# Content

[Light Mode](https://timmcmichael.github.io/skimpy-csharp/_0010_variables_and_data_types.html)  |  [Dark Mode (experimental)](https://timmcmichael.github.io/skimpy-csharp-dark/_0010_variables_and_data_types.html)  |  [PDF (experimental)](index.pdf)

# **Skimpy C#: An Introduction to Programming**

Tim McMichael[tim.mcmichael@estrellamountain.edu](mailto:tim.mcmichael@estrellamountain.edu)

Skimpy Contents

[Skimpy C#: An Introduction to Programming](index.html)

* [Skimpy Preface](_skimpy_preface.html)
* [How to Use This Book](_how_to_use_this_book.html)
* [Related Resources](_related_resources.html)
* [0. 0000 - Getting Started](_0_0000_getting_started.html)
* [1. 0001 - Computers and Coding](_0001_computers_and_coding.html)
* [**2. 0010 - Variables and Data Types**](_0010_variables_and_data_types.html)
  + [What’s the Point?](_0010_variables_and_data_types.html#_whats_the_point_3)
  + [2.1. C# Identifiers](_0010_variables_and_data_types.html#_c_identifiers)
  + [2.2. Numeric Data Types](_0010_variables_and_data_types.html#_numeric_data_types)
  + [2.3. bool Data Type (and Boolean Logic)](_0010_variables_and_data_types.html#_bool_data_type_and_boolean_logic)
  + [2.4. char Data Type](_0010_variables_and_data_types.html#_char_data_type)
  + [2.5. Strings](_0010_variables_and_data_types.html#_strings)
  + [2.6. Declaring Constants](_0010_variables_and_data_types.html#_declaring_constants)
  + [2.7. Outputting Variable (and Constant) Values](_0010_variables_and_data_types.html#_outputting_variable_and_constant_values)
  + [2.8. Math Calculations](_0010_variables_and_data_types.html#_math_calculations)
  + [2.9. Handling User Input](_0010_variables_and_data_types.html#_handling_user_input)
  + [Check Yourself Before You Wreck Yourself (on the assignments)](_0010_variables_and_data_types.html#_check_yourself_before_you_wreck_yourself_on_the_assignments_3)
* [3. 0011 - Methods](_0011_methods.html)
* [4. 0100 - Decisions](_0100_decisions.html)
* [5. 0101 - Loops](_0101_loops.html)
* [6. 0110 - Debugging and Generative AI](_0110_debugging_and_generative_ai.html)
* [7. 0111 - Classes and Objects](_0111_classes_and_objects.html)
* [8. 1000 - Arrays](_1000_arrays.html)
* [9. 1001 - Software Development Life Cycle](_1001_software_development_life_cycle.html)
* [10. 1010 - Graphical User Interfaces](_1010_graphical_user_interfaces.html)
* [11. 1011 - [TOPIC]]](_1011_topic.html)
* [12. Stuff That’s Tacked on the End](_stuff_thats_tacked_on_the_end.html)

## **2. 0010 - Variables and Data Types**

Help Make These Materials Better!

I am actively working to complete and revise this eBook and the accompanying videos. Please consider using the following link to provide feedback and notify me of typos, mistakes, and suggestions for either the eBook or videos:

[CIS162AD Course Materials Feedback (Google Form)](https://forms.gle/H9VgmhJkQcKx9iRi8)

This is the "pre-release" working draft of this **very incomplete** eBook. Content will change as chapters are developed and refined. Check back later for a more complete version.

### **What’s the Point?**

\* \*

Source code examples from this chapter and associated videos are available on [*GitHub*](https://github.com/timmcmichael/EMCCTimFiles/tree/main/Intro%20to%20C-Sharp%20(CIS162AD)/02%20Variables).

It’s fun to create programs that display stuff on the screen, but that’s pretty limited. To start doing things that are more interesting—​and useful—​we’ll need to keep track of some information. In computer programming, a piece of information is called **data**.

