Contents

CSCI 1300 Syllabus

1 About The Course

1.1 Course Description

Teaches techniques for writing computer programs in higher level programming languages to solve problems of interest in a range of application domains. Appropriate for students with little to no experience in computing or programming.

1.2 Course Objectives

This class covers the basics of computer programming using C++. The focus will be on understanding the basic components of computer languages. Upon completion of the course, the student can:

- Design and construct algorithms for problem solving by applying processes of abstraction and program decomposition.
- Implement fundamental programming constructs, such as variables, expressions, conditionals, and iterative control structures, in a higher-level language.
- Evaluate and implement simple I/O, such as user input and file I/O, including the necessity for external input to a program and the role of external data storage.
- Design and explain the role of functions in program construction, including an understanding of parameter passing and return values.
- Describe the properties of data types, including the primitive types of numbers, characters, and booleans, as well as more complex data types, such as arrays, records, and strings.
- Use an Integrated Development Environment to produce code that is free of syntactical, logical, and run-time errors. Understand the process of debugging as part of software development.
- Design and create code using object-oriented design methodology, including an understanding of objects and classes, information encapsulation, and efficient class design.
- Use third-party code to accomplish a programming objective. This includes learning to read code written by another individual and modifying the code, or using third-party libraries.
- Develop an understanding that software development is a dynamic, social process, and that learning how to seek out information is a necessary skill for success.

1.3 Course Outline

This is a 4 credit hour course. As such, you can be expected to spend 4 hours per week "in class" (3 in full class lecture, 1 in recitation) and 12 hours per week on additional course-related activities. These activities will include reading sections from the course text, watching pre-recorded topic videos, working on various assignments, engaging with course staff in grading interviews, and other activities.

Brief list of topics to be covered:

- Computer architecture and environment
- Variables and data types
- Variables and operators
- Strings: indexing, iteration, comparing
- Control structures: if/else statements
- Control structures: switch statements
- Control structures: while loops
- Control structures: for loops, iteration
- Functions
- Arrays
- I/O Streams
- File I/O
- Classes and objects
- Classes, functions, methods
- Inheritance
- Vectors
- Recursion
- References

2 Assignments and Grading

2.1 Assignments

Your grade is based on your performance in course activities including recitations, homeworks, quizzes, projects, and exams.

Your grade in the course will be based on the following components:

- **Homeworks (25%)**. There are 8 homework assignments that will asses your knowledge about the weekly concepts taught in lecture.
- **Projects (15%)**. You will complete two projects during the semester. Project 1 will be worth 5% of your grade, and Project 2 will be worth 10% of your grade. These assignments will build upon and integrate the concepts covered in previous coding assignments and give you opportunities to practice your skills on real-world problems.
- **Weekly recitation activities (10%, drop lowest)**. You will complete pre-quizzes before recitation and in-class worksheets during the recitation section. These are graded assignments and may often require working with a partner. Attendance in recitations is required.
- **Midterms and final (40%)**. There will be three midterms and a final exam that will evaluate your overall knowledge of the course material. Dates for these, and topics covered, are listed in the table below. Midterms will be held in-person during lecture, and the final exam will be held in-person during the scheduled final exam time. You must be present in-person for all exams, they will not be offered remotely.
- **Class participation (10%, drop 3 lowest)**. Most activities will take place during lecture. Other activities will be published via Canvas.

2.2 Extra Credit

There will be a number of other opportunities to receive extra credit in the form of additional points on an assignment or within a category. Take advantage of each and every opportunity.

2.3 Submission Policies

All assignments are due by the date and time specified in the assignment. Homework can be submitted up to two days late for 80% of full credit. After two days, no late work is accepted without prior permission of the instructor.

Recitations

Recitation pre-quizzes will include questions to get you adjusted to the content taught in lecture the previous week. Pre-quizzes will be due at **9:00 am on Mondays**. Recitations will then address these questions and provide the correct answers, and then in recitation you will complete a paper worksheet provided by your TA. The worksheets will be due at **11:59 pm on Wednesdays**.

Homeworks

There is a homework or project due every week. Homework will be due at **5:00 pm on Fridays**. Coding assignments will often be given immediate feedback using an auto-grade feature, but may also involve a grading interview process outlined below. Code files should have your name, date, and assignment number included as comments at the top of the file.

There will be a quiz associated with every homework and project. These quizzes will include both theory and code questions. Quiz will be due at 5:00 pm on Sundays.

Homework Late Policy:

All homework assignments, except Homework 0, may be turned in up to 2 days (48 hours) late with a 20% penalty. Late extension homework coderunner assignments will be available at 5:01 pm on Fridays and may be submitted as late as Sunday at 5:00 pm. If a student would have received a 95% had they turned in their homework on time, a late submission will earn them a 75% instead.

Interview Grading:

Following a project submission, each student will sign up for a grading interview about their project. If you are not interviewed for all projects, then you get no credit for those activities. The grading interview will consist of questions about the work you will have submitted the previous week (or two, for projects). These questions are designed to test your understanding of the solution code submitted. They will also serve as an opportunity for you check in about other topics related to the course. It is your responsibility to look on Canvas and sign-up for an interview slot. If you wait, and the interview slots dry-up, then you missed your interview. Sign-up early, do not wait.

If you have to reschedule a grading interview, you must email the TA at least one day in advance. Documented emergency situations will be evaluated on a case-by-case basis.

You are responsible for scheduling the grading interview during the designated grading period, which is usually 1 week. Once the period expires you will not be able to have the grading interview and you will lose all the points for the assignment. You are responsible for finding the room and arriving on time. There is a 1- minute "grace period" for being late, after that it is 10% off for each minute the student is late, at 6 minutes late you get a zero.

Advice: Get to the appointment 5 to 10 minutes early and use the extra time to prepare.

