declarative language. The first is that the queries no longer depend on the data representation. The RDBMS is free to store data however it wants. The second is increased software reliability. It is much harder to have "a little bug" in an SQL query than in a procedural program. Generally it either describes the data that you want and works all the time or it completely fails in an obvious way.

Another benefit of declarative languages is that less sophisticated users are able to write useful programs. For example, many computing tasks that required professional programmers in the 1960s can be accomplished by non-technical people with spreadsheets. In a spreadsheet, you don't tell the computer how to work out the numbers or in what sequence. You just declare "This cell will be 1.5 times the value of that other cell over there."

RDBMSes can run very very slowly. Depending on whether you are selling or buying computers, this may upset or delight you. Suppose that the system takes 30 seconds to return the data you asked for in your query. Does that mean you have a lot of data? That you need to add some indices? That the RDBMS query planner made some bad choices and needs some hints? Who knows? The RDBMS is an enormously complicated program that you didn't write and for which you don't have the source code. Each vendor has tracing and debugging tools that purport to help you, but the process is not simple. Good luck figuring out a different SQL incantation that will return the same set of data in less time. If you can't, call 1-800-USESUNX and ask them to send you a 16-processor Sun Enterprise 10000 with 32 GB of RAM.. Alternatively, you can keep running the non-relational software you used in the 1960s, which is what the airlines do for their reservations systems.

How Does This RDBMS Thing Work?

Database researchers love to talk about relational algebra, n-tles, normal form, and natural composition, while throwing around mathematical symbols. This patina of mathematical obscurity tends to distract your attention from their bad suits and boring personalities, but is of no value if you just want to use a relational database management system.

In fact, this is all you need to know to be a Caveman Database Programmer: A relational database is a big spreadsheet that several people can update simultaneously.

Each table in the database is one spreadsheet. You tell the RDBMS how many columns each row has. For example, in our mailing list database, the table has two columns: name and email. Each entry in the database consists of one row in this table. An RDBMS is more restrictive than a spreadsheet in that all the data in one column must be of the same type, e.g., integer, decimal, character string, or date. Another difference between a spreadsheet and an RDBMS is that the rows in an RDBMS are not ordered. You can have a column named row_number and ask the RDBMS to return the rows ordered according to the data in this column, but the row numbering is not implicit as it would be with Visicalc or its derivatives such as Lotus 1-2-3 and Excel. If you do define a row_number column or some other unique identifier for rows in a table, it becomes possible for a row in another table to refer to that row by including the value of the unique ID.

Here's what some SQL looks like for the mailing list application:

```
create table mailing_list (
        email           varchar(100) not null primary key,
        name            varchar(100)
);
```

The table will be called mailing_list and will have two columns, both variable length character strings. We've added a couple of integrity constraints on the email column. The not null will prevent any program from inserting a row where name is specified but email is not. After all, the whole point of the system is to send people e-mail so there isn't much value in having a name with no e-mail address. The primary key tells the database that this column's value can be used to uniquely identify a row. That means the system will reject an attempt to insert a row with the same e-mail address as an existing row. This sounds like a nice feature, but it can have some unexpected performance implications. For example, every time anyone tries to insert a row into this table, the RDBMS will have to look at all the other rows in the table to make sure that there isn't already one with the same e-mail address. For a really huge table, that could take minutes, but if you had also asked the RDBMS to create an index for mailing_list on email then the check becomes almost instantaneous. However, the integrity constraint still slows you down because every update to the mailing_list table will also require an update to the index and therefore you'll be doing twice as many writes to the hard disk.

That is the joy and the agony of SQL. Inserting two innocuous looking words can cost you a factor of 1000 in performance. Then inserting a sentence (to create the index) can bring you back so that it is only a factor of two or three. (Note that many RDBMS implementations, including Oracle, automatically define an index on a column that is constrained to be unique.)

