# Assignment 03: Iteration (looping, repetition) Control Structures

COSC 1437: Programming Fundamentals II

## Objectives

- Continue to practice using C/C++ relational and boolean operators
- Continue to practice using arithmetic operators in C++
- Create index variable controlled `for` loop to calculate sequences using a running sum
- Create and use sentinel controlled loops using `while` statements

## Description

In this lab/assignment, our goal is to learn how to use some basic repetition control structures available in C/C++, in order to repeat a block of code to perform a complex calculation. C/C++ has 3 repetition control structures:

1. `for` loop control structures for index controlled iteration
2. `while` loop control structures for using a sentinel condition to control looping
3. `do while` loop structures allowing for specifying a loop that is guaranteed to always execute at least 1 time.

In this lab we will only be using the first two types of repetition control structures. And you will also continue to get practice in declaring and using variables of different data types, and specifying arithmetic and logical calculations in programs.

## Overview and Setup

For all assignments, it will be assumed that you have a working VSCode IDE with remote Dev Containers extension installed, and that you have accepted and cloned the assignment to a Dev Container in your VSCode IDE. We will start each assignment with a description of the general setup of the assignment, and an overview of the assignment tasks.

For this assignment you have been given the following files (among some others):

| File Name | Description |
|---|---|
| `src/assg03-tests.cpp` | Unit tests for the tasks that you need to successfully pass |
| `src/assg03-library.cpp` | File that contains the code implementations, all your work will be done here |
| `include/assg03-library.hpp` | Header include file of function prototypes |

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub by accepting the assignment using the provided assignment invitation link for 'Assignment 03' for our current class semester and section.
2. Clone the repository using the SSH url to your local class DevBox development environment. Make sure that you open the cloned folder and restart inside of the correct Dev Container.
3. Confirm that the project builds and runs, though no tests may be defined or run initially for some assignments. If the project does not build on the first checkout, please inform the instructor.
4. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment.

# Assignment Tasks

## Lab Task 1 / In-Lab Work : Sum up range of values using an increasing step size

For the first task, you will need to define some variables and implement a while loop that will be controlled by a relational expression. To get started, make sure that you `#define` the task1 tests, and create the Task 1 issue in your GitHub classroom.

For this task, there is a function named `sumRangeIncreasing()` located in `assg03-library.cpp`. If you look at the first test in `assg03-tests.cpp` of this function, it does this:

```
// sum first 5 integers 1 + 2 + 3 + 4 + 5 == 15
CHECK(sumRangeIncreasing(1, 5, 1) == 15);
```

The first parameter to `sumRangeIncreasing()` will be the `begin` of a range of integers and the second will be the `end` of a range (look at the function documentation in `assg03-library.cpp`. The third parameter is a step size. So for example, this test should cause the first 5 integers to be summed up and returned, which is what is being checked here.

The second test shows the effect of the step size (the third parameter). If we instead call the function like this:

```
// check that you are using step size and including
// the end value inclusively in sum
// 1 + 3 + 5 == 9
CHECK(sumRangeIncreasing(1, 5, 2) == 9);
```

then we should step by 2 in the range to be summed. Thus you sum up the values $1 + 3 + 5 = 9$ as shown in the comment.

You need to use a `while` loop to implement the needed algorithm. Find the function named `sumRangeIncreasing()` in the `assg03-library.cpp` file and do the following to implement this behavior.

1. Define two local variables, both of type `int`. Call the first `sum` and the second `current`.
   - Initialize `sum` to 0, this will hold the running sum you will calculate and return.
   - Initialize `current` to be the `begin` value that is passed into this function as the first parameter.
2. Create a while loop. Lets use for example the first case where `begin=1`, `end=5` and `step=1`. You should have initialized your `sum` to 0 and your `current` should be set to 1, which was passed in as the beginning value of the range.
   - Inside of the loop you should add the `current` value into the running sum. So the first time in the loop, when sum is 0 and current is 1, the result put back into sum should be 1.
   - After you add the `current` value into the `sum`, increase `current` by the step size. Do this by adding `step` to `current` and assigning the result back into `current`.
   - These are the only two things that need to be done inside of the `while` loop. However you need to correctly stop the loop. As long as `current` is less than or equal to the `end` value of the range, you want to keep adding to the `sum` and stepping to the next value.
3. Initially the stub function given to you always returns 0. You should instead return the `sum` you calculate after your loop exits.

If you successfully complete the function, and save and compile and rerun the tests, you should find that all of the unit tests for Task 1 pass successfully. When you are satisfied your code is correct, create a commit and push your commit to your GitHub classroom repository. It is always a good idea to check the autograder after pushing a commit to ensure it is also compiling and passing the tests for the task you just completed on GitHub.

