Top of Form



**Introduction to C# Programming: Lesson** 1

**Chapter 1**

**Introduction**

Hello and welcome to *C# Programming for the Absolute Beginner*! My name is Mike Orsega, and I will be leading you through the C# programming language over the next few weeks. My educational background is diverse and interesting. I have a bachelor's degree in physics and significant graduate course work in geology. My latest degree is a master's degree of applied mathematics with a computer science option. During my thesis work, I programmed in a number of different programming languages, including C++, C, Java, Visual Basic, FORTRAN, and even MPI—a multiprocessor programming language.

Since graduation I have been teaching computer programming in C++, Java, Visual Basic, and C#. I have taught in traditional settings, with students seated in front of me, and I have taught many of these courses as online only. It is my belief that if you are able to learn one programming language very well, you can take those same principles and learn a new language. For this reason, I tend to teach my courses as programming logic combined with *syntax*, the correct order and use of words that are special to the programming language.

C# is a new programming language that takes the best features of C++, Java, and Visual Basic and rolls them up in a new language. I will attempt to prove this to you during our time together over the next few weeks. This course starts with a brief introduction to programming in general and why it is important. You will next learn how to create, compile, and run a simple program in C#. You will find that the general structure is the same for all programs in C#, whether they are simple or complex. Next, I'll show you how to create and use variables to store data. Once you are comfortable with using data, you will learn how to create *methods*, which are a series of statements that will carry out a specific task. Finally, you will be introduced to the *selection structure*, used to have your program make decisions, and the *repetition structure*, used to have your program repeat sections of code until some condition is met.

By the end of the course, you will have a strong base for writing programs. Have you ever done a set of calculations over and over and wondered if there was an easier way? Maybe you think this every year when it is time to do your taxes. You always add the same boxes together and always subtract other boxes. After this course, you should be able to write a simple program to help you with your taxes.

Your ability to write computer programs will make you more valuable to your employer. Maybe taxes aren't your biggest concern, but there may be a task at work that you perform over and over. After this course, you will be able to write programs that do that task for you. When my daughter was born, it seemed like my wife wanted to send announcements to every person to which she had ever spoken. So, instead of hand addressing all of those envelopes, I chose to write a program that took the addresses from a data file and sent them to the printer, one by one. Sure, it took about the same amount of time to write the program as it would have to address all of those envelopes, but now I am ready should we need to send out another set. Maybe you address a lot of envelopes at work; wouldn't this program help you? It would give you a little bit of time to relax or work on something else that needs to be done.

Our first lesson will start with a discussion of the components of a computer and why computer programs are important to these devices. We will start with a description of hardware and software. Then we will move on to learn about storage and memory and how they relate to our programs. Finally, this lesson will wrap up with a look at different problem-solving techniques.

**Chapter 2**

**Hardware and Software**

The physical equipment of a computer system is the system's *hardware*. Many different devices make up your computer's hardware. These devices can be broken into five main categories: input, output, the central processing unit, memory, and storage.

*Input devices* allow data or instructions into the computer system. A computer is not of much use if it cannot communicate with anything else. The input devices allow data into the computer to be processed. Most computers now come with a keyboard and mouse. These two input devices take human movement and translate it into electrical data that the computer can understand. That means when you press K on your keyboard, a signal is sent to the computer to let the processor know that you pressed K. Similarly, when you move the mouse to the left, a signal is sent translating that physical movement into a signal that the computer can interpret.

As you might guess, *output devices* allow the computer system to communicate results outside of the system. Devices like a monitor, printer, and speakers allow your computer to communicate with you. If you really stop and think about it, the thing that makes a computer so useful and popular is its ability to get data in from its input devices, do some calculations with that data, and deliver the results to its output devices. This is a very simplistic way to look at a computer, but it is truly what happens.

Many people, including myself unfortunately, sometimes use the terms *memory* and *storage* as if they represent the same thing. However, they really are two different things. Memory is directly connected to the CPU. Often people refer to memory as RAM—random access memory. This is a small area where data or instructions can be put so that the processor has quick access to the information. *Memory* is volatile, which means that when the power to the computer is shut off, everything in memory is lost. Before any statements in a program can be run, the program must be loaded into memory.