Exams

As discussed in the Assignments section above, there are three midterms and a final in this class that account for 40% of your final grade. **You must average at least 67% on your in order to receive better**

than a D+ in the class, regardless of your scores on the other aspects of the class. Each exam is 10% of your grade and your final exam can be used to replace your lowest midterm grade if it is higher than one of your midterms. The midterms are an individual assessment of your ability to apply the programming skills learned in lectures, recitation assignments, and homework assignments. If you do not show a skill level that is at or above this threshold, you will not receive a grade better than D+ in the course overall. A grade of C- in this class is required to take the next class in the computer science sequence. In addition, you will not get a passing score for the class by just showing up for the exams. You still need to turn in all other class work, including all projects.

Midterm 1	Friday, Oct 4
Midterm 2	Friday, Oct 25
Midterm 3	Friday, Nov 15

2.4 Grading Scale

At the end of the course, letter grades will be assigned via the following formula:

Letter Grade	Percentage Grade
A	93-100
A-	90-92.99
B+	87-89.99
B	83-86.99
B-	80-82.99
C+	77-79.99
C	73-76.99
C-	70-72.99
D+	67-69.99
D	63-66.99
D-	60-62.99
F	<60

3 Course Requirements

3.1 Method of Instruction

Attendance at all class meetings is highly recommended, and attendance in recitations is required.

10% of your grade is participation in lecture, which will include small in-lecture activities that you must be present to complete.

You are responsible for knowing the material presented during class, even if you were not in attendance when the material was presented. Previous experience has shown that students who do not attend class regularly often receive a failing grade and have to repeat the class the following semester.

Read the daily lecture material and watch pre-recorded topic videos before lecture. Lecture sessions will utilize discussion and activities that build on this content, along with the materials from previous days. Lectures will be recorded and posted to Canvas for later use within the course.

Recitation activities will include discussion and problem-solving, as well as provide a chance for addressing questions about course content. You will complete a Canvas quiz during recitation.

Your TA(s) will take attendance for recitation each week. If you need to miss recitation, make arrangements with your TA(s), to attend a different recitation section for the week. You can find the list of TA' s and the times for each recitation on the start here page. If you miss recitation, you will not get credit for the Recitation assignment that week.

Recitation will have similar expectations for lecture. Be professional, be on time, and do not be disruptive.

Good tips for maintaining a professional environment:

- Decide before lecture if you are going to attend.

- If you attend, be professional, be on time, and do not be disruptive.
- Turn off your cell phone.
- Bring your laptop, but restrict its use to class activity.
- Put away newspapers and magazines.
- Refrain from disruptive conversation.
- In general, respect all interactions related to this class and the students that desire to partake.

3.2 Required Texts and Other Materials

Textbook

Brief C++ Late Objects, Third Edition, Cay S. Horstmann, Wiley. 2017. ISBN: 978-1118674260.

This book is available in digital format and print format.

It is presented in an Enhanced E-text format readable on computers, tablets, and Kindle devices through Canvas. All Enhanced E-text sections include many different forms of guidance to help students build confidence and tackle the task at hand, including Self Check and Practice activities along with end-of-section Review Exercises, Practice Exercises and Programming Projects.

Content in the new edition is much improved, and if you choose an older edition you will miss out on the interactive activities and exercises.

Software

We will be using VS Code as our primary Interactive Development Environment (IDE). This installation process is supported in the Getting Started and Week 1 materials of the course.

We will also be using Canvas as our Learning Management System (LMS). Although you only need to do minimal work directly on Canvas, you will need to know how to navigate Canvas to access course materials, see announcements, submit your homework, understand your grades, and view course notes and project descriptions.

You will also need to be expected to communicate and collaborate on Ed. You can post your questions on Ed in the appropriate thread for each week, each assignment and each problem. Your questions will be answered by another student or one of the members of the instructional team. Please also answer any questions that you see on Ed. As a group we are much more efficient than any one individual.

Other Materials

You must have a back-up system. Lost work, internet was broken, computer froze up, etc. are not valid reasons for missing due dates.

For this course it is highly recommended that you get a Dropbox account, a Google Drive account or invest in a USB memory stick, to save your files. Computer crashes absolutely WILL happen and you are responsible for saving and keeping backup copies of your work. Also, Google Drive and Dropbox keep a versioning system. If you accidentally delete your file, you will be able to recover it.

A limited amount of printing may be required in this class. You need to ensure that your printing account has sufficient funds for this. Your initial allocation may deplete quickly, depending on your other printing activities. If this causes problems, please contact the course staff at csci1300@colorado.edu .

4 Collaboration

4.1 Collaboration, Plagiarism, and Honor Code

Collaboration is highly recommended and a valuable skill to develop at this point in your life.

Collaboration is defined as working with other people to ask, and answer, questions about how coding is done. This can be accomplished verbally or with pseudo-code or with pictures or with flow charts or with

discussions about what syntax means... Collaboration is NOT defined as showing someone your source code, or by cutting-and-pasting code, or by a group project where everyone talks but only one person codes, or by using a paid tutor to show you code, or by using internet sites to copy code.

Although you must work with other students to fully understand coding, you must never ever reveal your source code, or copy source code, or leave your browser open, or leave your computer unattended. Stolen code results in ALL involved parties being sent to honor council.

The Computer Science Department at the University of Colorado at Boulder encourages collaboration among students. You are encouraged to work with a partner for some assignments (Final Project), but you are also free to tackle the work alone.

Many forms of collaboration are acceptable and encouraged. In computer science courses, it is usually appropriate to ask others—TAs, LAs, instructor, or other students—for hints and debugging help or to talk generally about problem solving strategies and program structure. In fact, we strongly encourage you to seek such assistance when you need it. Discuss ideas together, but do the coding on your own .

Rule 1: You must not submit or look at solutions or program code that are not your own. Do not search for online solutions. This is not an appropriate way to "check your work," "get a hint," or "see alternative approaches."

Rule 2: You must not share your solution code with other students, nor ask others to share their solutions with you. Do not leave copies of your work on public computers nor post your solution code on a public website.

Rule 3: You must indicate on your submission any assistance you received. It is fine to discuss ideas and strategies, but you should be careful to write your programs on your own.