Anyway, now that we've executed the Data Definition Language "create table" statement, we can move on to Data Manipulation Language: an INSERT.

```
insert into mailing_list (name, email)
values ('Philip Greenspun','philg@mit.edu');
```

Note that we specify into which columns we are inserting. That way, if someone comes along later and does

```
alter table mailing_list add (phone_number varchar(20));
```

(the Oracle syntax for adding a column), our INSERT will still work. Note also that the string quoting character in SQL is a single quote. Hey, it was the '70s. If you visit the newsgroup comp.databases right now, I'll bet that you can find someone asking "How do I insert a string containing a single quote into an RDBMS?" Here's one harvested from AltaVista:

> demaagd@cs.hope.edu (David DeMaagd) wrote:
>
> >hwo can I get around the fact that the ' is a reserved character in
> >SQL Syntax? I need to be able to select/insert fields that have
> >apostrophies in them. Can anyone help?
>
> You can use two apostrophes " and SQL will treat it as one.
>
> ==========
> Pete Nelson | Programmers are almost as good at reading
> weasel@ecis.com | documentation as they are at writing it.
> ==========

We'll take Pete Nelson's advice and double the single quote in "O'Grady":

```
insert into mailing_list (name, email)
values ('Michael O''Grady','ogrady@fastbuck.com');
```

Having created a table and inserted some data, at last we are ready to experience the awesome power of the SQL SELECT. Want your data back?

```
select * from mailing_list;
```

If you typed this query into a standard shell-style RDBMS client program, for example Oracle's SQL*PLUS, you'd get ... a horrible mess. That's because you told Oracle that the columns could be as wide as 100 characters (varchar(100)). Very seldom will you need to store e-mail addresses or names that are anywhere near as long as 100 characters. However, the solution to the "ugly report" problem is not to cut down on the maximum allowed length in the database. You don't want your system failing for people who happen to have exceptionally long names or e-mail addresses. The solution is either to use a more sophisticated tool for querying your database or to give SQL*Plus some hints for preparing a report:

```
SQL> column email format a25
SQL> column name  format a25
SQL> column phone_number format a12
SQL> set feedback on
SQL> select * from mailing_list;

EMAIL                     NAME                      PHONE_NUMBER
------------------------- ------------------------- ------------
philg@mit.edu             Philip Greenspun
ogrady@fastbuck.com       Michael O'Grady

2 rows selected.
```

Note that there are no values in the phone_number column because we haven't set any. As soon as we do start to add phone numbers, we realize that our data model was inadequate. This is the Internet and Joe Typical User will have his pants hanging around his knees under the weight of a cell phone, beeper, and other personal communication accessories. One phone number column is clearly inadequate and even work_phone and home_phone columns won't accommodate the wealth of information users might want to give us. The clean database-y way to do this is to remove our phone_number column from the mailing_list table and define a helper table just for the phone numbers. Removing or renaming a column turns out to be impossible in Oracle 8 (see the "Data Modeling" chapter for some ALTER TABLE commands that become possible starting with Oracle 8i), so we

```
drop table mailing_list;

create table mailing_list (
        email           varchar(100) not null primary key,
        name            varchar(100)
);

create table phone_numbers (
        email           varchar(100) not null references mailing_list(email),
        number_type     varchar(15) check (number_type in
('work','home','cell','beeper')),
        phone_number    varchar(20) not null
);
```

Note that in this table the email column is not a primary key. That's because we want to allow multiple rows with the same e-mail address. If you are hanging around with a database nerd friend, you can say that there is a relationship between the rows in the phone_numbers table and the mailing_list table. In fact, you can say that it is a many-to-one relation because many rows in the phone_numbers table may correspond to only one row in the mailing_list table. If you spend enough time thinking about and talking about your database in these terms, two things will happen:

1. You'll get an A in an RDBMS course at any state university.
2. You'll pick up readers of Psychology Today who think you are sensitive and caring because you are always talking about relationships. [see "Using the Internet to Pick up Babes and/or Hunks" at http://philip.greenspun.com/wtr/getting-dates.html before following any of my dating advice]

