CHAPTER 5

VARIABLES AND OTHER BASIC ELEMENTS IN C# PROGRAMS

Now, finally, we really begin studying computer programming with the C# language. Variables are important elements in computer programs, and they are needed in almost every computer program. In this chapter we shall study different types of variables and use the variables in simple arithmetic computations. We shall also explore names and keywords which are very basic elements in source programs.

This chapter, as well as the following chapters, introduce many examples of C# source programs for you to study. A source program is a textual description of what the computer should do when the compiled version of the source program is executed by a computer. The source programs that you will see in this chapter contain variable declarations followed by executable program statements, action statements, which do something with the variables. The general structure of the programs is the following:

```
using System ;

class ClassName
{
   static void Main()
   {
      Variable declarations.

      Action statements that modify and print
      the contents of the variables that
      were declared above.
   }
}
```

As was already discussed in Chapter 2, the structure of our first programs is such that a static method named **`Main()`** contains the action statements of the program, and the **`Main()`** method is written inside a class declaration. At this phase, we do not attempt to profoundly understand the meaning of the class declaration. We'll just try to figure out how the internal statements of method **`Main()`** operate.

5.1 Integer variables (int, short, long, byte, uint, ushort, ulong, sbyte, char)

A variable in a source program is a basic program element that can be used to store numerical values. The value of a variable usually changes when the program is being executed. When we declare a variable in a program, we actually reserve a few bytes of computer's main memory to be used for a special purpose. The following is an example of a variable declaration:

```
int  integer_from_keyboard ;
```

This source program line which introduces a variable into a program can also be called a variable definition. The above source program line means that four bytes (32 bits) of memory are reserved to store an integer that will be read from the keyboard, and these four bytes can be referred to with the name **`integer_from_keyboard`**. Integers are whole numbers that have no decimal point. Integers can be positive or negative. Variables of type **`int`** are said to be 32-bit variables because they occupy four bytes in the main memory of the computer

A variable always has a type, such as **`int`** which is an abbreviation of the word "integer". The programmer, the person who writes the variable declarations in a program, must give a unique name to each variable. In the declaration above, the name of the variable is **`integer_from_keyboard`**. Variable declarations, like all C# statements, must be terminated with a semicolon ;.

Program **Game.cs**, presented as a program description on the following page opening, is an example program where two variables of type **`int`** are declared and used. The program is an extremely simple computer game. Unfortunately it is not a fair game because the user of the program will always lose. The program always wins by presenting an integer that is one larger than the number given by the user of the program.

A source program like **Game.cs** is a text file in a computer's hard disk memory before it is compiled. When a source program is compiled, we get an executable version of the program. The compiler is a computer tool that can convert a source program into executable form. The compiler reads and processes a source program file in the same order in which it is written. While compiling program **Game.cs**, the compiler sees first the variable declarations and reserves main memory for the variables. It then transforms the remaining statements, the action statements, to numerical instructions to be processed during program execution.

The action statements in a source program describe the activities a computer performs when the executable version of the program is run by a computer. The action statements of a program are executed in an order that corresponds with the order in which the statements are written in the source program. In the case of the program **Game.cs**, the computer performs the following activities:

- First it asks the user to enter an integer from the keyboard.
- It then reads the integer entered from the keyboard and stores it in variable **`integer_from_keyboard`**.
- Then it calculates a value that is one larger than the user-given integer and stores that value in variable **`one_larger_integer`**.
- In the end, it displays the values of both variables and informs the user that the computer won the game.

There are four action statements in **Game.cs**. Each statement is terminated with a semicolon (;). The variable declarations at the beginning are also statements, but they are not action statements. Variable declarations just reserve memory to store information.

Although the memory space that is reserved for an `int` variable is rather large, 4 bytes, there are always limitations how large values can be stored in an `int` variable. A 4-byte `int` variable can store values in the ranges

-2,147,483,648, ... , -1, 0, 1, ... , 2,147,483,647 (decimal numbering system)

-80000000H, ..., -1, 0, 1, ... , 7FFFFFFFH (hexadecimal numbering system)