*Storage*, on the other hand, is a larger, more permanent area where data or instructions can be stored. Storage devices include floppy disk drives, zip drives, and CD-R drives. As you may have guessed, since storage provides the benefits of larger space and nonvolatility, there must be some drawbacks. That drawback is speed; storage areas are much slower.

As a computer programmer, you will need to make good use of both memory and storage. As you will see when we start talking about variables later in the course, a variable represents a place in memory where you can store a value. That means when you create a variable, you are actually reserving a space in RAM to hold a value. No other part of the computer can use that space. That's pretty powerful, huh? You will want to make good use of this space and not take up any more than you really need. This is because, as you know, there is a limited amount of space and all of the computer's programs are using that area. If you declare a large number of variables for no reason at all, you are taking away resources from the rest of the computer. That may mean that your computer will slow down or even crash. However, most computers now come with such a large amount of memory that you are not likely to crash the computer by declaring too many variables.

However, people are starting to write programs for their personal digital assistants (PDAs), and these devices do not have anywhere near as much memory as desktop computers or even laptops. Therefore, it is good practice to get used to reserving as little memory as possible.

You will also need to use storage when you are writing your programs. First of all, you will want to save your programs to a location where you can get access to them at a later point in time. There may also be times when you do not only want to send the results of your program to the screen, but you will also want to send them to a file for storage.

So, you might be wondering where the calculations are performed. That would be in the *central processing unit* (CPU). The CPU does all of the mathematical and logical operations that are required. This unit is broken into two sections: the arithmetic and logic section and the control section. The *arithmetic and logic unit* (ALU) performs functions like adding, subtracting, multiplying, and dividing. Comparison operations are also performed in this area of the CPU. The control section of the CPU coordinates all of the processes that are currently taking place on the computer system.

A computer program, known as *software*, regulates every activity on a computer system. Let's say you were writing a letter to one of your best friends using a word processor. You may have guessed that the word-processing application is software. But are there any other pieces of software running at the same time? You might first think there aren't. But looking closer at your computer, you will see that there are many pieces of software running at the same time.

Just think about pressing the ENTER key. When you press the key on the keyboard, a signal is sent from the keyboard. Since the ENTER key is different from every other key on the keyboard, the keyboard must be sure to send the correct signal. This is done with the keyboard's software. Next, that signal is received by the CPU. The CPU now must determine what to do with that signal. That is taken care of with the CPU's software. When the CPU determines what to do, a new message is sent to the word-processing program. That program says that the cursor should be moved to the next line. Therefore, a signal is sent to the monitor to tell it to advance the cursor to the next line. Before that signal gets to the monitor, it must be interpreted by the monitor's software so that it is displayed correctly.

Wow, that is a lot of work just for pressing one key. Imagine how much work is done when you type in the entire letter, save it, and print it out on your printer! The reason computers are so powerful is not because they are so smart. Remember, they can just do a few simple things; however, they can do them over and over really fast. You will find as you learn to program that there are very few different instructions; the power comes in your ability to put those instructions together in a meaningful way.

Also, we will not be writing programs to tell the hardware components how to interact with one another. Instead, we will be writing programs that take raw data the user will enter from the keyboard and convert it into useful information that can be displayed on the monitor.  
  
  
  
  
**Chapter 3**

**Algorithms and Flowcharts**

Now that you have a general idea of what a computer is and why programs are so important to them, I next want to point your attention to problem solving. Every computer program is a solution to some problem. I mentioned earlier that there are only a few different instructions that the computer understands. It is the arrangement of these instructions that makes your program useful. However, very few people are able to sit down and write a computer program without doing any planning. That is where problem-solving techniques come in. You will want to list the steps your program should perform. This is called *making an algorithm*. An *algorithm* is a series of steps used to solve a given problem. A recipe for cooking pasta is an algorithm:

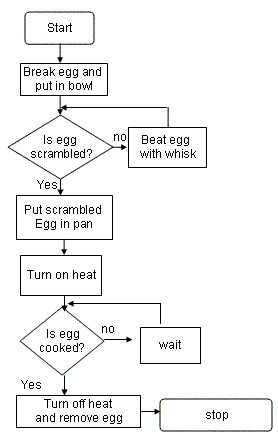
* Put water in the pot.
* Turn the heat on.
* Place the pot over the heat.
* Wait for the water to boil.
* Add the pasta to the water.