Another item worth noting about our two-table data model is that we do not store the user's name in the phone_numbers table. That would be redundant with the mailing_list table and potentially self-redundant as well, if, for example, "robert.loser@fastbuck.com" says he is "Robert Loser" when he types in his work phone and then "Rob Loser" when he puts in his beeper number, and "Bob Lsr" when he puts in his cell phone number while typing on his laptop's cramped keyboard. A database nerd would say that that this data model is consequently in "Third Normal Form". Everything in each row in each table depends only on the primary key and nothing is dependent on only part of the key. The primary key for the phone_numbers table is the combination of email and number_type. If you had the user's name in this table, it would depend only on the email portion of the key.

Anyway, enough database nerdism. Let's populate the phone_numbers table:

```
SQL> insert into phone_numbers values ('ogrady@fastbuck.com','work','(800)
555-1212');

ORA-02291: integrity constraint (SCOTT.SYS_C001080) violated - parent
key not found
```

Ooops! When we dropped the mailing_list table, we lost all the rows. The phone_numbers table has a referential integrity constraint ("references mailing_list") to make sure that we don't record e-mail addresses for people whose names we don't know. We have to first insert the two users into mailing_list:

```
insert into mailing_list (name, email)
values ('Philip Greenspun','philg@mit.edu');
insert into mailing_list (name, email)
values ('Michael O''Grady','ogrady@fastbuck.com');


insert into phone_numbers values ('ogrady@fastbuck.com','work','(800) 555-
1212');
insert into phone_numbers values ('ogrady@fastbuck.com','home','(617) 495-
6000');
insert into phone_numbers values ('philg@mit.edu','work','(617) 253-8574');
insert into phone_numbers values ('ogrady@fastbuck.com','beper','(617) 222-
3456');
```

Note that the last four INSERTs use an evil SQL shortcut and don't specify the columns into which we are inserting data. The system defaults to using all the columns in the order that they were defined. Except for prototyping and playing around, we don't recommend ever using this shortcut.

The first three INSERTs work fine, but what about the last one, where Mr. O'Grady misspelled "beeper"?

```
ORA-02290: check constraint (SCOTT.SYS_C001079) violated
```

We asked Oracle at table definition time to check (number_type in ('work','home','cell','beeper')) and it did. The database cannot be left in an inconsistent state.

Let's say we want all of our data out. Email, full name, phone numbers. The most obvious query to try is a join.

```
SQL> select * from mailing_list, phone_numbers;

EMAIL          NAME           EMAIL          TYPE   NUMBER
-------------- -------------- -------------- ------ --------------
philg@mit.edu    Philip Greenspun ogrady@fastbuck. work   (800) 555-1212
ogrady@fastbuck. Michael O'Grady  ogrady@fastbuck. work   (800) 555-
1212
```

```
philg@mit.edu    Philip Greenspun ogrady@fastbuck. home   (617) 495-6000
ogrady@fastbuck. Michael O'Grady  ogrady@fastbuck. home   (617) 495-
6000
philg@mit.edu    Philip Greenspun philg@mit.edu    work   (617) 253-8574
ogrady@fastbuck. Michael O'Grady  philg@mit.edu    work   (617) 253-8574

6 rows selected.
```

Yow! What happened? There are only two rows in the mailing_list table and three in the phone_numbers table. Yet here we have six rows back. This is how joins work. They give you the Cartesian product of the two tables. Each row of one table is paired with all the rows of the other table in turn. So if you join an N-row table with an M-row table, you get back a result with N*M rows. In real databases, N and M can be up in the millions so it is worth being a little more specific as to which rows you want:

```
select *
from mailing_list, phone_numbers
where mailing_list.email = phone_numbers.email;

EMAIL           NAME            EMAIL           TYPE   NUMBER
--------------- --------------- --------------- ------ --------------
ogrady@fastbuck. Michael O'Grady  ogrady@fastbuck. work   (800) 555-
1212
ogrady@fastbuck. Michael O'Grady  ogrady@fastbuck. home   (617) 495-
6000
philg@mit.edu    Philip Greenspun philg@mit.edu    work   (617) 253-8574

3 rows selected.
```

