CSCI 1300 – Practicum 2 Review Guide

1a. Data Types :=

When programming, we store the variables in our computer's memory, but the computer needs to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way. Some commonly used data types in C++ are:

1. **`int`** (for integers)
   - `int myInt = 5;`
2. **`char`** (for characters)
   - `char myChar = 'c';`
3. **`float`** (for floating-point numbers)
   - `float myFloat = 4.4531;`
4. **`double`** (for double precision floating-point numbers)
   - `double myDouble = 4.4531;`

1b. Casting :=

Casting is a method for converting a value across datatypes. For example, if you had a **`float`** variable with value 10.2129, but you needed to use that variable as an **`int`**, you could "cast" it as an integer with the following syntax options:

```
//option 1
float variable = 10.2129;
int variable_as_int = (int)variable;

//option 2
float variable = 10.2129;
int variable_as_int = int(variable);
```

Where the keyword in (**int**) or **int**() can be any valid datatype for which there is a conversion between the two datatypes.

Data types are *not* concrete values themselves, but rather, ways of interpreting values. This means that the same value can be 'cast' to different data types to create different values.

C++ contains a built in functions to cast a value to each data type - int() - float() - str() - bool()

2. Variables :=

A **variable** in simple terms is a storage place which has some memory allocated to it. Basically, a variable is used to store some form of data. Different types of variables require different amounts of memory, and have some specific set of operations which can be applied on them.

Variable Declaration:

A typical variable declaration is of the form:

```
//For one variable:
type variable_name;
//Or for multiple variables:
type variable_name, variable2_name, variable3_name;
```

A variable name can consist of alphabets (both upper and lower case), numbers and the underscore '_' character. However, the name **must not** start with a number.

Difference between variable declaration and definition

Variable declaration refers to the part where a variable is first declared or introduced before its first use. *Variable definition* is the part where the variable is assigned a memory location and a value. Most of the times, variable declaration and definition are done together.

See the following C program for better clarification:

```
//declaration and definition of variable 'a123'
char a123 = 'a';

//this is also both declaration and definition as 'b' is allocated
//memory and some garbage value
float b;

//multiple declarations and definitions
int _c, _e45;
```

3. Arithmetic :=

Throughout your programs, you will have to perform various calculations with and modifications of your variables. These are most often carried out using arithmetic operators, which are the basic math operations used in algebra.

Arithmetic Operator	**C++ Operator**	**Algebraic Expression**	**C++ Expression**
addition	+	$x + n$	x + n
subtraction	-	$x - n$	x - n
multiplication	*	$x * n$ or xn	x * n
division	/	x / n or $\frac{x}{n}$	x / n
modulus	%	$x \bmod n$	x % n
exponentiation	N/A	x^n	pow(x,n)
taking roots	N/A	$\sqrt[n]{x}$	pow(x,1/n)

4. Print/ Output_to_the_command_window / Display :=

When writing a program, printing your output to the console is a good way to gauge your work and assess if the program is working as you intend it. In the absence of a good debugging tool, print statements can let you see variable values or function returns to make sure your program is running properly. Below are some examples of various types print statements and their s.

```
//values of variables
float variable = 10.2129;
cout << variable << endl;

//text messages, strings
cout << "Hello World" << endl;

//option 2
//birthday_timer is a function that returns the days until my birthday
cout << birthday_timer() << endl;
```

5. Conditionals :=

Conditional statements in computer science are features of a programming language which execute different computations or actions based on the outcome of a *boolean condition*. Boolean conditions are expressions that evaluate a relationship between two values and then return either True or False. Conditionals are mostly used in order to alter the *control flow*, or the order in which individual instructions are carried out, of a program.
Largely, conditional statements fall into two types:

1. If-else statements, which execute one set of statements or another based on the value of a given condition
2. Switch statements, which are used for exact value matching and allow for multi-way branching

IF-ELSE statements:
The *if statement* selects and executes the statement(s) based on a given condition. If the condition evaluates to True then a given set of statement(s) is executed. However, if the condition evaluates to False, then the given set of statements is skipped and the program control passes to the statement following the if statement. If we have another *else statement* next to this, it is handled by the else block.

