**Review and Research**

You can review the video of Session 1 here:

<https://blizzard.sharepoint.com/portals/hub/_layouts/15/PointPublishing.aspx?app=video&p=p&chid=8aa7fa80-bfa2-4021-bf44-543dba93f693&vid=a67f69ba-a30a-4c2b-86b0-1af717c2d0ee>

One of your best resources for C# related information is Microsoft’s official documentation. The reading can sometimes be dry and tends toward the use of jargon, but the information you get is almost always going to be accurate and up-to-date. As always, email me at [semerson@blizzard.com](mailto:semerson@blizzard.com) if you have any questions about what you find there.

Here’s a reference for commonly used types:

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/built-in-types-table>

Here’s another for operators:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/operators>

You can check the ranges of various integer types here:

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/integral-types-table>

And you can learn more about float vs double here:

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/floating-point-types-table>

**Glossary**

At the end of each session I’ll provide definitions of some of the terminology we used in class. I’ll be striving for readability when writing these definitions; other, more technical definitions exist and you should consider researching those as well. If there are any additional terms you’d like to see defined be sure to let me know and I’ll add them as we go along. Note that these aren’t presented in alphabetical order, but rather in the order of more fundamental to more complex, grouped around common ideas.

Variable – A named value we can use to store data of various types. In C#, we usually begin variable names with lower-case letters.

Type – Refers to a specific kind of data in a programming language. Common types that we use in C# include integers (whole numbers), floating-point numbers (i.e., numbers with decimal components), Boolean values (true and false), characters, and strings.

Strong typing – When we say that C# is a strongly-typed language, we mean that we need to specify the type of every variable that we declare, and that functions need to specify what types they expect to receive as input and return as output. This arguably creates a little more work for us as programmers, but it also allows C# to offer us a lot of protection and efficiency in return… if we try to pass a string type to a function that expects an integer type, the compiler will catch the mistake and let us know.

Implicit type conversion – This happens when C# converts a variable from one type to another without our explicitly requesting it. For example, if you call a function that expects a floating-point number and you pass it an integer instead, C# may implicitly convert the integer to a float and then call the function normally using the float instead. There are strict rules about what conversions can happen in differing contexts; you can read up on these, but many of them you’ll probably discover through experimentation.

Operator – An operator is a character (or small group of characters) that represent a specific action to be taken, such as ‘=’ for assignment and ‘+’ for addition. Many operators are based on common mathematical operations. Operators can be used for math, assignment, comparison, and many other tasks that we’ll cover in time.

Namespace – A way of organizing chunks of code that are related to one another. For example, we might store classes and functions dealing with mathematics in a namespace called “Mathematics,” while code dealing with input and output might go into a namespace called “IO.”

Function – A block of code that, ideally, has been defined to perform a single, specific task. Functions can take input, return output, and should always be given clear, precise names. Functions names in C# usually start with upper-case letters.

Method – In C#, method is another name for a function. In other programming languages, methods and functions may refer to similar but slightly different concepts, but in C# the terms are interchangeable.

Compile – C# is a compiled language, which means that before we can execute a C# program, the code needs to be converted to another form. This process is called compilation, and the new form is called an “assembly.”

**Practice Exercises**

These exercises are grouped around concepts we’ve covered in class and range from very simple to more complex. A few reminders before I present them:

1. When compiling and running these exercises in Visual Studio, be sure to do so in Debug Mode. You can do this with the hotkey combination ctl+F5.
2. These exercises are meant to be a start to your practice, but if you want more ideas, contact me and let me know. I’ll be happy to make additional suggestions!
3. Don’t be afraid to repeat an exercise several times. Repetition in your practice is very helpful, especially in the beginning when you’re trying to get comfortable with syntax as well as new concepts.
4. If you get stuck on a problem or don’t understand why something is happening, please contact me and let me know. I’ll be happy to help.
5. An exercise may occasionally require you to use a concept we haven’t covered in class; when this is the case, the exercise will be marked with an asterisk there will be a link to research you can do help you find the new information you need.

*Basics, Types and Variables*

Ex 1. Create a new Visual C# Console application and name it whatever you wish. Within the Main() function, write a message to the console by calling the Console.WriteLine() function.