Probably more like what you had in mind. Refining your SQL statements in this manner can sometimes be more exciting. For example, let's say that you want to get rid of Philip Greenspun's phone numbers but aren't sure of the exact syntax.

```
SQL> delete from phone_numbers;

3 rows deleted.
```

Oops. Yes, this does actually delete all the rows in the table. You probably wish you'd typed

```
delete from phone_numbers where email = 'philg@mit.edu';
```

but it is too late now.

There is one more fundamental SQL statement to learn. Suppose that Philip moves to Hollywood to realize his long-standing dream of becoming a major motion picture producer. Clearly a change of name is in order, though he'd be reluctant to give up the e-mail address he's had since 1976. Here's the SQL:

```
SQL> update mailing_list set name = 'Phil-baby Greenspun' where email =
'philg@mit.edu';

1 row updated.

SQL> select * from mailing_list;

EMAIL              NAME
------------------- --------------------
philg@mit.edu       Phil-baby Greenspun
ogrady@fastbuck.com  Michael O'Grady

2 rows selected.
```

As with DELETE, don't play around with UPDATE statements unless you have a WHERE clause at the end.

Brave New World

The original mid-1970s RDBMS let companies store the following kinds of data: numbers, dates, and character strings. After more than twenty years of innovation, you can today run out to the store and spend $300,000 on an "enterprise-class" RDBMS that will let you store the following kinds of data: numbers, dates, and character strings.

With an object-relational database, you get to define your own data types. For example, you could define a data type called url...

http://www.postgresql.org.

Braver New World

If you really want to be on the cutting edge, you can use a bona fide object database, like Object Design's ObjectStore (acquired by Progress Software). These persistently store the sorts of object and pointer structures that you create in a Smalltalk, Common Lisp,

C++, or Java program. Chasing pointers and certain kinds of transactions can be 10 to 100 times faster than in a relational database. If you believed everything in the object database vendors' literature, then you'd be surprised that Larry Ellison still has $100 bills to fling to peasants as he roars past in his Acura NSX. The relational database management system should have been crushed long ago under the weight of this superior technology, introduced with tremendous hype in the mid-1980s.

After 10 years, the market for object database management systems is about $100 million a year, perhaps 1 percent the size of the relational database market. Why the fizzle? Object databases bring back some of the bad features of 1960s pre-relational database management systems. The programmer has to know a lot about the details of data storage. If you know the identities of the objects you're interested in, then the query is fast and simple. But it turns out that most database users don't care about object identities; they care about object attributes. Relational databases tend to be faster and better at coughing up aggregations based on attributes. The critical difference between RDBMS and ODBMS is the extent to which the programmer is constrained in interacting with the data. With an RDBMS the application program--written in a procedural language such as C, COBOL, Fortran, Perl, or Tcl--can have all kinds of catastrophic bugs. However, these bugs generally won't affect the information in the database because all communication with the RDBMS is constrained through SQL statements. With an ODBMS, the application program is directly writing slots in objects stored in the database. A bug in the application program may translate directly into corruption of the database, one of an organization's most valuable assets.

More

- "A Relational Model of Data for Large Shared Data Banks", E.F. Codd's paper in the June 1970 Communications of the ACM is reprinted in Readings in Database Systems (Stonebraker and Hellerstein 1998; Morgan Kaufmann). You might be wondering why, in 1999, eight years after the world's physicists gave us the Web, I didn't hyperlink you over to Codd's paper at www.acm.org. However, the organization is so passionately dedicated to demonstrating simultaneously the greed and incompetence of academic computer scientists worldwide that they charge money to electronically distribute material that they didn't pay for themselves.
- For some interesting history about the first relational database implementation, visit http://www.mcjones.org/System_R/
- For a look under the hoods of a variety of database management systems, get Readings in Database Systems (above)

Reference

- If you want to sit down and drive Oracle, you'll find SQL*Plus User's Guide and Reference useful.
- If you're hungry for detail, you can get God's honest truth (well, Larry Ellison's honest truth anyway, which is pretty much the same thing in the corporate IT world) from Oracle8 Server Concepts.