The general syntax of the if-else statement is as follows:

```
if ( condition ) {
     //execute some code here
}
```

Nested IF-ELSE:
If there are more than 2 branches within your program, we can use *nested if statements* to model this behavior within a program by putting one if statement within another.

```
if ( condition ) {
     //another decision to make within this branch
     if ( condition2 ) {
          //execute some code
     }
}
else{
     //execute some different code here
}
```

There are several important things to understand about if-else blocks.

Each block represents an isolated test. This means that within the context of an if-elseif-else block, once a condition is satisfied, the entire block is done. This is different from using multiple if statements, where every condition would be tested regardless of the success or failure of previous conditions.

An if-elseif-else block must begin with an if statement, and it can be ended with 0 OR 1 else statements. That initial if statement can be followed by **0 OR MORE else if** statements that check additional conditions. The final else statement should serve as a catch-all or default case. i.e. if none of the above cases are true, then "do this".

SWITCH statements:
Switch case statements are a substitute for long if statements that compare a variable to several integral values.

- The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression
- Switch is a control statement that allows a value to change control of execution.

Syntax:
With the switch statement, the variable name is used once in the opening line. A case keyword is used to provide the possible values of the variable, which is followed by a colon and a set of statements to run if the variable is equal to a corresponding value.

```
switch (n){
     case 1:
          // code to be executed if n = 1;
          break;
     case 2:
          // code to be executed if n = 2;
          break;
     default:
          // code to be executed if n doesn't match any cases
}
```

Important notes to keep in mind while using switch statements :

1. The expression provided in the switch should result in a constant value otherwise it would not be valid.
    a. switch(num) //allowed (num is an integer variable)
    b. switch('a') //allowed (takes the ASCII Value)
    c. switch(a+b) //allowed,where a and b are int variable, which are defined earlier
2. Duplicate case values are not allowed.
3. The **break** statement is used inside the switch to terminate a statement sequence. When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

4. The **default** statement is optional. Even if the switch case statement do not have a default statement,it would run without any problem.
5. Nesting of switch statements are allowed, which means you can have switch statements inside another switch. However nested switch statements should be avoided as it makes program more complex and less readable.
6. The break statement is optional. If omitted, execution will continue on into the next case. The flow of control will fall through to subsequent cases until a break is reached.

Relational Operators:
A *relational operator* is a feature of a programming language that tests or defines some kind of relation between two entities. These include numerical equality (e.g., 5 = 5) and inequalities (e.g., $4 \geq 3$). Relational operators will evaluate to either True or False based on whether the relation between the two operands holds or not. When two variables or values are compared using a relational operator, the resulting expression is an example of a *boolean condition* that can be used to create branches in the execution of the program. Below is a table with each relational operator's C++ symbol, definition, and an example of its execution.

>	greater than	5 > 4 is TRUE
<	less than	4 < 5 is TRUE
>=	greater than or equal	4 >= 4 is TRUE
<=	less than or equal	3 <= 4 is TRUE
==	equal to	5 == 5 is TRUE
!=	not equal to	5 != 4 is TRUE

Logical Operators
Logical operators are symbols that are used to compare the results of two or more conditional statements, allowing you to combine relational operators to create more complex comparisons. Similar to relational operators, logical operators will evaluate to True or False based on whether the given rule holds for the operands. Below are some examples of logical operators and their definitions.

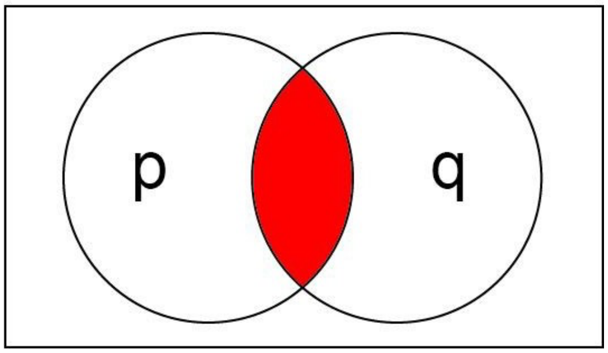
- **&&** (AND) returns true if and only if both operands are true
- **||** (OR) returns true if one or both operands are true
- **!** (NOT) returns true if operand is false and false if operand is true

Truth Tables
Every logical operator will have a corresponding *truth table*, which specifies the output that will be produced by that operator on any given set of valid

inputs. Below are examples of truth tables for each of the logical operators specified above.