In broad terms, there are two kinds of data:

**constant**

A piece of data that can’t change during program execution.

**variable**

A piece of data that may change (or "vary") during program execution.

In our Hello World code, the letters and characters we output are constant data.

*Example 2.6 -*HelloWorld*in C#*

Console.WriteLin("Hello World!");

The only way to change what’s output is by editing that constant—​in the quotation marks—​in our source code, before we run the program.

|  |  |
| --- | --- |
|  | The string literal in our Hello World code is just one kind of constant. We’ll learn about others later. |

In order for our programs to be flexible and responsive, we’ll need **variables**. I encourage beginning students to think of a variable like a container, such as a cup.

*Figure 2.1 - A photograph of the****actual****mug used in lectures.*

The container can hold one thing at a time. We can put coffee in it, or we can put water in it—​but we can’t put both. If we start with coffee and decide we want to put water in there, the coffee gets dumped out and is gone forever. That’s fine, but we better make sure we don’t want the coffee anymore before we put something else in the mug.

A variable works the same way. It can hold one value, and that value can vary—​remember, that’s where the name comes from—​but once it’s changed, that old value is lost.

I guess we could mix coffee and water, but then that’s really a new substance and we can’t get back our plain coffee (or our water). If we want to store coffee **and** water, we can do that by using two different containers.

From a programming perspective—​rather than a coffee perspective—​a variable is a location in the computer’s memory where we can store a value. In order to use that value, we have to know where to find it, so we give the memory location a name, technically called an **identifier**.

*Figure 2.2 - A really great quarterback*

And the computer will need to know how much memory to set aside for that variable, which depends on the kinds of values we want to store. Storing the number 7 doesn’t take very much space; storing a picture of John Elway takes a lot more space (but it’s worth it!). The **type** of data we can store in a variable is called its

…​drumroll, please…​

**data type**.

Reserving a memory location by specifying a data type and identifier is known as **declaring** a variable. Placing data in that variable is referred to as **assigning** a value to a variable.

|  |  |
| --- | --- |
|  | If you have programming experience in Python or JavaScript, the idea of a data type might be new to you. In those languages, the interpreter determines the data type for you—​so you don’t have to specify it. It’s one reason I prefer to teach beginners using C# or Java, which teaches you how data types work. |

Time To Watch!

Using Variables in C# [COMING SOON!]

### **2.1. C# Identifiers**

Variable names (more properly called variable identifiers) are the character sequences that identify a variable. There are a few rules that determine whether or not an identifier is valid:

* Can contain letters (upper or lower case), digits (0 - 9), and/or underscores (\_).
* The first character **can’t** be a digit.
* Can’t contain whitespaces (tab, space, etc.) or special characters (like @, #, $, etc.) except for the underscore.
* Can’t be a C# keyword or reserved word (words that are used in the C# language, like double or public)

In addition to those technical rules about identifier names, the convention in C# is to use descriptive names in "camel case" format, as described in the preceding video. Single-letter variable names, like x for a number, are fine in your math class, but are generally **not** okay when you’re coding. The expectation in my course is that your code will align with those conventions—​because that’s what industry people expect—​and you will lose points if you don’t.

### **2.2. Numeric Data Types**

Variables for storing numbers come in two flavors: **integers** and **floating-point** numbers. As you may remember from your math class, an integer is a whole number; that is, a number that doesn’t include any decimal places or fractional values. 5 is an integer, -824 is an integer, while 3.14 and 7 1/2 are not.

A floating-point number includes decimals, so 3.14 can be stored as a floating-point number. 7 1/2 can also be stored as a floating-point number, but only as a decimal (i.e., 7.5).

The most common numeric data types in C# are int for integers and double for floating-point numbers. You can get pretty far in programming just using those three, and in courses I teach you won’t need to use any other numeric data types—​but others do exist.