A 4-byte `int` can thus store 4,294,967,296 (100000000H) different values. To demonstrate the difficulties that arise when the storage capacity of an `int` variable is exceeded, program **Game.cs** is also executed with an exceedingly large input value in the program description. The computer tries to increment the value 2,147,483,647 which is stored in a 4-byte `int` variable. This results in the number -2,147,483,648 and not in 2,147,483,648. To explain this strange behavior of the program, we must remember that the memories of computers can contain only non-negative binary numbers. Negative numbers are represented so that some positive values stored in memory are considered as negative numbers. For example, the value 2,147,483,648 is treated as -2,147,483,648. The values that can be contained in 4-byte `int` variables have the following meanings:

```
VALUE IN MEMORY                  MEANING IN PROGRAM

2,147,483,648 (80000000H)        -2,147,483,648
2,147,483,649 (80000001H)        -2,147,483,647
2,147,483,650 (80000002H)        -2,147,483,646
.                                 .
.                                 .
4,294,967,294 (FFFFFFFEH)        -2
4,294,967,295 (FFFFFFFFH)        -1
0                                 0
1                                 1
.                                 .
.                                 .
2,147,483,646 (7FFFFFFEH)         2,147,483,646
2,147,483,647 (7FFFFFFFH)         2,147,483,647
```

Figure 5-1 shows how the variables of program **Game.cs** look like in the main memory of a computer, and how the values of the variables change when the program is executed with input value 1234. A variable declaration like

```
int  integer_from_keyboard ;
```

reserves four bytes from contiguous memory locations somewhere in the main memory. Right after the declaration of the variable, the contents of the four bytes are unspecified. After an assignment statement like

```
integer_from_keyboard  = Convert.ToInt32(
                                         Console.ReadLine() ) ;
```

is executed, the four bytes are given values that represent the number that was typed in from the keyboard.

The illustration in Figure 5-1 shows the general principle according to which memory is reserved for variables. The compiler may, however, optimize the use of memory if it finds out that it can save memory.

This line is not actually part of the program. This is a comment line that gives documentary information to the reader of the program. A double slash // marks the beginning of a comment line. The compiler ignores the double slash and the text that follows it on the same line.

Here two integer variables are declared. The names of the variables are **`integer_from_keyboard`** and **`one_larger_integer`**. The variables are referred to with these names later in the program.

```
//  Game.cs  (c) 2002 Kari Laitinen

using System ;

class Game
{
   static void Main()
   {
      int  integer_from_keyboard ;
      int  one_larger_integer ;

      Console.Write(
          "\n This program is a computer game. Please, type in "
        + "\n an integer in the range  1 ... 2147483646 : " ) ;

      integer_from_keyboard  =  Convert.ToInt32( Console.ReadLine() ) ;

      one_larger_integer  =  integer_from_keyboard  +  1 ;

      Console.Write( "\n You typed in " + integer_from_keyboard + "."
                  +  "\n My number is " + one_larger_integer    + "."
                  +  "\n Sorry, you lost. I won. The game is over.\n") ;
   }
}
```

This line of source code reads an integer from the keyboard and stores the read integer into variable **`integer_from_keyboard`**. The execution of the program stays on this line until the user of the program has entered an integer. This statement is executed so that first the **`ReadLine()`** method reads a line of text from the keyboard, and then the text is converted to an integer value with the **`ToInt32()`** method.

Texts inside double quote characters " " are strings of characters that will be displayed on the screen. \n among the text means that the text will begin from a new line. \n is said to be the newline character.

Game.cs - 1.+ A program that implements a simple computer game.

`Console` is a standard C# class that contains methods for writing text to the screen and reading text from the keyboard. The method named `Write()` can write text to the screen. The text that is going to be displayed on the screen consists of two strings of characters. These two character strings are concatenated with operator +, and the text is given inside parentheses to method `Write()`.

After this assignment statement has been executed, the value of variable `one_larger_integer` is one greater than the value stored in `integer_from_keyboard`.

