CS125 : Introduction to Computer Science

Lecture Notes #2
Architecture and Program Development

Lecture 2 : Architecture and Program Development

Data Encoding

The idea of *data encoding* is to take information in one form, and translate that information into an entirely different form, with the intention being that we can translate back to the first form eventually. We presumably want to translate information to the second form because the second form is easier to deal with in some way – otherwise, we would just keep the information in the first form to begin with!

Talking on the phone is a good example of this. Imagine you are in Champaign-Urbana and you are talking on the phone to someone in California. If you did not have a phone, and just stood in the middle of the Quad and yelled, the person in California is not going to hear you, no matter how loud you yell. They are just too far away; the sound generated by your voice can be carried through the air to the people standing near you, but that sound cannot be carried by the air all the way to California.

Having a telephone, however, changes this. When you speak, the telephone you are holding encodes the sounds of your voice into electrical signals. And though sound cannot easily be transported long distances, electrical signals can be transported long distances, though wires. Of course, this idea is useless unless your friend in California also has a working phone, since your friend cannot understand electrical signals! But, assuming there is a phone on the other end of the line to decode the electrical signals back into sounds, then your friend can hear your voice in California.

The concept here was that sounds were not convenient for sending across long distances, so we encoded the sounds (our voices) into a form that was more convenient to us – electrical signals – and then decoded the information back to the first form – sound waves – once the transport was done. If we think of vocal sounds as information, then when our goal is creating or hearing that information with our own body, then sound waves are the more useful form, and thus we want the information in "sound wave" form. But when our goal is transporting the information long distances quickly, then electrical signals are the more useful form, and thus we want the information in "electrical signal" form. We can freely convert between the "sound wave" form and the "electrical signal" form (using the phone hardware), depending on which form of the information is more convenient at the time.

Encoding data as bits

The concept of data encoding is of tremendous importance in the design of a computer. The reason is that a computer is nothing more than a big piece of electronic circuitry – effectively, a pile of electrical switches hooked together by wires. Each of the electrical switches is called a *transistor*, and each of them has only two settings or positions – "on", and "off". Likewise, any of the wires in the computer either has electrical current flowing through it at that particular moment, or else it doesn't have electrical current flowing through it at that particular moment. Since most of the time, we don't want to have to think about the particular circuitry layout on a computer chip, and what position switches are in and where current is flowing, we instead represent the idea of a single transistor or the flow of electricity on a single wire, with the idea of a *bit*.

A *bit* is a single digit that is always either `1` or `0`. We implement a bit in hardware with a transistor set to either "on" or "off", respectively, or with an electrical wire that either does (in the case of a `1`), or doesn't (in the case of a `0`), have electrical current flowing through it. But, thinking in terms of transistors and wires is more detail than we really care about in many situations, so commonly, when thinking about having a collection of transistors, or a row of wires, we will instead view it as a collection of bits. A *bit sequence* or *bit string* is then a collection of bits. For example, the following:

```
    1111100010111001
```

is a bit string that is 16 bits long. In the actual computer hardware, it would be represented by 16 transistors in a row, set as follows:

```
on on on on on off off off on off on on on off off on
```

or by a row of wires, set up as follows (with CUR meaning there is current on that wire, and NCUR meaning there is no current on that wire):

```
CUR CUR CUR CUR CUR NCUR NCUR NCUR CUR NCUR CUR CUR CUR NCUR NCUR CUR
```

When dealing with computers, *everything* gets encoded using bit strings. The example on the left below shows how we might encode a set of 4 integers using 2 bits. If we want to represent the number `3` in that case, we can do it using two bits, both set to `1` (or in other words, two transistors, both set to "on", or two wires, both with current). The example on the right below shows how we might encode 8 integers using 3 bits. The more bits we have, the larger the set of integers we can encode; given a bit string length of N, we have 2^N different bit strings of that length, and so we can encode 2^N different values (N is `2` in the example on the left below, and `3` in the example on the right below).

integer		bit string encoding
0	<-->	00
1	<-->	01
2	<-->	10
3	<-->	11

integer		bit string encoding
0	<-->	000
1	<-->	001
2	<-->	010
3	<-->	011
4	<-->	100
5	<-->	101
6	<-->	110
7	<-->	111

Basic computer architecture

Computer architecture is much more complicated than we are describing here, but the simplified view of things we are presenting is good enough for our purposes. You'll learn more details in other courses.