Other data types for integers are byte, short, and long. These different types exist because they use different amounts of memory. byte and short are smaller than int, while long is larger. The impact of these different memory sizes (or memory **allocations**) is that the types can store values of different sizes. For example, a byte uses 8 bits of memory and can store a number between -128 and 127 (inclusive), while an int takes 32 bits and can store a value between -2,147,483,648 and 2,147,483,647 (inclusive).

Interesting!

This all goes back to binary numbers. An 8-bit number has 8 digits, and we use the first digit to specify whether the number is positive or negative. That leaves 7 digits, and we can make 128 different combinations of 1s and 0s in 7 digits: 0000000, 0000001, 0000002, and so on, all the way up to 1111111. Those 128 possibilities give us the byte range of -128 to 127 (keeping in mind that we do need one of those combinations to represent zero).

Other floating-point data type in C# are float and decimal. 'float' is called "single" in some programming languages, which helps understand where the name double comes from: double uses twice as much memory (64 bits) as a float or "single" (32 bits)--and therefore its range of values is twice as big. Be aware that to make a float number in C#, you have to add the letter F (in capital or lowercase form) at the end of the number.

The decimal data type is slower and uses more memory than float or double, but it’s more precise. It’s used for financial calculations and other situations where precision is important.

*Example 2.7 - Numeric variable declarations in C#*

int myAge = 21;

float myGPA = 3.75f;

double myFriendsGPA = 3.54;

|  |  |
| --- | --- |
|  | The f suffix denotes that the value 3.75 is a float rather than a double. |

#### **2.2.1. Who Cares About Variable Size?**

The general rule in programming is to be as efficient with your resources (memory, storage, processing speed, network bandwidth, etc.) as possible. If you’re storing a person’s age, you don’t really need an int; nobody is going to be two million years old! A byte has plenty of room (up to 127) to store even the oldest person’s age, and it uses a fraction of the memory—​8 bits instead of 32. So I should be telling you to use a byte in this case.

And I should be insisting on decimal when we store money, because it’s more precise than double or float.

But I don’t worry about that stuff with beginning programmers for two reasons. First, it’s hard enough for a beginner to write programs that work—​so instead of asking you to deal with all of the numeric types, I just have you use int whenever you need a whole number, and double when you need something with a decimal. Second, these days even the wimpiest computer has waaaaaay more RAM than is needed by even the most complex program a beginner will write, so we don’t need to worry about it.

But understand that this attitude is only a teaching and learning aid. It’s like saying we shouldn’t worry about the price of groceries because we have plenty of money. That might be true, but it’s good to be careful with your money—​and it’s irresponsible of me to tell you to just waste your money.

As you get more comfortable with programming, use your memory resources efficiently. While you’re learning, just worry about getting your code to work.

### **2.3. bool Data Type (and Boolean Logic)**

A bool variable has only two possible values: true and false. It’s useful for tracking information that is only one thing or the other. Am I passing this class? The answer to that is either true or false--there is no other possibility. Does Julia own a car? Again, only two possible answers to that question; she either owns one or she doesn’t.

The best practice is to name a bool variable in a way that expresses this either/or state. That is to say, we usually name our bool variables using words like is, has, can, and so on.

*Example 2.8 - Boolean variable declarations in C#*

bool isPassing = true;

bool hasCar = false;

boolean variables go hand-in-hand with Boolean expressions which are statements that evaluate to be either true or false, like those questions above. We’ll look at this "Boolean logic" in the [0100 - Decisions](_0100_decisions.html).

### **2.4. char Data Type**

If we want to store a single character, like a student’s letter grade or their first initial, we can use the char data type. Char literals are created by putting a character in **single quotes**, and that character can be a letter, number, punctuation mark or symbol—​or some other weird stuff, too.