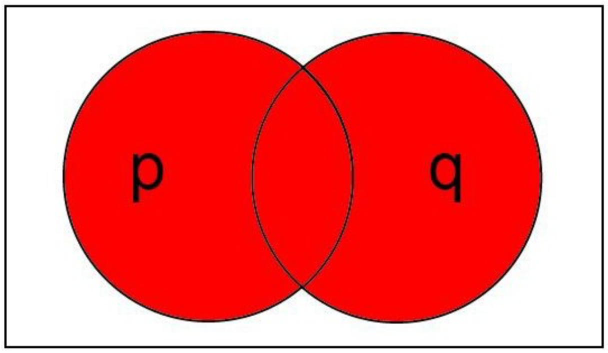
AND:
These operators return true if and only if both operands are True. This can be visualized as a venn diagram where the circles are overlapping.

p	**q**	**p && q**
True	True	True
True	False	False
False	True	False
False	False	False



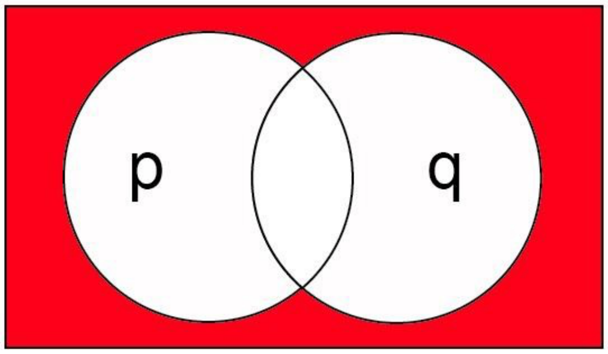
OR:
These operators return True if one or both of the operands are True. This can be visualized as the region of a venn diagram encapsulated by both circles.

p	**q**	**p || q**
True	True	True
True	False	True
False	True	True
False	False	False



NOR:
These operators return True if both of the operands are False. This can be visualized as the region of a venn diagram that is not within either of the circles.

p	**q**	**!(p || q)**
True	True	False
True	False	False
False	True	False
False	False	True



6. Arrays :=

An array is a collection of ordered values. We refer to the individual values in the collection as elements. The elements of the array can either be of a primitive type, or a user-defined type, but all the elements of an array are the same type. This is why we call arrays derived types; a double array is a different type than a double, but its elements are double type.

In the code below, two arrays are created. Double array `numbers` is created with size ten. It can only hold 10 double elements and cannot grow. At the moment it is declared, all of its 10 elements contain garbage data.
Contrast this with int array `sequence`, which has been initialized with the elements 1, 2, and 3.

```
double numbers[10]; /* Define a double array with 10 elements called
                       Numbers */
int sequence[3] = { 1, 2, 3 }; // Declare and Initialize an int array
```

The elements are arranged in an ordered sequence, in which each element is accessed by its index. The index indicates an element's place in the sequence. It is like an address. The elements of the array may be accessed or assigned to using the access operator `[]`. Array indices begin at 0. Note how `[]` is used differently during declaration (where it defines size, above) than it is with access (below).

```
numbers[0] = 41.6; // Assign a float value to the first index
int last =  sequence[2]; // Access last elem and assign to int var
```

Note that arrays themselves do not contain a concept of length, and you must be careful not to attempt to access elements that do not exist, or an error will (likely) result.

```
numbers[10] = 41.6; // ERROR: The 10th element is at index 9
int last =  sequence[-1]; // ERROR: Negative indices have no meaning
```

7. Defining Functions :=

A function is a named block of code which is used to perform a particular task. The power of functions lies in the capability to perform that task anywhere in the program without requiring the programmer to repeat that code many times. This also allows us to group portions of our code around concepts, making programs more organized. You can think of a function as a mini-program.