Be aware: all submissions are subject to automated plagiarism detection. The entire point of the CU Honor Code is that we all benefit from working in an atmosphere of mutual trust. Do not take advantage of that trust. CU employs powerful automated plagiarism detection tools that compare assignment submissions with other submissions from the current and previous quarters. The tools also compare submissions against a wide variety of online solutions. These tools are effective at detecting unusual resemblances in programs, which are then further examined by the course staff. The staff then make the determination as to whether submissions are deemed to be potential infractions of the Honor Code.

To support students in collaboration the Department has created a Collaboration Policy that makes explicit when their collaborative behavior is within the bounds of the Collaboration Policy and when it is actually academic dishonesty, which would be considered a violation of the University' s Honor Code. All students of the University of Colorado at Boulder are responsible for knowing and adhering to the University's Honor Code. Violations of this policy may also include cheating, plagiarism, academic dishonesty, fabrication, lying, bribery, and threatening behavior. Collaboration on homework assignments is allowed and encouraged. Students are most successful when they are working with other students to understand new concepts. The ultimate goal is that you fully understand the code you develop.

Plagiarism includes using material from outside sources (e.g. Internet sources, chatGPT, other AI tools, chegg.com, coursehero.com, a tutor) without clear identification and citation. Unless otherwise specified, you may make use of outside resources (internet, other books, other people), but then you must give credit by citing your sources in the comments inside your code.

Citing examples (assuming // indicates beginning of code comment):

- `// Modified version from https://github.com/Phhere?MOSS-PHP`
- `// Adapted from Program #7.2 in book "Accelerated C++" by Stroustrup`
- `// Worked with Joe Smith from class to come up with algorithm for sorting`
- `//Received suggestions from stackExchange website (see http://⋯)`
- `//Worked with a tutor on the algorithm for the STORE function`

A good rule of thumb: "if it did not come from your brain, then you need to attribute where you got."

Note: you do not need to cite if you are adapting from slides for the course or the required textbook for the course of from the hired staff for the course.

All homework assignments, all quizzes, all labs, and all exams will be required to be completed without outside resources . These will be clearly marked as individual assignments: the Canvas submission is individual. Use of outside resources would violate the collaboration policy.

4.2 Adhering to the Collaboration Policy:

Some examples of violating the collaboration policy include, but are not limited to:

- Sharing a file (source code) with someone else.
- Submitting a file that someone else shared with you.
- Stealing a copy of someone else' s work and submitting as your own, even with modification.
- Copying outside resources.
- Using outside resources and not citing your sources.
- Posting on websites like chegg.com. coursehero.com or craigslist.org soliciting a solution to an (or part of an) assignment.
- Soliciting help with commenting your code also constitutes a violation of the collaboration policy.
- Copying solutions from website like chegg.com, coursehero.com, and other websites.
- Using a solution the "tutor" gave you.

Examples of collaborating correctly :

- Sharing pseudo-code.
- Asking another student for a helpful suggestion. Verbally, not written.
- Reviewing another student' s code for bugs/errors.
- Working together on the whiteboard, or paper, to figure out how to approach and solve the problem. In this case you must include that person' s name in your collaboration list at the top of your submission. This includes working with a tutor.

One way to know you are collaborating well is if everyone is developing the code solution individually. This collaboration policy requires that you be able to create the code, or solve the problem on your own before you submit your assignment. You can brainstorm a solution or algorithm with a friend, but the submitted code should not be the same code.

Even if collaboration is stated, it is a violation of the Honor Code to submit effectively identical code with another student or an outside source.

If two or more students submit the same solution, claiming they have been "working with the same tutor" , that constitutes a violation of the Honor Code.

Any discovered incidents of violation of this collaboration policy will be treated as violations of the University' s Academic Integrity Policy and will lead to an automatic academic sanction in the course and a report to both the College of Engineering and Applied Science and the Honor Code Council. The academic sanction will be an automatic F.

Note: The instructor reserves the right to change the policy and to apply different academic sanctions if the violation justifies it. Students who are found to be in violation of the Academic Integrity Policy can be subject to non-academic sanctions as well, including but not limited to university probation, suspension, or expulsion.

Collaboration boundaries are hard to define crisply, and may differ from class to class. If you are in any doubt about where they lie for a particular course, it is your responsibility to ask the course instructor.

Students that leave their computers unprotected are also subject to course sanctions mentioned above. When similar code is found, ALL parties are subject to sanctions. This includes the person whose code was "taken" . Protect your source code at all times. Do not forget to sign out of public computers or leave your own machine unattended.

5 Communication

Please send all general course questions to: csci1300@colorado.edu

As a member of the CU community you are expected to consistently demonstrate integrity and honor through your everyday actions.

5.1 Professional Email Expectations

Any email correspondence related to the class should be sent from a colorado.edu email address. Please note that we do not read email between 5pm and 9am, or during the weekends. You can expect a response within 24 - 48 hours during the week and within 48 - 72 hours if sent on the weekend.

Send email messages to faculty and staff using a professional format.

Tips for a professional email include:

- Always fill in the subject line with a topic that indicates the reason for your email to your reader.
- Respectfully address the individual to whom you are sending the email (e.g., Dear Professor Smith).
- Avoid email or text message abbreviations.
- Be brief and polite.
- Add a signature block with appropriate contact information.
- Reply to email messages with the previously sent message. This will allow your reader to quickly recall the questions and previous conversation.

5.2 Office Hours

Faculty, TAs, and staff members are very willing to assist with your academic and personal needs. However, multiple professional obligations make it necessary for us to schedule our availability.

Suggestions specific to interactions with faculty and staff include:

- Respect posted office hours. Plan your weekly schedule to align with scheduled office hours. There is a variety of availability throughout the week.
- Avoid disrupting ongoing meetings within faculty and staff offices. Please wait until the meeting concludes before seeking assistance.

Our office hours are structured as group meetings.

Group office hours are on Monday through Friday, where multiple students can join in for help, and one or more members of the instructional team will be available to answer questions.

Please check out our Office Hours page on Canvas for up to date availability.

What are Office Hours?

It is much like a classroom. It is a place to ask questions, explore further, discussion strategies, explore related topics and support your classmates and contribute to the class. Office hours are optional.

How do we use Office Hours?