Each of the steps and the order in which you do them is very important. Imagine if you did all of the above steps in order, except that you decided not to put the water in the pot. What would happen to the pasta? I'm not a great cook, but I do know that you can't cook the pasta without the water.

People write algorithms in two ways. One method is flowcharting. A *flowchart* is a representation of the logical steps of a program drawn as a picture. There is a common set of symbols, each representing a different programming structure. They are as follows:

|  |  |  |
| --- | --- | --- |
| **Table 1.1. Programming symbols** | | |
| **Symbol** | **Name** | **Description** |
| https://api.ed2go.com/CourseBuilder/2.0/images/resources/prod/cpb-0/L01-01.gif | Process | Used for things like declaring variables, doing mathematical operations. |
| https://api.ed2go.com/CourseBuilder/2.0/images/resources/prod/cpb-0/L01-02.gif | Input/Output | Used for any type of communication between the user and the computer. |
| https://api.ed2go.com/CourseBuilder/2.0/images/resources/prod/cpb-0/L01-03.gif | Decision | Used when the computer must decide whether to execute a set of instructions. |
| https://api.ed2go.com/CourseBuilder/2.0/images/resources/prod/cpb-0/L01-04.gif | Terminal | Used for the start and end of the program only. |
| https://api.ed2go.com/CourseBuilder/2.0/images/resources/prod/cpb-0/L01-05.gif | Flowline | Used to show the direction of flow for the program. |
| https://api.ed2go.com/CourseBuilder/2.0/images/resources/prod/cpb-0/L01-06.gif | Predefined Process | This symbol represents a series of code statements that are defined somewhere else. Used to represent methods or subroutines. |

Inside each symbol, you will write the step that you want to perform. In general, flowcharts should start in the upper left corner of your sheet of paper and progress either down or to the right. Many people like to draw flowcharts because they are a graphic representation of the logic of the program. That is, they graphically show each step in the algorithm. Another benefit is that you can put your finger on the exact step at which you are located in the algorithm and know exactly what the next step is. The example below shows a flowchart of how to cook a scrambled egg:

  
Fig. 1.1. Flowchart for scrambling an egg

While flowcharting is an excellent way for graphical people to picture their algorithm, another common way to plan out your program is to use pseudocode.

*Pseudocode* is a method of using English-like phrases to describe the steps. I often think of pseudocode as the statements that would be placed inside of the flowchart symbols. You will place each statement on a new line on your sheet of paper. It is customary to indent all steps that will be inside of a decision or a loop. This makes it easier to identify which steps are part of the decision or loop and which ones are not. The following is the same algorithm for cooking a scrambled egg, only this time using pseudocode:

Start

Break the egg and put in a bowl

Repeat if the egg is not scrambled

Beat the egg with a whisk

Put the scrambled egg in a pan

Turn on the heat

Repeat if the egg is not cooked

Wait

Turn off the heat and remove egg

End

As you can see, both the flowchart and pseudocode do an excellent job representing the steps that need to be done in order to cook a scrambled egg. Actually, either tool works just fine to plan any algorithm. You may be wondering which one you should use. Well, give each of them a try and see which one is more comfortable and natural to you. You may come to find that some problems are easier to plan using flowcharts, while others may be easier using pseudocode.

**Chapter 4**

**Programming Languages**

Before we start discussing the different programming languages, I want you to think again about how a computer works. Computers are machines that run on electricity. They communicate and store their information in microscopic electronic switches that have their power either on or off. To make it more convenient, humans think of these letters as zero (off) and one (on). The term we give to each individual switch that holds a zero or one is a *bit*, short for binary digit. Computers typically group eight of these bits together, known as a *byte*, to store our English characters.