The two components of the computer that we are concerned with understanding are the *processor* and *memory*. The *processor* is where the circuitry for performing additions, subtractions, and so on, resides.

```
      -------------------------------------
      |                                   |
      |  processor (lots of circuitry     |
      |  here that we are not drawing in) |
      |___________________________________|
```

We need the ability to send data into the processor, and receive data out of it (for example, if you want to add `2` and `3`, you need a way to send the encoded values of `2` and `3` into the processor, plus you need a way to receive the result, the encoded value of `5`). In addition, we need to be able to tell the processor what operation it should be doing (for example, if we tell the processor to add `2` and `3`, we get a different result than if we tell the processor to multiply `2` and `3`). Therefore, this is a somewhat-more-accurate model for a processor:

```
             many wires to            many wires to
             encode one               encode a second
             input value              input value

               ||   |                  ||   |
               ||...|                  ||...|
              _||___|__________________||___|______
              |                                   |
many wires ---|  processor (lots of circuitry     |
to encode  ---|  here that we are not drawing in) |
an         ...|                                   |
operation  ---|___________________________________|
                         ||   |
                         ||...|
                         ||   |

                        many wires to
                        encode an
                        output value
```

That's basically all a processor is – a bunch of wires carrrying input, a bunch of wires carrying output, and circuitry between the input and output wires which manipulates the input signals in the desired way, to produce the desired output signals. The interesting thing about processors is how you design that circuitry – how it is that you can wire transistors together to perform addition, subtraction, etc. on encoded data values. Unfortunately, how to design a processor is beyond the scope of this course, although you will learn the beginning ideas in CS 231 and CS 232 (or ECE 290 and ECE 291).

The processor cannot store information, though – it can only process the information it is given. Where does that information come from, that we ultimately give to the processor? Well, that's where *memory* comes in. You can imagine memory as a shelving unit. Imagine there is a room with some shelving in it, and each each shelf is numbered, starting with zero at the top:

```
     ---------------------------
 0  |  top shelf
    |__________________________  <--- top shelf
 1  |  second shelf from top
    |__________________________
 2  |  third shelf from top
    |__________________________
 3  |  fourth shelf from top
    |__________________________
 4  |  fifth shelf from top
    |__________________________
 5  |  sixth shelf from top
    |__________________________
 6  |  seventh shelf from top
    |__________________________
```

I could tell you to do things such as "take the item from shelf 3 and move it to shelf 6", or "take the item from shelf 4 and throw it away". Anytime I want you to do something to one of the items on a shelf, I first tell you what shelf to go to by telling you the shelf number. It would not do you any good for me to say, "take the item from the shelf and throw it away", since you wouldn't know if I was talking about some item on shelf 0, or shelf 6, or shelf 4, or some other shelf.

You can think of computer memory as if it were a shelving unit, except the "items" we store on these "shelves" are bit strings – one per "shelf". The "shelves" are often called *memory cells* or *memory locations*, and the numerical labels on these memory locations are usually called *memory addresses*. The addresses of our memory locations start at 0, just as the shelves in our above example did. (When talking about memory, we'll use the convention of putting a letter `a` in front of the number, just to remind ourselves that it's an address. In reality, the address is represented as a bit string in the machine, just as the rest of our data is.)

```
      ---------------------------
 a0    |  row of 8 switches
       |__________________________
 a1    |  row of 8 switches
       |__________________________
 a2    |  row of 8 switches
       |__________________________
 a3    |  row of 8 switches
       |__________________________
       |       .
       |       .
       |       .
       |__________________________
a4094  |  row of 8 switches
       |__________________________
a4095  |  row of 8 switches
       |__________________________
```