- Discuss class content directly with the instructor and other students.
- Practice professional collaboration strategies.
- Instructors may use breakout groups or/and guide discussions, at their discretion.
- Instructors may decide which topics will be discussed based on what will optimize learning.
- Please wear appropriate attire and be aware of your environment.

What are Office Hours not?

- Answer forum.
- Tutoring session.
- Instructors cannot help you completely debug your code.
- Instructor may not reply to all inquiries and let other students speak as appropriate.

5.3 Ed

We will use Ed as the center of communication for this course. Whenever you have content related questions about the course, especially ones that you believe other students may also have, you should post these questions on Ed. You can find the link on Canvas.

Week 1: Introduction

Learning Goals

This week you will:

1. Become familiar with the Syllabus, to learn about the structure of the course and its rules and policies
2. Install and become familiar with your Integrated Development Environment (IDE) - Visual Studio Code
3. Be able to identify the parts of a computer and their function in the computer system
4. Write the "hello world" program in C++

1 Background

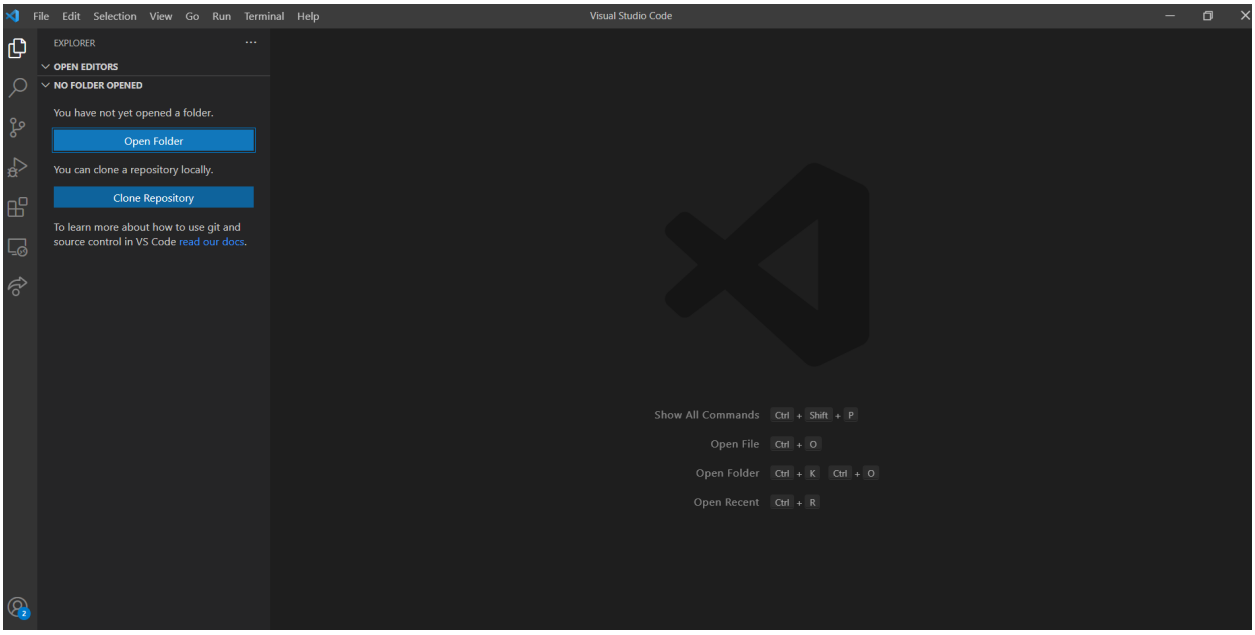
This week will be an introduction to a variety of foundational material for computer science. You will need to learn how to navigate the required software, basic variables, user input, and algorithms/pseudocode.

1.1 Navigating Terminals and Compiling Code

In Visual Studio Code, you can open an integrated terminal, which initially starts at the bottom of your workspace. But what is a terminal?

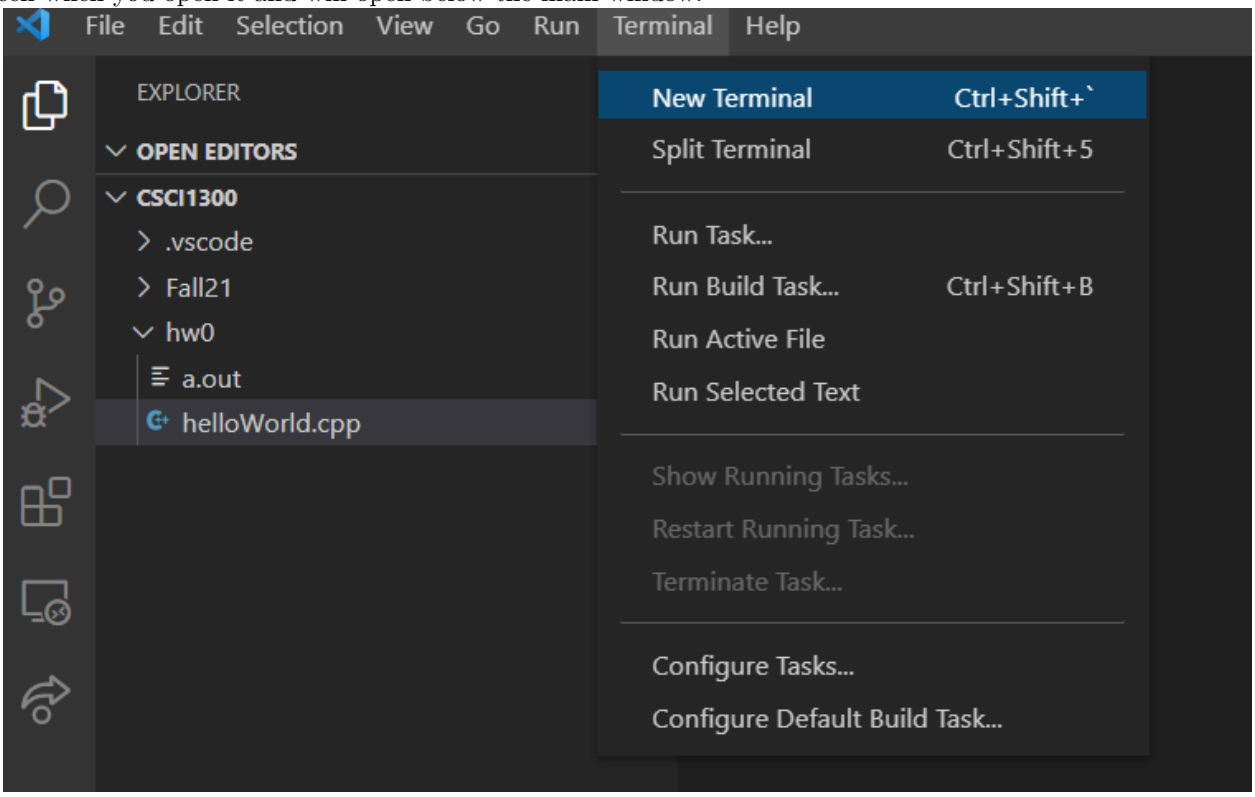
A **terminal** is an **interface** to your computer which allows you to execute any task on the computer directly through **commands**, without the use of a graphical user interface or GUI (like the file explorers on your system that you use to navigate to folders, create files etc). This allows you to directly execute tasks, which is often quicker than using the GUI.