## Task 2: Sum up range of values using a decreasing step size

To get started with Task 2, `#define` the `task2` tests in the `assg03-tests.cpp` file, and create your Task 2 issue on GitHub classroom.

If you successfully completed your work for the first task, then Task 2 will use exactly the same code, but with just a single change in your `while` loop exit condition.

The second task is done in a function named `sumRangeDecreasing()`, and it works similarly to the previous task, but instead we sum up a range of values with a negative step size. So for example, the first test for this task looks like:

```
// sum first 5 integers 5 + 4 + 3 + 2 + 1 == 15
CHECK(sumRangeDecreasing(5, 1, -1) == 15);
```

So here `begin` will be 5, `end=1` and the `step` size is `-1`. So we are summing up the series in reverse, from 5 down to 1.

Since this function requires that the step size be negative, and that `begin` be the larger value of the range, and end the smaller, you don't have to change your code to for example update `current`. Adding a negative step to the `current` term of the series will correctly decrease by `1` when step is `-1`.

Since the algorithm is almost identical for Task 2 it is suggested that you start by simply copying your working code from Task 1 into the `sumRangeDecreasing()` function:

1. Copy and reuse all of your logic from previous task into the `sumRangeDecreasing()` function.
2. If you implemented Task 1 correctly, then the only change you need will be in the `while` loop exit condition. Instead of going as long as `current` is LESS than or equal to the `end` value, you need to keep looping as long as `current` is GREATER than or equal to the end value. If you look at the first test above you should see why. `current` should start out as 5, and will get `-1 step` added to it each iteration of the `while` loop. So as long as current is bigger than or equal to 1, you need to continue performing the loop and adding the terms to the running `sum`.

If you successfully complete the function, and save and compile and rerun the tests, you should find that all of the unit tests for Task 2 pass successfully. When you are satisfied your code is correct, create a commit and push your commit to your GitHub classroom repository. It is always a good idea to check the autograder after pushing a commit to ensure it is also compiling and passing the tests for the task you just completed on GitHub.

## Task 3: General case to sum a range given either a positive or negative step

As usual start by defining `#define` the `task3` tests in `assg03-tests.cpp`, and creating the Task 3 issue in GitHub classroom.

What we really want is a general algorithm `sumRange()` that can sum up a series of values using either a positive or negative step size.

If you look closely at the `task3` tests, you will see that we simply (mostly) repeat all of the `task1` and `task2` tests on the new more general version of a function named `sumRange()`.

Again, like in Task 2, the only change you need to perform to get this to work is to modify the `while` loop condition correctly. Though in this case we will need a much complicated loop termination condition, so you will need to use some boolean operators for this task.

So do the following:

1. Start by copying the code, either from the increasing or decreasing version of the algorithm. It shouldn't matter because if you did these correctly and they are passing, the only difference between them should be the `while` loop condition you used.
2. You need to specify the while loop condition so that it keeps running the loop as long as either the `current` values is between `begin` and `end` OR ELSE if the `current` value is between `end` and `begin`. It is either/or because for increasing step we need to execute while `current` is between `begin` (the smaller value in the range) and `end` (the larger value). But when the step size is negative and we are decreasing, then `begin` is larger and `end` is smaller. Mathematically we need our loop condition to state

$$(\text{begin} <= \text{current} <= \text{end}) \text{ OR } (\text{end} <= \text{current} <= \text{begin})$$

So you will need a boolean expression that uses both `and` and `or` boolean operators to test if `current` is either between `begin` and `end` or between `end` and `begin`. **HINT:** You can't directly use the mathematical expression given above, it is not syntatically allowable C/C++ code.

As before, if you get the loop termination expression correct, it should be the only thing you need to change in order to get all of the Task 3 tests to pass. No other code should need to be changed to get this most general case to work.

If you successfully complete the function, and save and compile and rerun the tests, you should find that all of the unit tests for Task 3 pass successfully. When you are satisfied your code is correct, create a commit and push your

commit to your GitHub classroom repository. It is always a good idea to check the autograder after pushing a commit to ensure it is also compiling and passing the tests for the task you just completed on GitHub.

## Task 4: Sum a series of increasing values

As usual start by defining `#define` the `task4` tests in `assg03-tests.cpp`, and creating the Task 4 issue in GitHub classroom.

You can calculate an approximation of $\pi$ by computing the following infinite series (known as the Leibniz series):

$$\pi = \sum_{i=0}^{n} (-1)^i \frac{4}{1+2i} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} \cdots$$

The more terms you compute and sum up from this sequence the better your approximation of $\pi$ will be. The equation given above showed calculating the series for the first 6 terms, for when $i = 0, 1, 2, ..., 5$.