Unit 10 Scientific Writing

A scientific paper is a written report describing original research results. The format of a scientific paper has been defined by centuries of developing tradition, editorial practice, scientific ethics and the interplay with printing and publishing services. A scientific paper should have, in proper order, a Title, Abstract, Introduction, Materials and Methods, Results, and Discussion.

Title

A title should be the fewest possible words that accurately describe the content of the paper. Omit all waste words such as "A study of ...", "Investigations of ...", "Observations on ...", etc. Indexing and abstracting services depend on the accuracy of the title, extracting from it keywords useful in cross- referencing and computer searching. An improperly titled paper may never reach the audience for which it was intended, so be specific. If the study is of a particular species, name it in the title. If the inferences made in the paper are limited to a particular region, then name the region in the title.

Abstract

A well prepared abstract should enable the reader to identify the basic content of a document quickly and accurately, to determine its relevance to the reader's interests, and thus to decide whether to read the document in its entirety. The abstract should succinctly state the principal objectives and scope of the investigation where these are not obvious from the title. More importantly, the abstract should concisely summarize the results and principal conclusions. The abstract should not include details of the methods employed unless the study is methodological, i.e. primarily concerned with methods. The abstract must be brief, not exceeding 250 words or as otherwise defined by the journal. If the essential details of the paper can be conveyed in 100 words, do not use 200. Do not repeat information contained in the title. The abstract, together with the title, must be self-contained as it is often published separately from the paper in abstracting services. Omit all references to the literature and to tables or figures, and omit obscure abbreviations and acronyms even though they may be defined in main body of the paper.

Rules for Scientific Writing

- Interest, inform, and persuade the reader
- Write for your reader and write clearly
- Eliminate unnecessary redundancy
- Avoid digressions

- Don't over explain and avoid overstatement
- Avoid unnecessary qualifiers
- Use consistent tenses
- Use the precise word
- Simpler words are preferred over complex words and use concrete words and examples
- Simpler sentences are preferred over more complicated sentences
- Use the active voice (except generally in methods)
- Make sure the subject and verb agree
- Use affirmative rather than negative constructions
- Avoid use of the indefinite "this"
- Use transitions
- Cite sources as well as findings
- Proofread your paper carefully; spell check does not catch everything; "there" is spelled correctly but not if you meant "their"

In general, the best writing is simple and direct. Writing that is simple and direct is most easily understood. It also tends to be the most forceful and memorable. Use no more words than necessary — and never use a complicated word if a simpler one will do just as well. Many people seem to feel that writing in a complicated way makes one sound serious, scholarly and authoritative. While this type of writing may sound serious, it is no more authoritative than writing that is simple and direct. Certainly, it is more difficult to understand. Often, it sounds pompous and overbearing. If your purpose is to be understood in a way that is both forceful and memorable, adopt a style that is simple and direct.

Unit 11 IT Career Path

At some point, the IT professional rising from the ranks needs to make a major career decision: do I advance my career as a hands-on technical professional or should I focus on managing technical people? Here are some key points to consider when making this major IT career path decision.

What do I like to do?

There is a big difference between being hands-on technical and managing people, so start out by determining which you like best. Think back on the things you have done over the last 6 months and ask yourself what efforts really excited you. Do you enjoy working independently or with other people? In general, managers spend a lot of time working with other people while technical people will spend more time working independently with the various tools and technologies.

During your career you probably had some personality tests like Meyers Briggs and you should also review those to gain additional insights. If you test out as an Introvert with a deep technical slant, then staying technical might be the right step. Conversely, if you are an Extrovert who loves engaging with others, then management could be the right path.

Where will it take me and what will be the challenges?