This knowledge of how computers communicate is important, because if we want to give our computers instructions, then we had better learn how to speak in zeros and ones. This is exactly how early programmers programmed—in zeros and ones. These programs were said to be written in *machine language*. Below is an example of a machine language instruction used to load a value into a memory location.

000000 10001 10010 01000 00000 100000

Notice that there are a series of 32 numbers above, or 4 bytes. You may also notice that I have grouped the numbers in a certain way. Programmers found it easier to group the numbers in a certain way to make the instructions easier to read. By the way, please recall that this is one instruction. This particular instruction adds the values in two different places in memory and stores the result in another. Programs required many lines of code to do anything significant.

Another drawback of machine languages was that they were machine dependent. That meant that a program written on one type of computer could not be run on a different type of computer. We still have this problem with today's computers. That is, you typically cannot run programs that were written for Windows on a UNIX machine. Can you imagine writing a sophisticated program like a word processor in machine language?

The early programmers quickly realized these problems with machine language, so they developed a method to write programs using mnemonics. *Mnemonics* are just a way to help people remember things. This new set of mnemonics for the programmer is known as *assembly language*. Now, instead of having to remember the correct order of all the zeros and ones for an instruction, programmers could use words like *ADD* or *MUL* for doing things like adding numbers or multiplying numbers, respectively. The machine code list above became:

add $t0, $s1, $s2

The programmers would write their programs as assembly language instructions and then send these instructions to another program called an *assembler*. The assembler would take each of these instructions and translate them from the mnemonics into machine code. The result was a set of instructions that the computer could understand.

Although the mnemonics of assembly language were a help to programmers, it was still difficult to easily communicate with computers. The programmers needed to write out every small task that they needed the computer to do, and they had to know a language that was somewhat cryptic. The creation of *high-level languages* allowed programmers to write instructions that more closely resembled English. The best example of this can be seen in a segment of BASIC code.

PRINT "Pick a number between 1 and 10"  
INPUT first  
PRINT "Pick a number between 1 and 10"  
INPUT second  
sum = first + second  
PRINT first; " plus "; second; " equals "; sum  
END

You may be able to see that this program will ask the user to enter two numbers, store those values, and calculate the sum. Finally, the program prints a message to the user telling them the sum of the two numbers. While these are definitely not English sentences, it is not difficult to guess what the program is doing by just reading the words.

The creation of these languages made it possible for average people to start writing computer programs more easily. Students ranging in ages from college to junior high school began learning how to write computer programs. These students were then able to create very useful programs with just a few different commands.

The original high-level languages—BASIC, COBOL, FORTRAN, and C—allowed programmers to write *procedure-oriented programs*. These programs were written with an emphasis on how the task was to be accomplished. The instructions were written in a step-by-step fashion, from the beginning of the task to its completion. The programmer would control the order in which the computer processed the instructions. This order is very important. Using the code above as an example, you will notice that the programmer could not calculate the sum of the two numbers if the user has not been given a chance to enter them first.

To plan the solution to problems in a procedure-oriented language, programmers typically use what is called *top-down design* to assist them in planning their program. This methodology begins with the programmer writing a single statement that describes the overall goal of the program. The programmer next determines what needs to be done, but not necessarily how to do it. After working through Lesson 1, this should sound familiar to you. This is where the programmer will write out the pseudocode or flowchart for the program. The final step in top-down design is to explicitly state how the program will accomplish each of its tasks.

Many programmers agree that it is easiest to break larger programs into smaller, manageable tasks. Using the example above you will notice that the program has three parts: get the data from the user, calculate the sum, and display the result. You will find that if you can break the problem into the smaller pieces, programming becomes much easier. This is because programming at this point becomes simple translation. That is, you are just translating simple tasks such as *get a number from the user* into the code *INPUT first*.

High-level languages allow the programmers to write code that looks more similar to English, but how does the computer understand this code? The answer is a compiler. A *compiler* is a computer program that is similar to an assembler. In a similar way, a compiler is a computer program that takes a high-level language and translates it into machine language. This ability for programmers to use these high-level languages to write programs led to many people writing computer programs without a tremendous amount of technical know-how.

