

Chapter 2. Computer Hardware and Software

If we're going to write C# programs to run on our computer (or our professor's computer), it will sure help if we understand the basics of what a typical computer looks like. In this chapter, we'll look at basic computer organization, talk about how software works on that organization, and even get to writing and running our first C# program.

2.1. Hardware Organization

Before we start talking about how the hardware in a computer is organized, what is hardware in the first place? Well, the easiest way to think about it is that hardware will hurt if you drop it on your foot! In other words, hardware consists of the parts of the computer that we can see and touch. Most computers, and certainly almost all PCs, are composed of 4 different kinds of hardware – the Central Processing Unit (CPU), memory (main and secondary), input devices, and output devices.

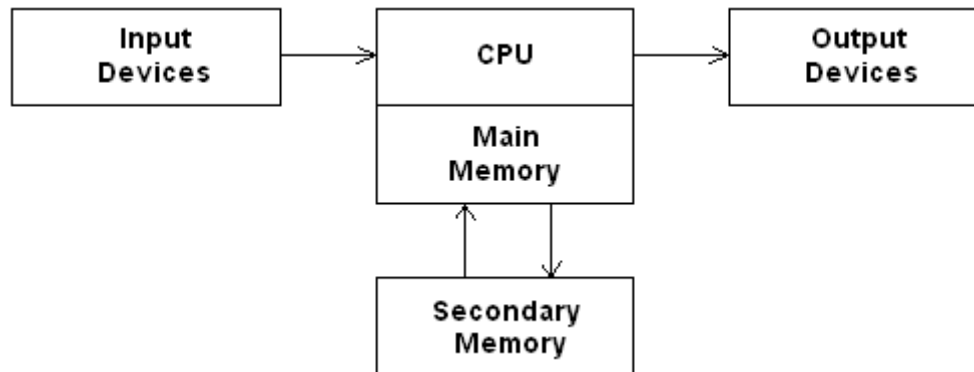


Figure 2.1. Basic Hardware Organization

The *Central Processing Unit*, commonly called the *CPU*, is really the “brains” of the computer. Computers are actually much dumber than you might think – the only thing your computer can do is execute instructions, one after the other, until you turn it off! The trick to making computers do useful things is in the instructions we tell them to execute; those instructions are in the form of computer software, which we'll get to in the following section. In any case, the CPU is the part of the computer that executes those instructions, including deciding what to do next, completing mathematical computations, and so on.

But where does the CPU get the instructions to execute? From the *main memory*. There are really two kinds of memory in the computer – main memory and secondary memory – but the CPU can only get its instructions from main memory. Main memory comes in two forms: Random Access Memory (RAM) and Read Only Memory (ROM). Read Only Memory is useful for storing some permanent information, such as instructions about what to do when the computer is turned on, but it can't be changed as the computer is running. Random Access Memory can be changed, and that's what we'll use when we write our programs. One other comment about RAM – it's called volatile memory, which means that whatever is stored in RAM will be erased when you turn off your computer.

Wait a minute, you say, I can run lots of programs when I turn my computer on; they don't just disappear! That's because they're also stored in *secondary memory*, which is non-volatile. The most common form of secondary memory is your hard disk. The program instructions to be executed are stored on your hard drive until you decide to run that program. At that point, the instructions are loaded into main memory so the CPU can get and execute them.

Ok, we know that memory can store instructions, but what else can it store? Just about anything! Ultimately, everything in the computer is stored in *binary* – that is, ones and zeros – but as long as we can encode whatever we're trying to store as ones and zeros, we can store it in memory. That means we can store numbers, letters, pictures, sound files, and all kinds of stuff in memory. We'll talk more about how we use memory as we get further into our programming.

So far, we've talked about the parts of the computer that execute instructions (the CPU) and that hold those instructions (the memory). At this point, we have a computer that can run programs all day (and even all night) long. There's one problem, though – we can't provide any input to those programs, and we can't see the results either. To do that, we need to use input and output devices. Can you imagine how boring a game would be if we couldn't provide any input to interact with the game world and didn't get any output (like a graphical display) to find out what's happening in the game world? That would be horrible!

As you might expect, *input devices* are hardware devices we use to provide input to the computer. Some of the more obvious input devices are the keyboard and the mouse, but there are lots of other input devices as well: game controllers, light pens, touch pads, track balls, microphones, scanners, and so on. Basically, anything that we can use to provide input to the computer is an input device.

Output devices, on the other hand, are the devices that provide output to the user of the computer. The most obvious such device is your computer monitor. Any output from the computer can be provided to you on the monitor, so it's clearly an output device. What are some other examples? How about speakers, force-feedback joysticks (which also serve as input devices), printers, and so on. To make our computer really useful, we need both input devices and output devices.

One last thing before we move on to software. How are all these hardware components hooked together? Using something called a bus. Think of the bus as a telephone line that everyone can listen to. All the hardware gets connected to the bus, then the hardware components can “talk” to each other by sending messages on the bus. OK, let's move on.