VS Code has a built in terminal that you can open and use to interact with your computer's files and any code you write. To access this terminal, you will need to open VS Code. It may look like this the first time you open it:

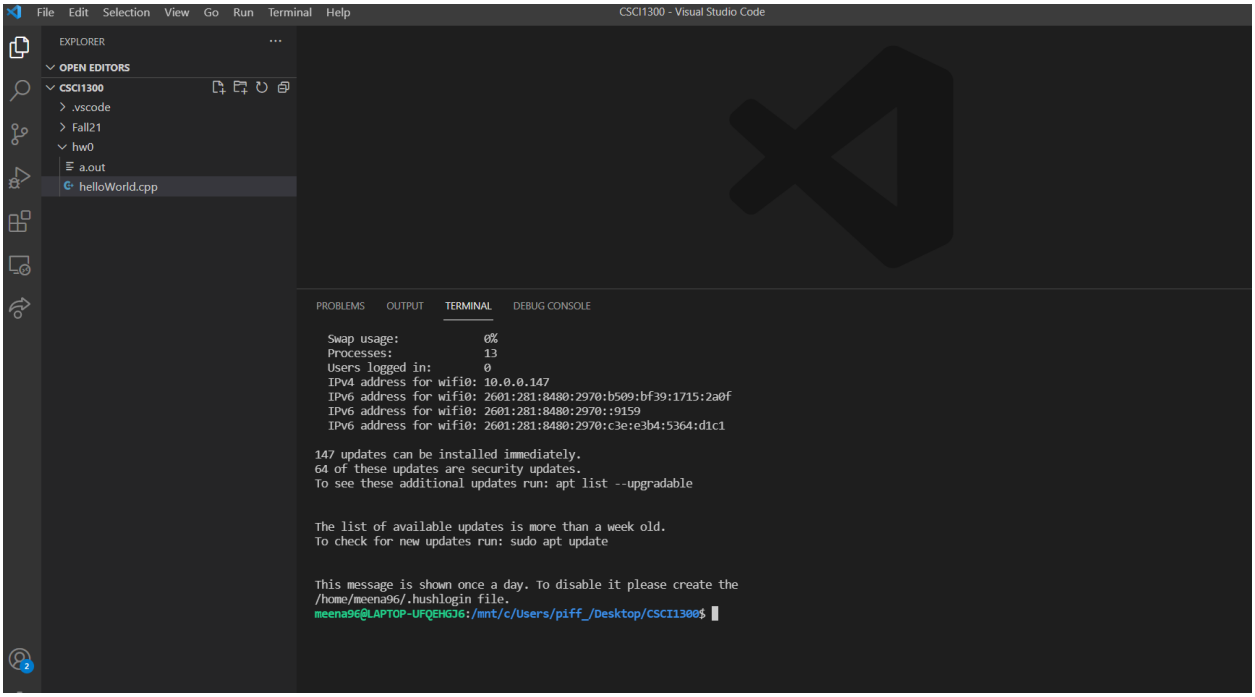


If you have not opened any folders yet in VS Code, you will need to start by opening one. You can do this by clicking "Open Folder" and navigating to whichever folder you would like to use. I would encourage you to make a new folder called "CSCI_1300" to keep all of your course materials organized.

Once you have a folder open, you can open a terminal. To open the Terminal tab, go to the menu bar, locate Terminal and click on it. Select "New Terminal" to open a new terminal. A terminal looks like a dark screen when you open it and will open below the main window.



The new screen should look something like this:



You will see your name and your device name on the terminal tab. Note that this can look different depending on your OS. The general anatomy of a terminal window is as follows:

1. Everything before the : tells you the username and the device name where you are logged in
2. Everything between the : and $ is your current directory (think of a directory as the folder you' re in). Note that you do not need to be in the same directory as the screenshots shown above. Yours will depend on your computer. Mac/Linux and Windows will look different, but both will state the current directory (or "folder") that you are in.
3. $ represents the end of the prompt, after which you can enter a command.

File browsing using the terminal is like using Windows explorer/Finder or clicking on folders and navigating to different folders on your machine.

In the terminal, instead of clicking on folders we use text commands to tell the computer what we want. If we want to go to a folder where we saved our last homework, we can type the commands to navigate to that folder and display its contents.

You can read file and folder locations by their "pathname", which is just a list of directories needed to get to your current location. Note: "folder" and "directory" mean the same thing, but most computer science texts will use the word "directory". There are two different ways of expressing pathnames that you will come across; one is a constant pathname, which describes how to access a location from anywhere. A second one is a relative path name, which tells you how to get to a location from your current location. If you are already in a folder. You may see these pathnames have something like "..", which tells the computer to go back a directory from the current location, but otherwise the style of these pathnames will look similar.