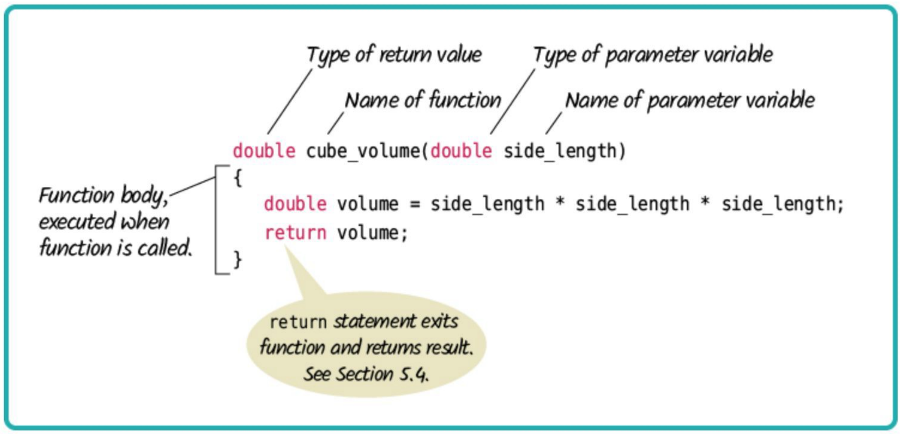
Depending on whether a function is predefined or created by programmer; there are two types of function:

1. Library Function
2. User-defined Function

Library functions are pre-written functions, grouped together in a ***library*** of related functions. For example, the C++ math library provides a sqrt() function to calculate the square root of a number. Libraries other than the built-in C++ libraries can be found online.

C++ allows programmers to define their own functions. These are called user-defined functions. Every valid C++ program has at least one function, the main() function.

We pass values to functions via ***parameters***. In general, the parameters should be all the information needed for the function to do its work. When that work is complete, we would like to use the result in other code. The function can ***return*** one value of the specified type. A function may be used anywhere that is appropriate for an expression of that type. A function may also return nothing, in which case its return type is ***void***.


Type of return value
Type of parameter variable
Name of function
Name of parameter variable
Function body, executed when function is called.
return statement exits function and returns result. See Section 5.4.

```
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}
```

Here is the syntax for the function definition:

```
return_type function_name( parameter_list )
{
// function_body
}
```

- return_type is the data type of the value that the function returns.
- function_name is the actual name of the function.
- parameter_list refers to the type, order, and number of the parameters of a function. A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument.
- function_body contains a collection of statements that define what the function does. The statements inside the function body are executed when a function is called.

When we want to run (invoke) a function, we ***call*** it using syntax like this:

```
variable = function_name( arguments )
```

A function has its own ***scope***. That means that the parameters of the function, as well as ***local variables*** declared *within* the function body, are not accessible outside the function. This is useful because it allows us to solve a small problem in a self-contained way. Parameter values and local variables disappear from memory when the function completes its execution.

When we call a function, the ***flow of execution*** changes. When a program begins running, the system calls the main() function. When control of the program reaches a function call inside the main(), execution moves to that function and all code inside the function is executed. After the function finishes, the execution returns in the main() function and the statement following the function call is executed. Note that when functions call other functions, a similar change in execution flow occurs.

Here is a function that has an array as an argument:

```
double array_sum( double arr[] , int length )
{
// function_body
}
```

When an array is passed to a function, it is ***pass by reference***. That is, the array parameter is a reference to the array argument that was passed to the function. Any changes that are made to the array within the function, are also reflected in the array that was passed to the function.

8. Loops :=

Loops in programming come into use when we need to repeatedly execute a block of statements. For example: Suppose we want to print "Hello World" 10 times. This can be done in two ways as shown below:

Iterative Method

The iterative way to do this is to write the cout statement 10 times, as below.

```
int main()
{
cout << “Hello World” << endl;
cout << “Hello World” << endl;
cout << “Hello World” << endl;
cout << “Hello World” << endl;
cout << “Hello World” << endl;
cout << “Hello World” << endl;
cout << “Hello World” << endl;
cout << “Hello World” << endl;
cout << “Hello World” << endl;
cout << “Hello World” << endl;

return 0;
}
```

In a loop, the statement needs to be written only once and the loop will be executed 10 times as shown below.