*Example 2.9 - Character variable declarations in C#*

char myLetterGrade = 'A';

char firstInitial = 'T';

|  |  |
| --- | --- |
|  | Some people pronounce the char data type like the word "chart" without the "t". And some people pronounce it like the word "care". Either is okay. |
|  | Also, I pronounce it "char" like "chart" without the "t" and firmly believe that only a sociopath would say it like the word "care". |

A char is really just an integer, but the number it holds conforms to a standard list of character values called **ASCII** (with the fun pronounciation "Ass key"). In this standard, the number 65 is a capital 'A', 66 is 'B', and so on. Lowercase letters are considered different characters, so 97 is 'a' and 98 is 'b'. Check out the [complete ASCII table](https://en.wikipedia.org/wiki/ASCII#Table_of_codes) if you’re curious.

### **2.5. Strings**

A char is pretty limited since it can only hold a single character. If we want to put a collection of characters together to make words and sentences, we need multiple chars grouped into a single variable. That data type is called string, because it strings together a bunch of chars, like a string of holiday lights.

*Figure 2.3 - A*string*variable is multiple*char*s strung together like a set of lights.*

string literals are denoted with **double quotation marks**; remember that char values use single quotation marks.

*Example 2.10 -*string*variable declarations in C#*

string address = "3000 N. Dysart Road";

string bestClass = "CIS150AB";

Strings are really important and there’s all kinds of fun and useful stuff we can do with them, but we’ll need to save all of that for later while we stay focused on the basics.

|  |  |
| --- | --- |
|  | Always remember that, in C#, single quotes ' mean a char and double quotes " mean a string. It’s easy to get them mixed up—​especially if you’re used to Python, where they are interchangeable—​but your code won’t compile if you mix them up. |

### **2.6. Declaring Constants**

A constant is similar to a variable, with two rules:

1. A value must be assigned when the constant is declared.
2. The assigned value can’t change during program execution.

To create a constant, add the keyword const at the start of your statement, followed by the rest of a declaration and assignment statement you’d use for a variable. So that people looking at your code can easily tell it’s a constant rather than a variable, it should be named with all capital letters, using the underscore character to separate words.

*Example 2.11 - Constant declarations in C#*

const double SALES\_TAX\_RATE = 8.7;

const int MINIMUM\_AGE = 18;

const String FAVORITE\_CLASS = "CIS150AB";

There are a few different reasons to use constants in your code. For now: \* Constants improve readability—​they identify the purpose of a value within your code. \* Constants prevent writing code that accidentally changes a value that shouldn’t change. \* Constants make code easier to maintain/update. \* In some situations, constants are more efficient than variables.

Interesting!

The naming convention used for C# constants is called **snake case**. More specifically, since it’s all caps, people refer to it as **screaming snake case**. Snake case with lowercase letters is the standard convention for variables in Python, among other languages.

### **2.7. Outputting Variable (and Constant) Values**

Assigning a value to a variable or constant does not produce any output. If we want to display the output of a variable—​or a constant—​we just put the identifier in a Write() or WriteLine() statement without any quotation marks:

*Example 2.12 - Basic variable output in C#*

String artistName = "Elvis Presley";

int birthYear = 1935;

Console.Write(artistName);

Console.Write(" was born in ");

Console.WriteLine(birthYear);

This code outputs

Elvis Presley was born in 1935.

#### **2.7.1. Concatenating Output**

We can combine output into one statement by creating a string with multiple pieces using the + symbol.

*Example 2.13 - Outputting variables with concatenation*

string artistName = "Elvis Presley";

int birthYear = 1935;

Console.WriteLine(artistName + " was born in " + birthYear);

Creating a String using the ` symbol is called \_concatenating\_. Be thoughtful when concatenating, because the ` symbol is also used to do addition with numbers, as you’ll see soon.

#### **2.7.2. Interpolating Output**

Another way to output variables and constants is to use interpolation. Interpolation is a way to insert the value of a variable or constant into a string. There are two ways to interpolate in C#. The easiest for beginners is to put the $ symbol in front of the opening quotation mark of the string, and then put the variable or constant name in curly braces {} where you want those values inserted.