You will need to learn the basic text commands to navigate your computer system using the terminal. Important commands to learn include how to change which folder you are in, how to see the contents of your current folder, and more. Here is a list of commands and their meanings:

`ls`	list directory contents. Note, this a lowercase l, not an uppercase "i". This will 'list' everything in the current directory.
`cd [location]`	change directory, taking you to the specified [location].
`mkdir [name]`	make directory, which will make a new directory inside of your current location using the provided [name].
`cp [from here] [to here]`	copy a specified file from the first specified location to the second specified location.
`mv [from here] [to here]`	move a specified file from the first specified location to the second specified location.
`zip [zip file] [listed files]`	Zip all listed files into a new zip file called [zip file].
`rm [file name]`	permanently delete the file called [file name].

Note, the square brackets are not part of the command but only to illustrate where you should insert some other text.

Finally, you will need to know how to compile code from your terminal. Let us say that we have some code in a file called "helloWorld.cpp". In order to compile this program, we would have to navigate to the directory containing this file. Once there, you will enter the command:

```
g++ -std=c++17 helloWorld.cpp
```

`g++` is the compiler program.
`-std=c++17` is the version of C++ we want to use.
`helloWorld.cpp` is the file we wish to compile.
This command creates a file named a.out which is the compiled version of the code in helloWorld.cpp, which can be executed. You can run this compiled code using the following command:

```
./a.out
```

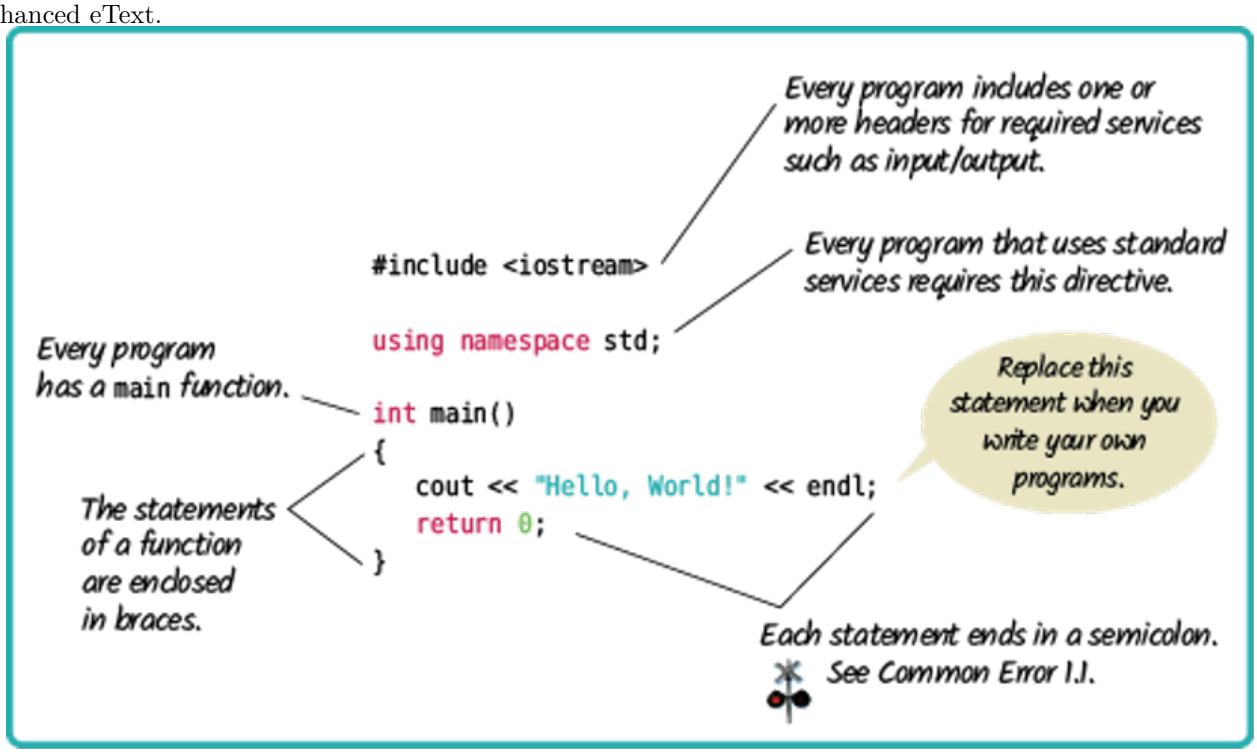
You can add the `-o` flag to your compiling command to give the output file a name. Additionally, in the near future you may also find it helpful to begin compiling with flags that tell you more information about possible errors in your code. These flags include:
`-Wall`
Wall is short for "Warn All", which will turn on most of the warnings in C++. This will help identify various possible ways your code might go wrong, including array bounds errors and other helpful messages.
`-Werror`
Werror will treat all warnings as errors. This will prevent you from skipping past the possible sources of error in your code, and you will need to make sure all warnings are resolved prior to compiling.
`-Wpedantic`
This flag enables warnings that alert you about language constructs that are not ISO C or ISO C++ standard compliant. This is particularly helpful to identify constructs that may not be uniform in other compilers, which could cause problems with your code on other machines. This will help prevent instances where your code works on your personal computer, but does not work on CodeRunner or on the grader's computer. All together, your command line prompt will look something like this:
`g++ -Wall -Werror -Wpedantic -o myName.out -std=c++17 myCodeFile.cpp`
Pro Tips:

- Tab Complete: if you're typing something in the command line that' s very long, but unique, you can hit tab when you're partially through and it will try to fill in the rest (kind of like autocomplete). If it doesn't, and you press tab twice, it tells you everything it has as options.
- Command history browsing: if you have typed a command and want to repeat it, just press the up arrow. It will bring up your last executed command. Pressing up again will go to the one before. Pressing down will go forward in time through the list.

1.2 Anatomy of a C++ Program

You will need a couple of components for the computer to be able to understand how to translate code that you can read into something it can actually do.

Here is a snapshot of a basic "Hello World" program from the course textbook, Brief C++: Late Objects, Enhanced eText.