In computer programming, a loop is a sequence of instructions that is repeated until a certain condition is reached. An operation is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number.

1) Counter not Reached: If the counter has not reached the desired number, the next instruction in the sequence returns to the first instruction in the sequence and repeat it.
2) Counter reached: If the condition has been reached, the next instruction “falls through” to the next sequential instruction or branches outside the loop.

There are mainly three types of loops: **while** and **for**

for Loops:

A for loop is a repetition control structure which allows us to write a loop that is executed a specific number of times. The loop enables us to perform n number of steps together in one line.

Syntax:

```
int main()
{
      for ( initialization expr; test expr; update expr)
      {
            //body of the loop
            //statements we want to execute
      }
      return 0;
}
```

In a for loop, a loop variable is used to control the loop. First initialize this loop variable to some value, then check whether this variable is less than or greater than counter value. If this conditional is true, then, then loop body is executed, and the loop variable gets updated. Steps are repeated until the condition is no longer met.

- **Initialization Expression:** In this expression we have to initialize the loop counter to some value. for example: int i=1;
- **Test Expression:** In this expression we have to test the condition. If the condition evaluates to true then we will execute the body of loop and go to update expression otherwise we will exit from the for loop. For example: i <= 10;
- **Update Expression:** After executing loop body this expression increments/decrements the loop variable by some value. for example: i++;

To print a statement 10 times using a for loop, the code would look like this:

```
int main()
      {
      for (int i = 0; i < 10; i++)
      {
            cout << "Hello World" << endl;
      }
      return 0;
}
```

For loops are extremely useful for iterating over arrays. When we iterate over an array, we set up the for loop so that the loop variable will be equal to each of the array indices. We can then use this index to perform a similar action to every element of the array:

```
int main()
{
      double numbers[3] = {1,2,3};
      for (int i = 0; i < 3; i++)
      {
            // Multiply each element of the array by 5 and print the result
            cout << numbers[i]*5 << endl;
      }
      return 0;
}
```

while Loop:

While studying for loops, we have seen that the number of iterations is known beforehand, i.e. the number of times the loop body is needed to be executed is known to us. while loops are used in situations where we do not know the exact number of iterations of loop beforehand. The loop execution is terminated on the basis of test condition.

Syntax:

We have already stated that a loop mainly consists of three statements – initialization expression, test expression, update expression. The syntax of the three loops – for, while and do while mainly differs on the placement of these three statements.

```
int main()
{
   int i = 0; // initialization expression
   while ( i < 10 ) // test expression
   {
      cout << "Hello World" << endl; //body of the loop
      i++; // update expression
   }
   return 0;
}
```

9. Strings:=

In C++, string is a data type that represents sequences of characters instead of a numeric value (such as int or float). A string literal can be defined using double quotes. So "Hello, world!", "3.1415", and "int i" are all strings. We can access the character at a particular location within a string by using square brackets, which enclose an ***index*** which you can think of as the ***address*** of the character within the string.

Importantly, strings in C++ are indexed starting from zero. This means that the first character in a string is located at index 0, the second character has index 1, and so on. For example:

```
string s = "Hello, world!";
cout << s[0] << endl;  //prints the character 'H' to the screen cout <<
s[4] << endl;  //prints the character 'o' to the screen cout << s[6] <<
endl;  //prints the character ' ' to the screen cout << s[12] << endl;
//prints the character '!' to the screen
```

There are many useful standard functions available in C++ to manipulate strings. One of the simplest is length(). We can use this function to determine the number of characters in a string. This allows us to loop over a string character by character (i.e. *traverse the string*):

```
string s = "Hello, world!"; cout
<< s.length() << endl;                    //This will print 13
for (int i = 0; i < s.length(); i++)
{
cout << s[i] << endl;
}
```

This will print each character in the string "Hello, world!" to the screen

one per line. Notice how the length function is called.
The correct way:

- s.length()

Common mistakes:

- length(s)
- s.length
- s.size()

This is a special kind of function associated with objects, usually called a *method*, which we will discuss later in the course.