*Example 2.14 - Outputting variables with interpolation*

string artistName = "Elvis Presley";

int birthYear = 1935;

Console.WriteLine($"{artistName} was born in {birthYear}");

You can also interpolate values by placing integers in the curly braces, which will insert the value of the variable or constant in the order they appear in the Write() or WriteLine() statement. Beginners often find this style a little more confusing, but it’s good to know about it—​and programmers from some other languages may prefer it.

*Example 2.15 - Outtputting variables with interpolation - alternative syntax*

string artistName = "Elvis Presley";

int birthYear = 1935;

Console.WriteLine($"{0} was born in {1}", artistName, birthYear);

The {0} and {1} in the string act as placeholders, and then the values are added on with commas after the string. artistName is the first variable listed, so it goes in the first placeholder; birthYear is the second variable listed, so it goes in the second placeholder.

#### **2.7.3. Choosing an Output Method**

For our purposes, there’s no difference between outputting using separate Write() statements, concatenation, or interpolation; you can use whichever approach you prefer. Most programmers end up using interpolation because it’s easy to write—​and easy to read. It also lends itself well to formatting output, which means making the output look nice and readable; we don’t worry about that here, but eventually you’ll find it useful.

But as far as I’m concerned, use whatever approach you like.

### **2.8. Math Calculations**

To start doing some calculations, we’ll use operators. You can think of an operator as a symbol that performs a calculation or other action. You’ve been using an operator already: the assignment operator, which uses the = symbol. The action it completes is assigning the value on the right of the = symbol to the variable on the left. Arithmetic operations work in a similar way. In C#, there are five basic arithmetic operators:

|  |  |
| --- | --- |
| *Table 1. C# arithmetic operators* | |
| Operator | Description |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division (quotient) |
| % | Modulo (remainder) |

The arithmetic operators work pretty much the way you’d expect, except maybe modulo--which might be a term you’ve never heard before. Each operator acts on the value to either side:

*Example 2.16 - Simple arithmetic operations in C#*

int sum = 5 + 7;

int difference = sum - 2;

|  |  |
| --- | --- |
|  | The value of sum will be 12 |
|  | The value of difference will be 10 (i.e., 10 - 2) |

Time To Watch!

Arithmetic Operations in C# [COMING SOON!]

#### **2.8.1. Order of Operations**

Early on in your math studies you learned about order of operations when an arithmetic expression has more than one calculation, and it works the same in C#. We call this operator precedence, and here are the guidlines:

1. Any operations enclosed in parentheses are evaluated first, following the rest of the rules here.
2. Multiplication, division, and modulus are evaluated next: the \*, /, and % operators. If there are more than one of these operations in the expression, they are evaluated from left to right.
3. Addition and subtraction are evaluated last. As above, if the expression contains more than one + or - operator, they evaluate from left to right.

Consider the following examples:

*Example 2.17 - Operator precedence in C#*

int result1 = 17 - 4 \* 6 / 3;

int result2 = 17 - 4 / 2 + 2;

int result3 = 17 - 4 / (2 + 2);

|  |  |
| --- | --- |
|  | result1 is 8: 4 \* 6 is 24, then 24 / 3 is 8, and then 17 - 8 is 9. |
|  | result2 is 17: 4 / 2 is 2, then 17 - 2 is 15, and then 15 + 2 is 17. |
|  | result3 is 16: (2 + 2) is 4, then 4 / 4 is 1, and then 17 - 1 is 16. |

#### **2.8.2. More Arithmetic with Less Typing!**

There’s a pretty consistent rule of thumb in coding that says programmers want to type as little as possible, so programming languages often provide shorthand ways of writing code that’s used frequently. Compound assignment operators (also called shorthand operators) simplify the syntax when you need to change a variable’s value relative to it’s existing value. For example, if we want to add 10 to a weight variable that already has the value 145, we could use the following:

weight = weight + 10;

C# starts on the right side of the assignment expression and retrieves the current value of weight, which is 145, adds 10 to that value, and stores the result back in weight.