2 Recitation

2.1 Software Setup

You will need to install and become familiar with the software we will be using this semester. Follow the appropriate installation guides in Appendix A for your particular computer (Windows or Mac) to install VS Code.

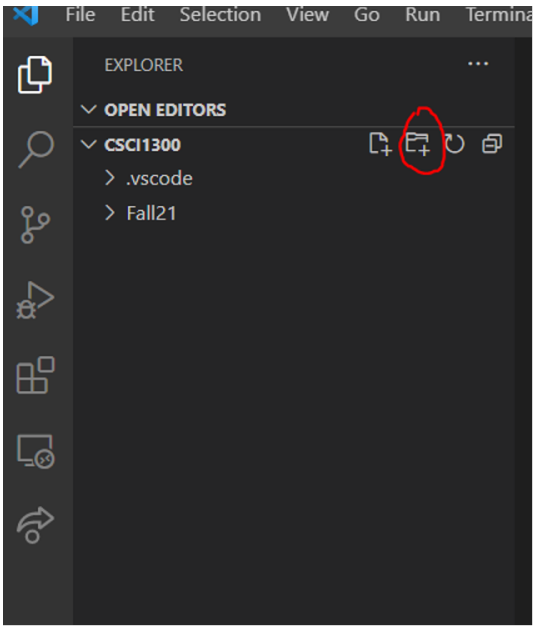
Once you have installed your software you will need to open VS Code and open the terminal. You will need VS Code and a working terminal. Take a screenshot and submit it on Canvas for this week's Recitation assignment.

3 Homework

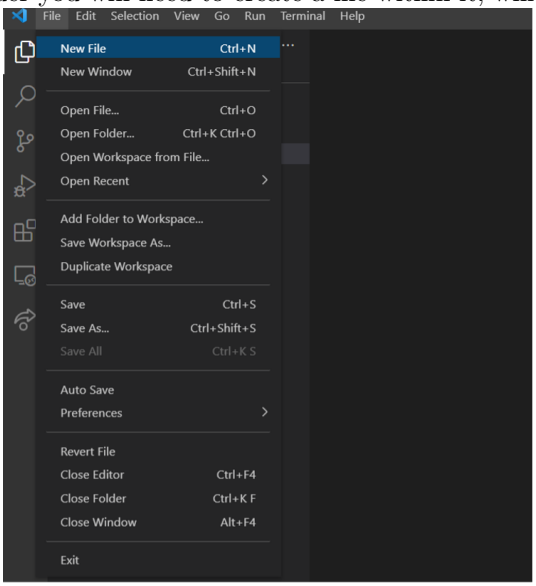
3.1 Hello World

The "Hello, World!" program is one of the simplest programs in a programming language, and it is often used to illustrate the basic syntax of a programming language. We will need to first create a folder to store our program file in and then create the file to write the program.

First, open VS Code. You can make a new folder by clicking this button:



Name the folder something intuitive, like "Week_1", "Recitation_1", or similar. Once you have your folder you will need to create a file within it, which you can do with the toolbar across the top:



You should name your file `helloWorld.cpp` – this file name structure is important. Files of uncompiled C++ code should end with `.cpp`, and the usage of a capital letter at the beginning of each word makes it legible even without spaces or punctuation. This style of writing names is often called "camel case".

Type the following code into your file:

```
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
}
```

Save your file. A quick shortcut to save is Cmd + S (In Mac) or Ctrl + S (in Windows). Now you will need to compile and run your program. You can review the background information to see how to do this. Once you have completed this and successfully run your program, take a screenshot of your VS Code window including both the file and the terminal.

3.2 Hello World: Improved

Now, let us explore this program a little more. Add a statement to use the standard namespace. Insert `using namespace std;` at the beginning of the code, then we can remove the `std::` prefixes. Your new code should look like this:

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello, world!" << endl;
}
```

Run the program again using the steps above. Take an additional screenshot.

3.3 Hello CSCI 1300!

Let's modify the file from Recitation to print "Hello world! Hello CSCI 1300". The text inside of the quotation marks is printed as it is. It's case sensitive too! To see the updated output, compile and run again.

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello, world! Hello CSCI 1300" << endl;
}
```

Once again, take a screenshot. For this section you should have three screenshots for the different Hello World programs.

Create a ZIP file that contains these three screenshots, and the three .cpp files you created, and submit it on Canvas.

Week 2: Variables and Operators

Learning Goals

This week you will:

1. Be able to identify and understand variable types and operations on variables
2. Implement variables and operations in a C++ program to solve a computational problem
3. Write arithmetic expressions and assignment statements in C++
4. Appreciate the importance of comments and good code layout
5. Create programs that read and process input, and display the results

1 Background

1.1 Variables and Operators

This week we learned about variables and basic arithmetic operators in computer science.

Variables in computer science are similar to variables you have seen in other math and science courses. A **variable** is a value that can change, depending on conditions or on information passed to the program.

A computer stores variables differently depending on the type of information they are meant to contain, so we must declare the type of variable. This table will tell you the names of different types of variables in C++:

Type of variable	Declaration in C++	Size	Description
Integer	int	2 or 4 bytes	a number that has no decimal value
Float	float	4 bytes	a number that has short decimal values
Double	double	8 bytes	a number that can have longer decimals
Character	char	1 byte	stores a single letter/character using ASCII values
Boolean	bool	1 byte	stores either True or False

In order to create and use a variable, you must declare it using the appropriate declaration from the above table and then give it a name. You may also provide it an initial value, but this is not required to create a variable. Some examples include:

Example 1.1.1. Here is an example of declaring variables:

```
int myNumber = 0;
double myDecimal = 3.1415;
char myEmptyChar;
```

After declaring variables, you can access them without repeating their data type. You can also perform basic operations, including the ones listed in this table:

Operator	Description	Example
+	Adds together two values	x + y
+=	Adds to itself and stores the new value	x += y
-	Subtracts one value from another	x - y
*	Multiplies two values	x * y
/	Divides one value by another	x / y
%	Modulus; this returns the division remainder	x % y
++	Increment; this increases the value of a variable by 1	++x
--	Decrement; this decreases the value of a variable by 1	--x

Example 1.1.2. Here is an example of declaring variables and using operators:

```
int myNumber = 4;
int mySecondNumber = 5;
int mySum = myNumber+mySecondNumber;
int myThirdNumber = 6;
mySum = mySum + myThirdNumber;
int myFourthNumber = 7;
mySum += myFourthNumber;
```

By the end of this code, the variable `mySum` would contain 4+5+6+7, or 22.