In our example above, we have 4096 memory locations (i.e., 4096 numbered shelves), with addresses 0 through 4095. Each memory location holds a row of 8 switches – i.e., it holds one 8-bit-long bit string. The purpose of memory – much like the purpose of a shelving unit – is to be a storage unit where data can be stored until it is directly needed. At that time, it can be read or written by the processor. That is, memory basically supports two operations:

1. given an address, obtain the 8-bit bit string stored at that given address, and send it back to whatever part of the hardware requested that information
2. given an address and an 8-bit bit string, write the given 8-bit bit string into the memory cell at that given address – meaning that the 8-bit bit string is now stored at that address for as long as we need it to be

It may be that some data value we are trying to store takes up more than 8 bits. For example, we might decide to encode 2^{32} different integers as 32-bit bit strings. If we did that, we could not fit our 32-bit bit string into one memory cell, since each memory cell only holds 8 bits. So, in such cases, we break our larger bit string into 8-bit pieces and store them in *consecutive* memory cells. For example, if your piece of information needed 32 bits to represent, and the storage of this information started at the memory cell with address **a104**, then you need 4 memory cells to store the information, since 32 bits will take up four 8-bit cells. And, since the information storage starts at **a104**, your information not only takes up the memory cell at **a104**, but will also take up the memory cells at **a105**, **a106**, and **a107** as well. Those four consecutive memory cells, located at addresses **a104**, **a105**, **a106**, and **a107**, would together hold the 32 bits that our bit string takes up. In general, that is how we store larger chunks of information – we break it up into many consecutive 8-bit memory cells, which collectively store our larger bit string.

So, the image below is the model of a computer that we will deal with in this class. The assorted input signals to the processor come (mostly) from the memory itself, and the output signals get written back into the memory.

```
                                          address   bit string
                                          ____________________________
                                          | a0   | row of 8 switches |
                                          |------|-------------------|
_____________________________________     | a1   | row of 8 switches |
|                                   |-----|------|-------------------|
|                                   |-----| a2   | row of 8 switches |
|  processor (lots of circuitry     |-----|------|-------------------|
|  here that we are not drawing in) |-----|  .   |         .         |
|___________________________________|     |  .   |         .         |
                                          |  .   |         .         |
                                          |------|-------------------|
                                          |a4095 | row of 8 switches |
                                          |______|___________________|
```

The stored-program computer

Since bit strings are all a computer understands, we need to make sure the following two things are true in order for computers to be able to accomplish anything:

1. all our data is encoded as bit strings and stored in memory
2. all our instructions to the processor, are encoded as bit strings and stored in memory

That is, not only is our data stored in bit string form, but our instructions to the computer – our "computer programs" – are also stored in bit string form. That is the essential idea behind a *stored-program computer* – that the same hardware that is used to store our data, could also be used to store our programs instead, because our programs are encoded into bits just as our data is.

So there are two key concepts here, one related to instructions, and one related to data. The first important idea is that whatever task we want a computer to do, we need to tell it to do that task, by supplying it with some huge sequence of bits that encodes our instructions to the machine. That means that writing a program is a matter of coming up with some proper series of instructions that the processor should run through, and then encoding those instructions into a form such that the program can be stored in the computer's memory. Our large sequence of bits, specifying our program, is broken into 8-bit pieces and stored in consecutive memory cells.

The second important idea is that any given data value is also stored as a sequence of bits. If our data value is only 8 bits long, it can be stored in one memory cell (since each cell can hold up to 8 bits). Larger bit strings get broken up into 8-bit pieces and stored in consecutive memory cells, just like larger bit strings representing programs, were broken up into 8-bit pieces and stored in consecutive memory cells.