Programming in procedure-oriented languages is a natural way to organize your code. But these programs usually are simple command prompt programs. That is, these programs run at the command prompt and display white text on a black background. There are no graphics, buttons, menus, or check boxes like we are used to seeing in our programs today. These programs are only able to ask simple questions and display their results in simple tables. A second problem with the procedure-oriented approach is that these programs use a tremendous amount of redundant code. Programmers found themselves rewriting the same code over and over in completely different programs.

The development of *object-oriented programming* allowed the programmer to think of their programs as a number of objects that interact with one another. This concept was taken from the world in which we live. Look around you right now. Everything you see is an object. Each object can be described by a set of characteristics and a set of behaviors. For example, think about a pencil. Your pencil may have the characteristics of being yellow with a red eraser on one end. This pencil probably has the behavior that if you press the lead end onto a sheet of paper and move the pencil, it will draw a mark on the paper. If you turn the pencil around and use the eraser end, it will remove any marks that were made on that paper. I agree that this is a very simple way to look at the world, but simple is good, right?

You might again be thinking, "How does this apply to programming?" Well, think about a program that you use all the time. No doubt at some point when you are using the program, you will have to click on a button. Have you ever noticed that all buttons look roughly the same? Each button usually has a three-dimensional appearance, and it has some text written inside of it. These are the button's characteristics. In addition, every button has the behavior that, when the button is clicked, the program will perform some task.

The exciting thing about object-oriented programming is that it does not limit us to creating objects like buttons and text boxes. The fact is that we are able to write a blueprint for any object that we wish to represent. Take, for example, a bank account. Every bank account has an account name, an account number, and a balance. These three things would become our account object's characteristics. In addition, each account will need to have a number of behaviors. An example of a behavior for an account, my favorite, would be to withdraw money from the account. This behavior would need to provide a way to check that there is enough money in the account and then actually remove that amount from the account. You can also imagine that there would be behaviors to deposit money, create a new account, or even change the name on the account.

This ability for the programmer to focus on objects makes programming easier. First, it allows the programmer to write code that models things we interact with on an everyday basis—things like bank accounts, dates, and cars, to name a few. It also allows the programmer to write code to create an object in one program that can be used in another program. For example, can you imagine writing the description for an object that models a deck of cards? Well, once you actually write the code to create this object, you can then use it in a program that simulates poker or solitaire or any other card game. That means that you now have an easier way to reuse code that you have written.

As you may have guessed, object-oriented languages are high-level languages. That means that the code looks similar to English. It also means that the programmer needs to use a compiler to translate the code into machine language so that the computer can understand it. Something that you may find interesting about object-oriented languages is that many are direct descendents of earlier procedure-oriented languages. Visual Basic was developed based on BASIC. In fact, the syntax of the two languages is almost identical. Similarly, the C# programming language (object-oriented) is the next generation of C++ (object-oriented), which is based on C (procedure-oriented).  
  
  
  
  
**Chapter 5**

**Summary**

In this lesson, you have explored the components of a computer and perhaps thought about them in a different light. You now see that each part of a computer must interact with the other parts, and they do this through computer programs. These programs have been written by others to make our lives easier. In this course, we will be writing software, computer programs developed to accomplish a specific task. We will learn how to write programs to perform redundant calculations quickly and with great precision.

In addition to learning about the parts that make up a computer, you also learned two different problem-solving techniques. Most large computer programs are complex in nature, so it is very important to plan out what you are going to do before you start doing it. This will give you a better idea of how to proceed and should help you to sort out any problems you may have with your logic before you start to worry about how to get the computer to do these tasks. Remember, it is very easy to translate from one language to another; the greater difficulty comes when you are planning the logic. Flowcharts and pseudocode are excellent planning tools that both work very well.

Finally, this lesson took a brief look at the history of programming languages. You saw how programmers went from writing code in zeros and ones to thinking about the program as a series of interacting objects.

In the next lesson, you will learn a little bit about the C# programming language and what makes it unique. You will also learn where you can get a free copy of the C# compiler and how to install it. Best of all, the next lesson will have you sitting down to the computer to write your first C# program!  
  