For each career path, consider where it will take you in 5 and 10 years. If you are looking to management, how far will you go? Not everyone reaches the VP/CIO/CTO level, so try to determine where you would like to end up.

If you are technical, where do you want to specialize? One of the chief challenges in staying technical is to continually prove your worth in the face of the increasing number of younger technicians who have grown up with a Blackberry in their hand. So choosing the right in-demand technical skills is important as the ColdFusion programmer that was hot a while ago is not in demand today.

Now think about what your job would be like on a daily basis. The manager will have to deal with various personalities and navigate personal agendas, while the technical

professional will mostly be focused on meeting the requirements and timelines for projects and initiatives. If “politics” makes you crazy, then management might not be the path for you. On the other hand, if you are terrific at liaising across various groups and driving consensus, then management could be the right choice.

Of course you can consider salary and compensation, but unless you are going for the C-level and high level management roles, this probably will not be your key factor in making a decision here. You can check out PayScale.com and Salary.com to get an idea of what folks are being paid in the jobs you identified.

What will I need to know in the future to be secure in my job?

For each career choice, consider what you would need to do to advance and maintain your career track. Management positions will require a strong understanding of the key business areas so you can partner with your internal customers to advance their goals through the use of technology. Those who are secure in their jobs are the ones who have an intimate understanding of the market, the competition and the inner workings of the company as this level of knowledge is not easily replaced.

On the technical side, no matter what you know now, you will have to learn new tools and technologies moving forward. Does that excite you or make you groan? Most importantly, you need to specialize in an area that cannot be easily outsourced and replaced. This is much easier on the applications side than the infrastructure side so consider how secure and in-demand you will be with your acquired skills. Consider too whether you are committed to securing the training needed to advance your skills, even if your employer does not provide this for you.

Validating your decision

Once you have gone through this analysis, some excellent next steps to ensure you are making the right decision are to talk to people who are in the positions you have identified and find out what their job is like. Even better if you can find a Mentor to guide you. You can also hire an IT Career Coach to help provide the assessments and guidance to assist you.

For most IT professionals, this is the most important decision in your career so take the time to do the analysis and validation before you make the plunge.

Unit 12 How to Get a Job in Computers

Steps

1 **Survey the field**. The first thing you need to think about is exactly what kind of job in computing you want. Each job has its own special requirements, so you should assess your own skills and then decide which job might be best for you. Supply and demand is important too. Remember that traditional programming jobs are moving to China etc. But new roles are coming up like Business Analysis, Testing and Compliance. Please see Types of Computing Jobs below for an overview of the most common types of computer jobs.

2 **Play**. Sit down in front of the computer and just play and experiment. This is a great way to learn new programs, but isn't the best way to learn how to configure an operating system or write programs. At the very least, you'll become comfortable with computers by doing this.

3 **Find a Mentor**. You probably know someone who knows more about computers than you do. Learn from them. Once their knowledge is used up, find someone even more knowledgeable to learn from. Soon, you'll be the expert, and people will start coming to you!

4 **Read a Book or a Website**. These days, there are websites that teach you just about anything to do with computers, from the basics all the way through advanced programming. If you Google any specific problem you have, you are sure to find an answer. If you want to find random computer tips, just Google "computer tips" or other phrases like that. Many websites have random tips that help you learn more and more about computers.

5 **Get certified by reliable company**. Same company that offers software (Red Hat, Sun, Microsoft, Oracle and many others) may offer the paid official exams, on success giving the written confirmation about your competence. As they do not teach you and just test the existing knowledge, this is frequently very cheap in comparison to the paid course. You will win against candidates that do not know the technologies they say they understand and just list the known names in their CV.

6 **Get On the Job Training**. If you already have a computer-related job (but want a better one), find someone at work you can learn from, or take on new projects where you can learn as you go along. It will be hard at first, but the more you learn, the

better your skills will become, and you'll become eligible for promotions or for better jobs at other companies.