Developing a Java program

The following is one of the simplest possible Java programs – all it does is print one line of text to the screen.

```
public class HelloWorld
{
   public static void main(String[] args)
   {
      System.out.println("Hello World!");
   }
}
```

The program above is just composed of text characters, just like those you might type into a word processor if you were writing a term paper. However, one important difference between writing a computer program and writing a term paper, is that the text of the computer program must be *exact*. If you write a term paper, and accidentally type “the” as “teh”, the paper will still be readable – a human reading the paper can figure out that that is just a typo, and can continue reading your paper. However, if you have a typo in a computer program, it renders the program completely incorrect. For example, in the following text, we have forgotten the ‘c’ in “static”:

```
public class HelloWorld
{
   public stati void main(String[] args)
   {
      System.out.println("Hello World!");
   }
}
```

The program above is not even an actual Java program! The loss of one letter not only means it’s not the same program as the first one, but in addition, in this case it renders the program not even legitimate at all. (Not every typo will have that effect, but many will.) Preciseness is very important when writing computer programs.

Developing a program is a four-step cycle:

1. edit (either write a new program, or change an existing program)
2. compile (encode program in bit string from)
3. run program through tests to see how well it works
4. debug – find errors in the test results and deduce their cause

We will look at each of those four steps in detail now.

Step 1 of writing a program is to type the program into a file using a *text editor*. A *text editor* is a program that is similar to a word processor, in that you are allowed to type and edit text, but provides the sort of features important for program development, rather than the sort of features important for writing term papers. In particular, text editors generally:

- provide some of the same features as word processors, such as allowing the entering and editing of text, support for cut-and-paste and search-and-replace, the ability to save and open files, etc.
- provide some features very useful for programming, such as automatically indenting in complex ways, color-coding certain syntax features, checking to make sure you have a close-parenthesis for every open parenthesis, etc.
- save the program as plain text – that is to say, the file a text editor saves contains nothing but the text you typed in, and thus should be conveniently viewable using a different text editor. This is in contrast to using a word processor, where generally if you save a file in a word processor, there's all sorts of fancy extra formatting added to the file, and so either you have to use that particular word processor to view your file later, or else someone needs to do a lot of work to enable some other piece of software to read your file.

Some examples of text editors are `pico`, `emacs`, `xemacs`, `vi`, and `vim` on Unix, `TextEdit` and `BBEdit` on Mac OS X, and `NotePad` and `TextPad` on Windows. In addition, some of those editors are actually cross-platform; `Xemacs`, for example, is available on all three platforms. A good text editor can save you a great deal of time in developing your programs, so it is worth your while to become proficient in the use of a good text editor! `Xemacs` is a text editor that we'll give you some beginning guidance in how to use, but you are free to both learn more about `Xemacs`, or to learn a different text editor instead. At any rate, learning how to use a text editor well is one of the many situations where your courses will point you in the right direction, but most of the work will be left up to you. If you want to get through all your computer science courses using only the minimum amount of information we give you about text editors during the first few weeks of this course, you *can* – but you'll find that spending a bit more time now and again learning more features of your chosen text editor can save you a great deal of time in the future.

Once you type in your program text and save the file, that file is known as a *source file* or *source code file*, and the text inside that file is known as *source code*. In Java, we will name our source code files with the `.java` suffix, so our files will have names such as `Foo.java` or `Spacing.java` or `Test.java`. Our first program above would go into a file named `HelloWorld.java`, because the name on the first line after `public class` is `HelloWorld`. (We'll talk about the issue of file names in the next lecture packet.)

Step 2 of writing any program is to *compile* the source code for the program. That is, the next step is to encode your source code into bit string form so that the machine can understand it. The process of encoding your source code into bit string form is known as *compiling* and the software that does this encoding for you is called a *compiler*. This step is necessary because the source code is meaningless to the actual hardware. As we discussed before, your computer hardware is nothing but a collection of transistors and wires, and so you need to convert your information (the program you've written) into a form that the hardware can make sense of – namely, bit strings.