  
  
**Supplementary Material**

|  |
| --- |
| [How to Write Pseudocode](http://www.minich.com/education/psu/cplusplus/stylesheets/pseudocode.htm)  http://www.minich.com/education/psu/cplusplus/stylesheets/pseudocode.htm |
| Although there are no hard rules for writing pseudocode, this document gives some good direction on how to write good pseudocode. |
| [Computer Languages History](http://www.levenez.com/lang/)  http://www.levenez.com/lang/ |
| This site makes an attempt to list every computer language ever created and plots them on a timeline. It also provides links to each language, where a description of that language exists |
| [Visual C# .NET Express Edition](http://www.microsoft.com/express/vcsharp/)  http://www.microsoft.com/express/vcsharp/ |
| This page will allow you to download the Express Edition (which is free) of Microsoft's Visual C#. This download includes the C# compiler along with an Integrated Development Environment (IDE). We'll be using the compiler for this course but NOT this or any other IDE.   If you choose to install this download, you will NOT need to download the "Redistributable Package." |
| [Microsoft .NET Framework 3.5 Redistributable Package](http://www.microsoft.com/downloads/details.aspx?FamilyId=333325FD-AE52-4E35-B531-508D977D32A6&displaylang=en)  http://www.microsoft.com/downloads/details.aspx?FamilyId=333325FD-AE52-4E35-B531-508D977D32A6&displaylang=en |
| If you have some extra time, you may want to follow this link to the Microsoft site where you can download the C# compiler (and other things), if you have not already done so. This "Redistributable Package" will contain all of the files necessary to compile the C# programs for this course.  **Note**: Please be sure that you run Windows Update (**Start** > **All Programs** > **Windows Update**) on your machine before installing the Redistributable Package. |

**FAQs**   
  
**Q:** All of this material seems very technical for a course for 'the absolute beginner.' Am I going to have to know everything about how my computer works in order to learn how to write programs?  
  
**A:** Please do not panic. The reason for all of the technical material in Lesson 1 was to show you how important programming is to a computer. The programs we will write will start off very simply and will never reach the complexity of the examples given in the lesson.  
  
  
**Q:** I find it very easy to write the pseudocode, but I just can't figure out how to draw the flowcharts. Is that okay?  
  
**A:** Certainly. Remember, you are free to use either a flowchart or pseudocode. The important thing is that you plan what you are going to do before you start working with the computer. This will make your job of programming much easier.  
  
  
**Q:** Will we need to know how a compiler translates the code we write into machine language that the computer can understand?  
  
**A:** While some people find the inner workings of compilers to be very interesting, knowledge of how they do what they do will not be necessary for this course.  
  
  
**Q:** When I try to run my program by typing the name of the class at the command prompt, I get an error saying that this file does not exist, but I know my source code file is there. What is wrong?  
  
**A:** Remember, your source code file is the one that you wrote. The computer will need a file written in a language that it can understand. That is the reason why we use the compiler. Chances are that you did not successfully compile your program, so no executable was created.  
  
  
**Q:** Are we required to submit our Assignments to you for grading? If so, how/where do we do that?  
  
**A:** There is no requirement, and really no mechanism, for you to submit your work to me for grading. Instead, these exercises are sort of self-grading; your programs will either work, or they won't. The only requirement for you to get your *Completion Certificate* is to pass the Final Exam at the end of the course.  
  
Actually you'll find as you get into the course that I've included *sample solutions* along the way. Please do not view these programs as the way I expect you to solve the problems, just one possible way. Many times I learn so much by viewing other people's code, which is why I've included these programs.

**Assignment**   
  
  
Write out an algorithm for how to tie your shoes.

First, do this by drawing a flowchart.

When you are satisfied that your algorithm is correct, give it to someone else and see if they can use your directions to tie their shoes.

Next, do the same thing using pseudocode (English-like statements).

Which one was easier for you? Which one was easier for your user?

|  |
| --- |
| Copyright © 1997 - 2012 Education To Go. All rights reserved. The material on this site cannot be reproduced or redistributed unless you have obtained prior written permission from Education To Go. Education To Go and ed2go are registered trademarks of Education To Go, a part of Cengage Learning. |

Bottom of Form