7 **Take a Course**. This is the most obvious approach, and yet many in the industry have long careers in computing in any of the jobs above without any formal training. Still, not all computer skills are easy to teach yourself, and as more and more students graduate with degrees in computer science, the competition will make it harder for the self-trained to land the best jobs. A degree, certificate course, or specialized certification such as an MCSE will greatly improve the odds.

8 **Get Your Foot in the Door**. Once you have the skills you need to get a job, you still have the hardest part ahead of you - getting hired. Since your resume probably doesn't reflect computer work experience, you'll need to add a "Skills" section that lists all of the skills you've acquired. You might also want to mention something about computers in an "Interests" or "Hobbies" section. Make sure your resume looks extremely professional. You're submitting it to folks who use a word processor to write their grocery list - you don't want to give them something you threw together on an old ribbon typewriter.

9 **Network**. Find out where the computer guys (or girls) hang out. You'll be surprised how much info you can get just talking to people in the field. And you might also find that it's not your cup of tea. Most people that WORK in computing don't fit the stereotype. There are a lot of game players in the industry, but there are very few high paying jobs that allow you to play all day. It is a real career that requires a LOT of work and a lot of coffee drinking.

Types of Computing Jobs

Data Entry - This is a job just about anyone can get. Basically, you take information from a piece of paper and use it to fill out a form on the computer. Many old hands who started out in this role are now heading up computer departments.

- Secretarial/Administrative - This position involves some basic office skills. Not only must you understand the basics of using your computer and a few applications, but you'll probably also be expected to take dictation, answer phones, type letters, and keep things organized. In terms of computer skills, you should know how to use word processing, accounting, and spreadsheet programs at the very least. People in this role often move into other computing roles such as Managers, Meeting Organizers and Human Resources. Naturally you can move into mainstream computing areas, particularly QA and Testing.

- Power User - Not so much a position as a status of being an extremely proficient user of (typically) Microsoft Office or similar tools. Advanced users of these tools become familiar with the basics of computer programming through starting with Excel macros or Access database programming. One can become very valuable to a small business by learning such skills, and even start to consult with other small businesses at rates typically starting around $50 an hour.
- Customer Service/Telesales - These positions usually place a higher emphasis on phone skills than computer skills, but you should know at least the basics of how to use your computer.
- Technical Support (Production Support) - Most companies consider technical support to be an entry-level computer job. You are expected to know the operating systems on which the product you'll support will run, and you'll also need to know the basics of any programs that product might interact with. The good news is that the company will teach you what you need to know about their products - you just need to learn everything else. Success in technical support requires good problem-solving skills and a great deal of attention to detail. Technical Support and Problem Management is a rapidly growing area. Users now rely heavily on Help Lines, International Support Centers and the like.

- Software Quality Assurance (SQA) Engineer - You need to know as much as the best technical support personnel. You need to be a problem solver, a detective, and sometimes even a Customer Service representative. You'll also need some basic programming skills, since more and more companies are beginning to rely on automated testing. The best SQA engineers understand a little (or a lot) about every aspect of computers, from building them to using them to programming them.
- Software Engineer (Developer or Programmer) - To get a job at a top software shop such as Microsoft or Google, you'll need a degree in computer science and detailed understanding of the field. However getting a developer position in some small company may be easier. What do you need to know is the language in which you'll be programming. It is also important to know database fundamentals and (if programming for Windows) the Windows API. Knowing more than one programming language is very helpful. Understanding many of the basic fundamentals of computer science (e.g. linked lists, arrays, pointers, object oriented programming) will be essential in demonstrating your proficiency.
- Business Analyst (Analyst or Systems Analyst or Analyst/Programmer or User Analyst) - This is a relatively new title, but the role is as "old as the hills". People

can become a BA with any mix of business and computing skills. It is really a matter of looking at what the company is really after. A good BA should know the process from end to end. The BA is primarily the connection between the business and the developers. To get into this job, and into computing, good knowledge of a business is helpful. So, if you gain good knowledge through your job, and maybe do a computer course, you can get your foot in the door.