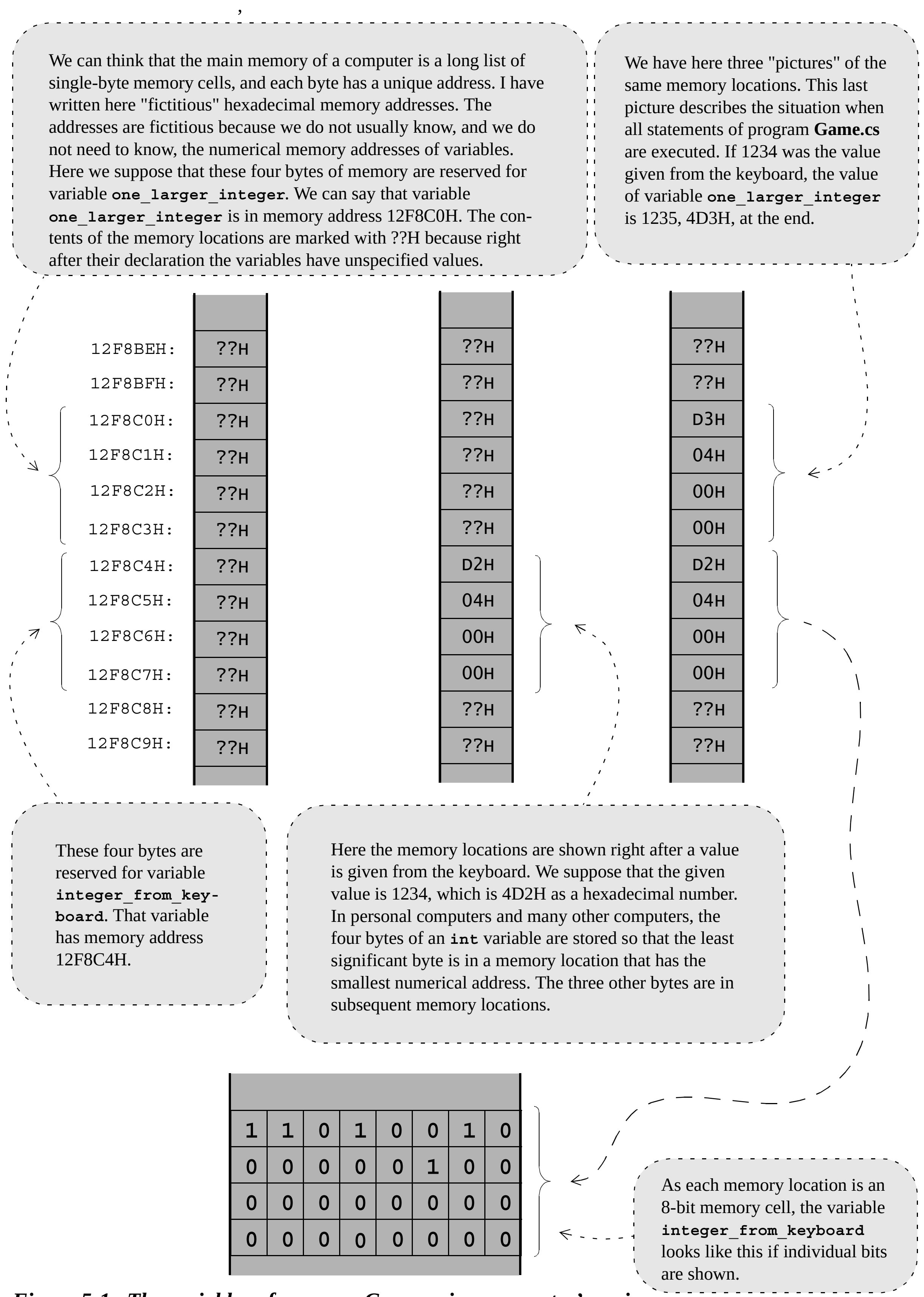
```
      Console.Write(
          "\n This program is a computer game. Please, type in "
        + "\n an integer in the range  1 ... 2147483646 : " ) ;

      integer_from_keyboard  =  Convert.ToInt32( Console.ReadLine() ) ;

      one_larger_integer  =  integer_from_keyboard  +  1 ;

      Console.Write( "\n You typed in " + integer_from_keyboard + "."
                +  "\n My number is " + one_larger_integer    + "."
                +  "\n Sorry, you lost. I won. The game is over.\n") ;
```

It is possible to output many types of data in a single call to method `Write()`. Here the values of the integer variables are displayed between strings of characters given inside double quotes. Operator + is placed between different types of data. The + operator converts the numerical values stored in the variables to character strings, and joins these character strings to the other character strings given inside double quotes. A semicolon (;) terminates the entire statement.

Game.cs - 1 - 1. The action statements of the program.

```
D:\csfiles2>Game

 This program is a computer game. Please, type in
 an integer in the range  1 ... 2147483646 : 1234

 You typed in 1234.
 My number is 1235.
 Sorry, you lost. I won. The game is over.

D:\csfiles2>Game

 This program is a computer game. Please, type in
 an integer in the range  1 ... 2147483646 : 2147483647

 You typed in 2147483647.
 My number is -2147483648.
 Sorry, you lost. I won. The game is over.
```

Game.cs - X. In the second execution too large an input value is given to the program.