Ex 2. Declare a string variable and initialize it with the text, "This is one half". Declare a second string and initialize it with the text "of a sentence." Use the Console.Write() function and your two string variables to write "This is one half of a sentence." to the console.

Ex 3. Declare an int variable named "myInt" and try to initalize it with the following values: 10, 100, 1000, 10000, etc., adding a zero each time. When you eventually reach a value that is invalid, read the resulting error message. Determine what different type you can declare that will allow you to store larger integers.

Ex 4. Declare a float variable named "myFloat" and assign it the value "1.123456789f". Declare a double and assign it the value "1.123456789" (Note that the only difference between these values is the 'f' at the end of the float value.) Use Console.Writeline() to write each variable to the console. Compile and execute the program and think about the results; how is a float different from a double? Keep experimenting with different values to get a feel for how these two types behave.

*Operators*

Ex 1. Declare an int named "x" and initialize it with the value "0". Practice changing its value to 5 using the following expressions: (1) x = x + 5; (2) ++x; (You'll need to do this a total of 5 times!) (3) x += 5;

Ex 2. Declare an int named "x" and initialize it with the value "3". Using the '\*' and '\*=' operators, try to raise x to the powers of 2, 3, and 4.

Ex 3. Declare a float named "x" and initalize it with the value "6.5f". Declare a float named "y" and initalize it with the value "-6.5f". Declare a bool named "compare." Initialize "compare" as follows: compare = x > 1.5f; Finally, write "compare" to the console using Console.WriteLine(). Predict whether the result will be true or false, then compile and execute your program to check your prediction.

Ex 4. Repeat the above exercise with each of the following assignments for "compare." In each case, predict whether the result will be true or false before execution. If your prediction is incorrect, try to work out why (and be sure to ask for help if you aren't sure!)

compare = x > y;

compare = x <= 6.5f;

compare = 6.5f == y;

compare = x < y || x > y;

compare = x > 0f && y > 0f;

compare = x \* -1f == y;

compare = x != y && x < y;

*Functions*

Ex 1. Write a function called PrintString that takes a string as input and has no output (use the keyword 'void' to indicate that there is no output.) Your function should write the provided string to the console. Here is what it will probably look like:

static void PrintString(string s)

{

Console.WriteLine(s);

}

Now declare a string called "message" and initialize it with the text, "Test string!". Call your PrintString function with "message" as its input. Compile and and run your program to ensure that the function is working properly.

As a follow-up, try calling your function with a string literal instead, e.g.,

PrintString("Stay awhile and listen!");

What is the result? Try to figure out why you got the result that you did.

Ex 2. Write a function that, when given an integer value n, returns the summation of 1 through n. If you're not familiar with it, here's a mathematical formula your function can use to get the correct result:

summation = n \* (n+1) / 2

Test your function by calling it with the value '5' and using the console to confirm that the return value is 15.

Ex 3. Write a function called "TakeDamage" that simulates a character in a game taking hitpoint damage. The function should take two integers as input: One to represent the current HP of the target, and one to represent damage taken. Return the number of HP remaining.

\*Ex. 4. Write a function that takes an integer as input and returns true if the integer is positive and false if it is 0 or negative. Write a second function that takes an integer as input and returns true if the integer is even and false if it is odd (to work this out you’ll probably need to use the ‘remainder’ operator. see the link below for help!)

Test both of your functions to see if they seem to be working properly. If so, write a third function that takes an integer as input and returns true if it is a positive integer and false if not. Have your third function make use of your first two functions in order to return the correct answer.

Link to help on the remainder operator:

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/remainder-operator>

\*Ex 5. Continue to practice writing functions by imagining different simple tasks that they can perform. Here are some ideas:

- Given three floating point numbers, return the average of all three.

- Given two integer values, return true if they are equal and false if not.

- Given an integer value, return the square of that integer.

- Given two strings, combine both strings into a single string and write the new string to the console.

- Given the lengths of two sides of a right triangle, return the length of the hypotenuse (take a look at the Math class linked below to find a function you can use to calculate square roots, as well as many other common mathematical operations.)

<https://msdn.microsoft.com/en-us/library/system.math(v=vs.110).aspx>