- Tester (Test Manager) - This one may not seem glamorous, but Testing is seen by the employers as being Number One in importance. It is often an easy way to break into computing, and you don't get many people say "Boy, I really want to be a Tester." Once in this job, you really get to know the whole process, and can easily get into Compliance or Management. Caution. Its usually the Test Manager, who gets the blame if the implementation goes wrong. But who cares. He can always get another job, as most know about this.
- Graphic Designer - A graphic designer create digital art such as designs for company logo, advertising brochures, and websites.
- Database Administrator/Programmer (DBA) - Database specialists are often software engineers, but not all software engineers work with databases, and some database specialists do not have high formality software engineering or computer science training, having come in via support-oriented career paths which can lead into database administration. DBAs are highly compensated and command considerable influence in typical corporate IT settings. Some DBAs get started by programming Access databases, move to SQL Server, and then to Oracle, through pursuing applied, product-specific certifications. Once a DBA, one can then move into data architecture and systems analysis.
- MIS/Network Administration/User Support - MIS (Management of Information Systems) is responsible for making sure that a company's network of computers is working properly at all times. This includes everything from showing the users how to send an e-mail to upgrading or repairing the computers to managing network resources such as file servers, network printers, and Internet firewalls. For user support positions, you need to be an expert at the operating systems in use by computers on the network and the network itself. You also need to know the fundamentals of hardware repair, the Internet, and the applications in use on the network. Network administrators need to know all of that plus how to set up network hardware, cabling, and network resources. Larger companies prefer their MIS personnel to have (or at least be pursuing) special certifications that prove they know their stuff.
- Technical Writer (Technical Author, Documentation Analyst) - To be a Technical Writer, you must understand computer basics and the product about which you're

writing. You also need to know the programs you'll be using for your writing, such as word processors, desktop publishing programs, web languages such as HTML, and Windows Help-authoring tools. You'll also need to be a good writer (or trick people into thinking you are). The best Technical Authors tend to be ex or trained Journalists or English Teachers, who have an obvious head start. Ex Teachers do have a reputation of doing very well in the computer arena, possibly due to their presentation and management skills.

- Compliance - This is a rapidly increasing area, due to exposure of Companies to large payouts (can run into billions) to Government Authorities due to breaking the rules. To get into this area, you just need to show an interest in checking what others do, and making rules. Employers are interested primarily in your knowledge of computer processes, for example, how the Accounts Receivable System works, end to end. Compliance sections generally have large budgets too!
- Medicine/Diagnostic Imaging - There are lots of new jobs for computer literate people in Medicine. CT, PET, and MRI scanners all run complex software that should be operated by people with good computer skills.
- Production Analyst - Another key position. This guy runs the "real" system, and also is in charge of OKing the new systems that the developers are writing. So, if you are into power, this is the job for you.
- Medicine/Diagnostic Imaging - There are lots of new jobs for computer literate people in Medicine. CT, PET, and MRI scanners all run complex software that should be operated by people with good computer skills.
- Computer Manager (Project Leader, Executive Director, Vice President and others) - There are probably more of these jobs in computing than anything else, so don't rule it out. The industry is top heavy and full of titles, especially now that much of the real work is being done in India! Remember that these guys can earn very big money. The key job of a manager in computing is to convince users to keep funding computer projects.
- Computer Contractor - Even though this role has been around for a long time, there is still a demand. Computer Contractors are usually experienced Professionals but not Managers. Typical Contractor roles are Business Analyst, Tester and Developer. Remember that many computer teams are made up predominantly of Contractors, and that they can make good money, in a booming economy.
- Onshore Consultant - Typically a Senior Position but based in a foreign country. Onshore Consultants can be anything from Senior Managers to Developers. An example of an Onshore Consultant is a Professional from China, Pakistan etc.

working in Canada.

- Offshore Consultant - A growing industry. The Offshore Consultant is based in his own country and gets his work from overseas, for example, a Developer based in China getting Specifications from Singapore.