We can think that the main memory of a computer is a long list of single-byte memory cells, and each byte has a unique address. I have written here "fictitious" hexadecimal memory addresses. The addresses are fictitious because we do not usually know, and we do not need to know, the numerical memory addresses of variables. Here we suppose that these four bytes of memory are reserved for variable **`one_larger_integer`**. We can say that variable **`one_larger_integer`** is in memory address 12F8C0H. The contents of the memory locations are marked with ??H because right after their declaration the variables have unspecified values.

We have here three "pictures" of the same memory locations. This last picture describes the situation when all statements of program **Game.cs** are executed. If 1234 was the value given from the keyboard, the value of variable **`one_larger_integer`** is 1235, 4D3H, at the end.

These four bytes are reserved for variable **`integer_from_key-board`**. That variable has memory address 12F8C4H.

Here the memory locations are shown right after a value is given from the keyboard. We suppose that the given value is 1234, which is 4D2H as a hexadecimal number. In personal computers and many other computers, the four bytes of an **`int`** variable are stored so that the least significant byte is in a memory location that has the smallest numerical address. The three other bytes are in subsequent memory locations.

As each memory location is an 8-bit memory cell, the variable **`integer_from_keyboard`** looks like this if individual bits are shown.

Figure 5-1. The variables of program Game.cs in a computer's main memory.

In addition to variables of type **`int`**, C# has other types of integer variables. These variables are sometimes called integral types. Type **`long`** is another integer type. Variables of type **`long`** occupy 8 bytes (64 bits) of memory. They are thus longer integer variables than variables of type **`int`**. If we declared the variables of program **Game.cs** as

```
long  integer_from_keyboard ;
long  one_larger_integer ;
```

the program would work correctly with much larger numbers as it does now.

A third integer variable type is **`short`** which is a 2-byte, 16-bit, variable. This type may be useful when your program uses only integer values that are smaller than 32,767. If you need to store very many such values, you can save some memory by using type **`short`** instead of type **`int`**. In small programs, though, there is no need to use variables of type **`short`** because memory is rather abundant in modern computers.

In many cases, negative integers are not needed in source programs. In such situations, integer variables can be declared to be non-negative. C# has variable types **`uint`**, **`ulong`**, and **`ushort`** to declare "unsigned" non-negative integer variables. A variable of type **`uint`**, for example, is similar to an **`int`** variable except that it cannot have any negative values. Instead, the variable can have larger positive values. If the variables in program **Game.cs** were declared

```
uint  integer_from_keyboard ;
uint  one_larger_integer ;
```

there would be no difficulties in running the program with input values up to 4,294,967,294.

`byte` is a variable type to store non-negative 8-bit values in the range 0 ... 255. A variable of type **`byte`** occupies a single byte of memory. **`sbyte`** is a variable type that is similar to the **`byte`** type. The difference between these types is that **`sbyte`** variables can contain negative values.

Variables of type **`char`** are 2-byte (16-bit) variables that are used to store 16-bit Unicode character codes. The first 256 Unicodes are the same as the codes in the ASCII coding system. **`char`** variables are like integer variables because the used character codes are integer values. However, it is better not to use **`char`** variables in situations where integer variables are needed.

The integer types **`int`**, **`short`**, **`long`**, etc. will be used when we need to store whole numbers in our computer programs. The most common integer type is **`int`**. Integer variables are convenient when we want to count something in our programs. Table 5-1 summarizes the integer types and other variable types in C#. Figure 5-2 shows how some variables look like in the main memory of a computer. The other variable types will be discussed later in this chapter. Type **`bool`** is a special variable type to declare so-called boolean variables. Boolean variables can be given only two values: **`true`** and **`false`**. We shall study these variables in more detail when we encounter them in some example programs.

Exercises with program Game.cs

Exercise 5-1.	Modify the program so that it always loses the game by presenting a number which is one smaller than the number given by the user. With operator - it is possible to perform subtractions in C#.
Exercise 5-2.	Modify the program so that it prints three numbers that are larger than the number given by the user. For example, if the user types in 144, the program must print 145, 146, and 147.