In Java, the encoded file has a `.class` suffix and otherwise has the same name as the source code file. For example, if you sent the file `HelloWorld.java` as input to the Java compiler, the output would be a file named `HelloWorld.class`, and it is this file that would contain the encoded version of your program.

The compiler's job is actually more than just encoding your program. This is because the compiler can only encode actual Java programs. If you sent the compiler the following

```
SDFSDF SWEasasdf 9745972348963qq anczdciero283  2je9 23 83e 123 q
```

then the compiler can't produce an encoded version of that, since it isn't a Java program to begin with – it's just a bunch of random characters. There are particular rules about what is and what isn't a Java program – our first "hello world" example earlier is indeed a Java program, and the random garbage above is clearly NOT a legal program...but the second "hello world" example, the one with the typo, isn't a legal program either, although it is closer to being a legal program than the above nonsense is.

What we are getting at here is that *programming languages* are different than *natural languages.* Natural languages are languages that people speak – languages such as English, Spanish, Japanese, and so on. We as people understand natural languages, so you might think they'd be ideal for writing programs in, but unfortunately, natural languages are not precise enough – the same sentence can have multiple meanings depending on context, the tone of voice it is spoken in, etc. Whereas when we specify a set of instructions to the computer, we don't want the computer to assume a different meaning than we intended. Therefore, we have very exact specifications for programming languages, so that it is very clear what is a legal program in that language and what isn't a legal program, as well as exactly what each legal program should do when it is run.

You can imagine a spectrum of possibilities:

```
<----------------------------------------------------------->
  |              |                                     |
 natural       high-level                           bit strings
languages      languages
```

The far right end of the spectrum is where we have instructions encoded as bit strings – the only language the machine actually understands (for this reason, the language of bit strings that have meaning to the machine is also known as *machine language*). The far left end of the spectrum is the collection of languages we speak, which are very understandable to us but not precise enough for usage as programming languages. Close to natural languages, but not quite so far to the left, are the languages in which most modern program development is done, and these programming languages are called *high-level languages* because of how much their syntax resembles natural languages instead of bit strings. These languages are closer to our level of speech, than bit strings would be, and that makes it much easier to develop programs in a high level language than it would be to develop programs by writing out bit string after bit string from scratch.

So, getting back to the compiler, if you send a file of legal source code to the compiler, the compiler can produce the encoded version of that source code...but if you send a file of source code that is NOT a legal program for that language, then the compiler cannot produce any encoded version of your program, since you haven't actually given the compiler a real program to begin with. However, what the compiler *will* do for you in this case, is to tell you what areas of your program violate the syntax rules of the language – that is, the compiler will tell you what lines had errors, and what it thinks those errors are. (For example, if the earlier "hello world" program with the one typo was sent in as input to the Java compiler, the Java compiler might tell us that there is an error on line 3, and furthermore, would tell you that the word "stati" on that line is not something the compiler can make any sense of.)

These kinds of errors – where a program is written that does not conform to the specification of the language – are called *syntax errors*, and the way you fix them is by changing your source code so that what you have written *does* conform to the specification of the language. The compiler's job is to tell you what syntax errors you have in your source code, and, if you don't have any syntax errors, to produce an encoded version of your source code.

Step 3 of program development is to run your program. At this point, you have an encoded version of your source code – that is, you've translated your source code so that it is in a form the machine can understand. However, there are many different kinds of machines! Some of these machines are similar to each other – for example, a computer built around the newest Pentium processor (which is the processor most commonly used in Windows machines), probably isn't all that different from a computer built around last year's Pentium processor, and so the encoding for the two machines is probably the same, or at least, very very similar. On the other hand, the difference between a computer built around a Pentium processor, and a computer built around a PowerPC processor (which is the processor used in Macintosh machines), is somewhat more significant. Because those processors are so different from each other, the encoding you use for one machine is different than the encoding you use for the other machine.