2.2. Computer Software

So we've talked about the parts of a computer and how they're hooked together, but let's face it – the computer is really just an expensive paperweight unless we've got software to run on it! Remember, all the computer can do is execute instructions. Computer software is the set of instructions the computer will run.

Now, we said earlier that everything gets stored in the computer as ones and zeros. Does that mean we have to write our software as sequences of ones and zeros? No, thank goodness! All we have to do is be able to convert our programs into instructions that the CPU can understand (these are often called *machine instructions* or *machine code*); we'll talk about that process soon. In the meantime, you should know that the earliest programmers actually DID write their programs in binary.

Luckily, the art and science of programming has progressed far beyond having to write our programs in binary. In fact, we can now pick from a large number of programming languages; C#, C#, C++, Ada, Visual Basic, Lisp, and Eiffel are just a few of these languages. We'll talk specifically about how C# works in the rest of our discussion, but many of the concepts are the same no matter which language you use.

Before we talk about how we get a program to run on the computer, let's talk for a minute about programming languages in general. One of the key things you need to remember as you start to program is that programming languages have both *syntax* and *semantics*. When we talk about a language's syntax, we mean the rules that determine the order of the words we include and the punctuation we use. This is a lot like the grammar rules associated with more "normal" languages (like English). For example, English grammar rules tell us how we can build and punctuate a sentence. Similarly, C# tells us how we can build and punctuate the instructions (called statements) we create in that language.

Semantics, on the other hand, relates to the meaning of the sentences (or instructions) we build. For example, the English sentence "The large blue fox jumped over the moon" is grammatically correct, but it doesn't really make sense. Similarly, we can also create C# statements that are syntactically correct but don't really make sense. The bottom line is that when we learn a new language, like English or C#, we need to learn both the syntax and the semantics of that language.

Ok, so we learn the syntax and semantics of a programming language, and we're ready to write one to run on our computer. How do we do that? Well, the first step is to type our program in using either an *editor* (like Microsoft Notepad) or an *Integrated Development Environment (IDE)* (like Microsoft Visual C# 2008 Express Edition). Because IDEs are so much more powerful than basic editors, we'll be using an IDE throughout this book. No matter how we type in our program (which is called typically called *source code*), when we're done with this step we have a program written in our programming language of choice (in our case, C#).

Now we need to somehow convert our program into instructions that the CPU can understand. Programming languages fall into two classes for this step: *interpreted languages* and *compiled languages*. In interpreted languages, an *interpreter* converts the program one statement at a time, and the CPU executes the instructions as they're converted. This can be very helpful as you're developing your program, but it can also be fairly slow because you're doing the conversion while you run your program. In compiled languages, a *compiler* converts the entire program to the machine instructions; you can then have the CPU run those instructions. Generally speaking, compiled programs run faster than interpreted programs because the translation to machine instructions has already been done. We should note that both interpreters and compilers check the syntax of the program provided to them to make sure they're grammatically correct before translating them.

So is C# a compiled language or an interpreted language? Well, it's actually both! That's one of the characteristics that makes C# such a powerful programming language. When we compile our C# programs, the compiler doesn't actually generate machine instructions; instead, it generates something called *Intermediate Language (IL)*. The IL, which you can think of as .NET instructions, isn't specific to any particular CPU, which makes the IL portable to any machine. When it's time to actually run the program, the IL is interpreted into machine instructions by the .NET Common Language Runtime (CLR). This hybrid approach gives C# great portability, and program execution time doesn't suffer too much because a lot of the translation effort occurs when the compiler generates the IL. Figure 3.2 summarizes how this process works.

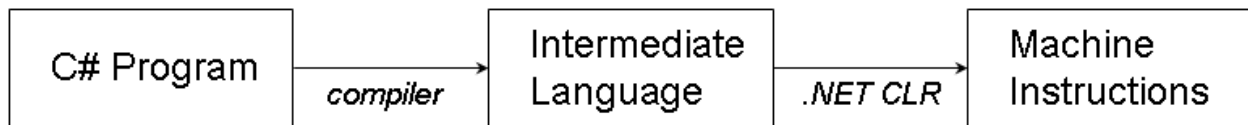


Figure 2.2. C# Compilation and Execution Process

The bottom line for computer software is that we can write that software using any of a large number of programming languages. Before the computer can run it, though, we need to translate the program from the language we're using into the machine instructions the CPU can understand. We can do that using an interpreter, a compiler, or a hybrid approach like C# uses. Once we have the machine instructions, we can actually have the computer run the software to do something useful. Remember – computer hardware is just an expensive paperweight without software to run!

2.3. Writing and Running a C# Program

At this point, we have a basic understanding of how computer hardware is organized and how computer software can be written to run on that hardware. Let's actually go through the process of writing, compiling, and executing (running) a C# program so you can actually see this really work.