We can combine the addition operator (`) with the assignment operator (`=`) to make a compound addition operator: `=, which allows us to rewrite the above line of code as:

weight += 10;

You can use compound assignment operators for all of the arithmetic opeerations:

* += adds the value on the right to variable value on the left.
* -= substracts the value on the right from the variable on the left.
* \*= multiplies the value on the left by the value on the right.
* \= divides the variable value on the left by the value on the right.
* %= divides the variable value on the left by value on the right, then assigns the **remainder**.

An operation we might not use much now, but will start using a lot when we learn to write loops, is incrementing a value, or adding 1 to a value. The increment operator (two plus symbols) gives us a very easy way to do that. On somebody’s birthday, for example, we could write:

age++;

The ++ simply adds 1 to the current value of age. The decrement operator is --, and it subtracts 1 from a variable’s value. If we’re counting down the number of days until our next birthday, we could execute this expression each morning:

daysRemaining--;

Increment and decrement only require one operand, so we call them unary operators.

|  |  |
| --- | --- |
|  | There are two forms to the increment and decrement operators: **prefix** and **postfix**. These examples use the postfix form, putting the operator after the variable name, whereas a prefix form would have the operator before the variable name: ++age. There’s a subtle difference in how postfix and prefix operations work, but for now you can use them interchangeably. I mention it here only because you might see code examples online using the prefix form. |

### **2.9. Handling User Input**

Until now, our code hasn’t been interactive—​each execution of a program results in the exact same output, and the user never has the chance to input anything. To produce output, we’ve been using Console.WriteLine and Console.Write to display text on the screen. For input, we’ll use Console.ReadLine (and, occasionally, Console.Read).

*Example 2.18 - Simple user input in C#*

string artist = "";

Console.Write("Enter the name of your favorite music artist: ");

artist = Console.ReadLine();

Console.WriteLine($"Your favorite music artist is: {artist}");

Console.ReadLine only deals with string values; even if the user types in a number, it will be stored as a string--which means we’ll need to convert it to a number if we want to do any math with it.

*Example 2.19 - Incorrect use of user input*

int age = -1;

Console.Write("Enter your age: ");

age = Console.ReadLine();

age++;

Console.WriteLine($"Next year you will be {age}");

|  |  |
| --- | --- |
|  | This code will not compile because Console.ReadLine returns a string, but we’re trying to assign it to an int. |

#### **2.9.1. Converting Data Types**

We’ll see later that C# can convert from one numeric type to another using a process called type casting. But we can’t just cast a string to numeric type because the string might not contain a number. Instead, we need to use a method to convert the string to a number.

The Convert class has a few methods that can help us with this conversion. The two we’ll use most are:

* Convert.ToInt32 converts a string to an int.
* Convert.ToDouble converts a string to a double.

|  |  |
| --- | --- |
|  | The 32 in ToInt32 is the number of bits used to store the number, and that’s the size of the int data type. There are methods to convert to 16-bit and 64-bit integers as well, though we won’t be using them. |

*Example 2.20 - Correct conversion of user input*

int age = -1;

double gpa = -1.0;

Console.Write("Enter your age: ");

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

age++;

Console.WriteLine($"Next year you will be {age}");

Console.Write("Enter your GPA: ");

gpa = Convert.ToDouble(Console.ReadLine());

gpa += 0.5;

Console.WriteLine($"If you improve by half a letter grade, you GPA will be {gpa}");

Time To Watch!

User Input and Conversions in C# [COMING SOON!]

### **Check Yourself Before You Wreck Yourself (on the assignments)**

#### **Can you answer these questions?**

1.

2.

3.

← Previous: [0001 - Computers and Coding](_0001_computers_and_coding.html) | ↑ Up: [Skimpy C#: An Introduction to Programming](index.html) | Next: [0011 - Methods](_0011_methods.html) →

Tripucka Edition, revised 2025-03-23

# Transcript

# Transcript

# Transcript

# Transcript

# Transcript