This is a big part of why software you can buy at the store for a Windows machine, probably doesn't run on a Macintosh, and vice-versa. If I were to write a story in English, and translate it to Spanish, someone who only understands Japanese isn't going to understand my original story, nor will this person understand the Spanish translation. Now, I could go back to my original story and create a second translation if I wanted – a translation to Japanese – but if I've only done the one translation, to Spanish, then anyone who can't understand the original English version and who also can't understand the Spanish translation, cannot read the story. Similarly. you can have the compiler translate your source code to run on a Pentium processor, but a PowerPC processor will not be able to make sense of the Pentium translation. You can run a second translation, and translate your source code to the PowerPC machine code, if you want, but if all you've done is the Pentium translation, then a PowerPC processor cannot run your program, since it doesn't understand the original source code, and it doesn't understand the encoding for a Pentium.

There's another option, as well – it is possible you could make up a machine – imagine a machine that nobody has built yet – and then translate your source code into the machine code for that machine. In that case, the machine code you end up with, would not run on a Pentium, nor would it run on a PowerPC. It is designed to run on this other processor which you dreamed up but which no one has built yet. Why would you do this? Well, the difficult part about performing an encoding, is going from the high level language, to some machine language. Once you have a machine language version of a program, translating that to a different machine language, is relatively easy by comparison. So, you could do the hard work of translating your code for a machine – your imaginary machine, or *virtual machine* – and then put the code out on the internet. People who want your program, could then download it, and would only have a small amount of work to have to do – they'd need to translate your encoding, to the encoding for their own processor. For example, if your computer had a Pentium processor, you could download my encoding for an imaginary machine, and then further translate that for your Pentium machine. Since the translation from one machine code to another is not hard, your task is not hard.

This is the Java model. To run a Java program, you will have to run a separate program – the Java *virtual machine* – which is a piece of software that (among other things) translates the encoded `.class` file into a form that can run on the actual machine you are on. The advantage to this process is that it makes the `.class` files somewhat *portable* – they can be run on *any* computer, as long as that computer has virtual machine software installed. If you write a program in Java, you don't need to release one version for the Pentium processor, and another version for the PowerPC processor. You can simply provide the `.class` files, and they should run on any processor that has a virtual machine available. The downside is that running your program can take a bit longer, since the virtual machine is performing the additional translation as your program runs. If speed is your top priority, therefore, Java might end up not being the best language to use...but if portability is a higher priority, and you can afford to have things slow down a little, then the Java model might be a useful one for you. (This entire discussion has been simplified a bit, but that's the basic idea.)

Step 4 of program development is to debug your program (if necessary) and then return to step 1.

Once you have a program and have translated it to machine language, you then can run the program, as we just discussed. However, the machine might not do what you want. Certainly, the machine will do what your program tells it to do, but if you've specified the wrong sequence of instructions in your program, then of course the "wrong" things will be done. These are known as *logic errors* – when the actual program you've written doesn't actually do what you intended the program to do, and so you therefore have to change the program to one that *does* do what you want it to do. That is, you need to figure out where you made an error in your chosen sequence of instructions, and then correct that error. This is generally not easy to do – it's the main difficulty in program development. When software you use crashes, or makes some other sort of mistake, it's because the programmer of that software made some logic errors that were not found and fixed before the program was made publicly available for people to use.

The debugging cycle – the cycle for fixing logic errors – is basically the same as the cycle for fixing syntax errors. That is, you make what you think are the correct changes to your program to get it to work correctly, and then you re-translate using the compiler and try to run the program again. You repeat this cycle until the program works correctly – or at least, until you believe it works correctly. Beginning programmers often take a haphazard approach to this cycle. They change any random thing in the hopes that it will work, and then try again. *Don't do this!*. You want to reason through your errors. Look carefully at your program code (the high-level language instructions you have written). Look carefully at the output your program is producing. And try and understand *why* the program is producing that incorrect output instead of the output you want it to produce. Once you've figured out why the program is doing what it is doing, you will then know what needs to be done to fix the program.