There are numerous C# IDEs available for a variety of prices, but Microsoft Visual C# 2008 Express Edition is an excellent IDE that you can use for free. In fact, now would be a great time to set up your entire development environment; you should go to Appendix A and follow the instructions there. Don't worry, I'll wait right here for you.

Let's start by typing in your program. Start up the IDE and click File -> New Project. Click the Console Application icon just like you did in Appendix A, rename the application to PrintRainMessage, and click OK. Next, type in the program shown in Figure 3.3; most of the code has already been provided by the IDE, so you only have to type in the code in the shaded area in the figure. I've added some comments as well, but you don't have to type those in, just type in the 3 lines of code in the shaded area. We'll talk a lot more about this program, and C# programs in general, in the next chapter, so don't worry that a lot of it may seem strange to you at this point.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PrintRainMessage
{
    /// <remarks>
    /// A class to print a rain message
    ///
    /// Author: Axl Rose
    /// </remarks>
    class Program
    {
        /// <summary>

```

```

/// Prints a rain message
/// </summary>
/// <param name="args">command-line arguments</param>
static void Main(string[] args)
{
    Console.WriteLine("Hello, world");
    Console.WriteLine("Chinese Democracy is done and it's November");
    Console.WriteLine("Is it raining?");
    Console.WriteLine();
}
}
}

```

Figure 2.3. Program.cs

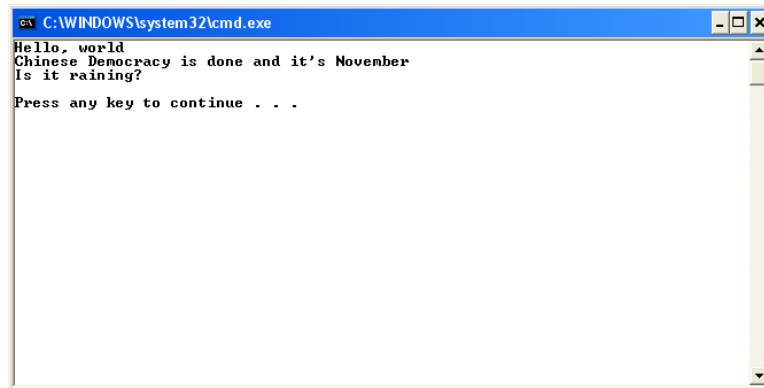
Once you've typed the program in, you need to save the project. Click the Save All icon (or use File -> Save All), pick the location where you want the project to be saved, and click Save. The source code you just typed in is saved in a file named Program.cs. The cs extension tells the compiler, and anyone else looking at the file name, that the file contains C# source code¹. The other rule is that the file name should really match the name of the class – in this case, Program; the IDE always takes care of this when you start a project, so you'd have to do something weird to mess this up. Finally, capitalization matters in C# as well, so make sure you pay attention to that. When you save the above program, the IDE should save everything fine.

Ok, we now have our C# project saved. Our next step is to compile the program into Intermediate Language. To do that, simply press F6 and wait for the Build Succeeded message in the lower left corner of the IDE window. If you end up with errors in the Error List pane (you can't miss them, they have a red circle with an X inside it), go back and figure out what you typed in wrong. Computers are great because they do exactly what we tell them to do, but they're also merciless because they do exactly what we tell them to do! Perhaps in programming more than almost any other endeavor, the details are incredibly important. The code you type in from Figure 2.3 above needs to match exactly what's there, capitalization and all; pretty close won't cut it in a computer program.

Once you have a program that compiles successfully (Build Succeeded), you can run it by pressing Ctrl-F5. You should get the window shown in Figure 2.4 below (see Appendix A to find out how to change the background and text color in the Command Prompt window).

That's all there is to it! We type in our program using an IDE, compile it into Intermediate Language using the IDE's built-in compiler, then run it. We'll talk a lot more throughout the rest of the book about how we write useful C# programs to solve interesting problems, but at least now we know the steps we go through to compile and run them.

¹ The programs that we write in C# and other programming languages are called source code because they're the initial source of the machine instructions that are ultimately generated.

A screenshot of a Windows command prompt window. The title bar reads "C:\WINDOWS\system32\cmd.exe". The window contains the following text:

```
Hello, world  
Chinese Democracy is done and it's November  
Is it raining?  
Press any key to continue . . .
```

Figure 2.4. Program Output

2.4. Exercises

1. List and describe the four major kinds of computer hardware.
2. Where must a program reside before the computer can actually execute it?
3. What's the difference between volatile memory and non-volatile memory? Which kinds of computer memory are volatile? Which are non-volatile?
4. Describe the difference between syntax and semantics.
5. Describe the difference between an interpreter and a compiler. Is C# interpreted or compiled (be careful!)?
6. List the steps we take to write and run a C# program.
7. Using the Program.cs source code as an example, write and run a C# program that prints a message of your own.