Notice that the we alternate between adding and subtracting the terms. When $i = 0$ we add in $\frac{4}{1}$. When $i = 1$ we add in 4 and subtract $\frac{4}{3}$ from that, etc.

We will use an `if` condition statement to alternate the addition and subtractions when calculating the running sum of the series in the next task. But to begin with, let's instead calculate the simpler:

$$\sum_{i=0}^{n} \frac{4}{1+2i} = \frac{4}{1} + \frac{4}{3} + \frac{4}{5} + \frac{4}{7} + \frac{4}{9} + \frac{4}{11} \cdots$$

Here if it is not obvious, we are just going to just add up all of the terms in this series. In the function named `sumSeriesIncreasing()`, we pass in one parameter `n`. This specifies how many terms of `i` we want to generate and sum up. So if `n = 5` we want to generate and sum up the terms for $i = 0, 1, 2, 3, 4, 5$ of the series.

Do the following in the `sumSeriesIncreasing()` function to generate and sum up the indicated terms:

1. Like before, start by defining local variables named `sum` and `term`. Both of these need to be of type `double` for this task. This time initialize both of these local variables to `0.0`
2. You are required to use a `for` loop to perform indexed controlled looping in this task. Create a for loop that uses `i` as and index variable / counter. The loop should run from `i = 0` to when `i <= n`.
3. Inside of the loop calculate the $i^{th}$ term each time and put the calculation in the `term` variable you defined. For example when `i` is 0 you need to calculate `term = 4.0 / 1.0`, when `i` is 1 the term is `term = 4.0 / (1.0 + 2.0)`. If you look at the equation above, the denominator of this term is basically $1.0 + 2.0i$. **HINT:** as hinted at here, use `4.0` and `1.0` instead of `4` and `1`. The former will be interpreted as float/double expressions, and this will force C/C++ to perform the calculation as a `double` type. If you do something like `4 / 3`, C/C++ performs integer division, and you get a result of integer 1, instead of a real valued result like you expect.
4. If you correctly calculate the $i^{th}$ term, then you next just add this term into your running `sum`.
5. As given this function is a stub function that returns `0.0`. Once you are correctly calculating the sum of the series in your `for` loop, you should return the resulting `sum` as the result at the end after your `for` loop exits.

The tests for Task 4 test that you are correctly calculating the sum for all values of i from 0 to 5, and then tests some bigger i from the series summation.

If you successfully complete the function, and save and compile and rerun the tests, you should find that all of the unit tests for Task 4 pass successfully. When you are satisfied your code is correct, create a commit and push your commit to your GitHub classroom repository. It is always a good idea to check the autograder after pushing a commit to ensure it is also compiling and passing the tests for the task you just completed on GitHub.

## Task 5: Estimate $\pi$ using the Leibniz Series

If you successfully complete task 5, we can make an easy and small modification to use this summation to estimate $\pi$. The only difference is that we need to alternate adding and subtracting subsequent terms of the series instead of always just adding them into the sum as you did for Task 4.

As usual you should start by `#define` the `task5` tests, and creating the Task 5 issue in your GitHub classroom. Then start your work by copying the code you have from Task 4 in the `sumSeriesIncreasing()` into the `estimatePi()` function. This function takes the same parameter `n` as before.

Inside of the for loop, you need to add an `if` condition statement to perform alternating addition and subtraction. You are required to use the modulus operator `%` in this task to test if `i` is odd or even. For example, if you modulus by 2, then the remainder will be 0 for any value of `i` that is even (and the remainder will also correctly be 0 when `i=0`. Likewise when `i` is odd, the remainder will be 1. You can use this fact to create an `if / else` control structure that will add the current term for even `i` iterations, and will subtract the term for odd `i` iterations.

If you successfully complete the function, and save and compile and rerun the tests, you should find that all of the unit tests for Task 5 pass successfully. When you are satisfied your code is correct, create a commit and push your commit to your GitHub classroom repository. It is always a good idea to check the autograder after pushing a commit to ensure it is also compiling and passing the tests for the task you just completed on GitHub.

# Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is OK. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this assignment, up to 40 points will be given for having completed at least the initial Task 1 / In Lab task. Thereafter 15 additional points are given by the autograder for completing tasks 2 through 5.

## Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull request comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
    - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
    - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).
4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors.

Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.

6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

## Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Git Tutorials
- Git User Manual
- Git Commit Messages Guidelines
- Test-driven Development and Unit Testing Concepts
- Catch2 Unit Test Tutorial
- Getting Started with Visual Studio Code
- Visual Studio Code Documentation and User Guide
- Make Build System Tutorial
- Markdown Basic Syntax