1.2 Characters

Characters are a variable type that is made of a single byte, which means they can encode for a fairly limited number of characters. We can identify which characters we can store using an ASCII table shown below.

Characters can be referred to by placing the chosen character in single quotation marks, like this:

```
char myChar = 'a';
```

Characters can also can be accessed using numerical values. Since every sequence of binary numbers that encodes a character could also instead be interpreted as an integer, we can use those numbers to identify which character we are talking about. The integer values that correspond with each letter are shown in the ASCII table.

There are a few traits that you may find helpful as you move forward: firstly, observe that all lower case letters are listed in consecutive alphabetical order, and secondly that all upper case letters are similarly listed in consecutive alphabetical order. This means that the space between a lowercase ‘a’ and a capital ‘A’ is the same as the space between a lowercase ‘b’ and a capital ‘B’ and so on. This is useful for converting characters from lowercase to uppercase.

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

1.3 Strings

In C++, a `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[2] << endl;  //prints the character 'l' to the screen
cout << s[9] << endl;  //prints the character 'r' to the screen
cout << s[12] << endl; //prints the character '!' to the screen
```

Note that when a character in a string is accessed with square brackets, the character has type char. For example:

```
string str = "Example"; //this is a string
char c = str[1]; //this is a char
```

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.

```
string s = "Hello, world!";
int s_length = s.length();
cout << s_length << endl; //This line will print 13 to the screen
```

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

```
s.length()
```

Common incorrect ways:

```
length(s)
s.length
```

Another useful function available for strings is `substr()`. This function allows us to access a subset, or a small portion, of a longer string. The substring function takes two arguments:

- The starting index of the substring you would like to capture
- The length of the substring you would like to capture (optional)

Note that the second argument is optional. If you don't pass a second argument to subtring, then the function will print the entirety of the string, beginning with the character at the position specified in the first argument. Note that `substr()` always returns a variable of type `string`, regardless of the length of the substring.

For example, consider the code below:

Example 1.3.1. Substring example code:

```
string str = "Coding is fun!";
cout << str.substr(0, 6) << endl;
cout << str.substr(6) << endl;
cout << str.substr(1, 1) << endl; //prints a string of length one
```

This will ouput the following:

```
Coding
 is fun!
o
```

Note: The second line of output begins with a space.

Both `length()` and `substr()` are special kinds of functions associated with objects, usually called methods, which we will discuss later in the course.

1.4 Terminal Input and Output

It is important to be able to take input from the user or to print output to the terminal as your program runs. You can do this using `cin` and `cout` for input and output, respectively.

The Hello World program you have seen provides an example of using `cout` to display output.

Example 1.4.1. Hello World for C++:

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello, world!" << endl;

    return 0;
}
```

You can similarly ask the user for a particular value and store that value in a variable using `cin`:

Example 1.4.2. An example of collecting user input to sum two integer variables and then printing their sum

```
#include <iostream>

using namespace std;

int main() {
    int firstNum;
    int secondNum;
    cout << "Please provide one number:" << endl;
    cin >> firstNum;
    cout << "Please provide a second number:" << endl;
    cin >> secondNum;
    cout << "The sum of these two numbers is " << firstNum + secondNum << endl;

    return 0;
}
```

1.5 Pseudocode and Algorithms

Pseudocode is used to develop algorithms. An algorithm is a procedure for solving a problem.

An algorithm describes actions to be executed and the order in which those actions are to be executed. In other words, an algorithm is merely the sequence of steps taken to solve a problem; like a recipe. An algorithm is not computer code. Algorithms are just the instructions which provide a clear path for you to write the computer code.

Example 1.5.1. An algorithm for adding two numbers together:
Step 1: Start
Step 2: Declare variables num1, num2, and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
Step 5: Display sum
Step 6: Stop

The main difference between an algorithm and pseudocode is that an algorithm is a step by step procedure to solve a given problem while pseudocode is a method of writing an algorithm. In other words, an algorithm is how to solve a problem while pseudocode is how to implement that solution.

Algorithm	Pseudocode
An unambiguous specification of how to solve a problem.	An informal high-level description of the operating principle of a computer program or other algorithm.
Helps to simplify and understand the problem.	A method of developing an algorithm.

Pseudocode is informal language that helps programmers develop algorithms (or recipes). Although there are no hard and fast rules for pseudocode, there are some suggestions to help make pseudocode more understandable and easy to read.

For instance, consider indenting all statements showing a “dependency” , like statements that use: While, do, for, if.

Several keywords are often used to indicate common input, output, and processing operations.

Input:	READ, OBTAIN, GET
Output:	PRINT, DISPLAY, SHOW
Compute:	COMPUTE, CALCULATE, DETERMINE
Initialize:	SET, INITIALIZE
Add one:	INCREMENT, BUMP

For looping and selection, the keywords that you might consider writing include:

- Do While...
- Do Until...
- Case...
- If⋯then...
- Call ... with (parameters)
- Call
- Return ....
- Return
- When

Try to indicate the end of loops and iteration by using scope terminators. For instance use if⋯(statements) ... endif.

Other words you may find useful while writing pseudocode include: Generate, Compute, Process, set, reset, increment, compute, calculate, add, sum, multiply, subtract, divide, print, display, input, output, edit, test, etc.

Be sure to indent if the indentation fosters understanding. Being clear is the purpose of pseudocode, and a very desirable goal to strive for.

Here are some examples of pseudocode:

Example 1.5.2. Will a grade pass?

```
If students grade is higher than or equal to 60
^^IThen Print, "Passed"
else
^^IPrint, "Failed"
```

Example 1.5.3. How do you find the area of a rectangle?

```
Read the length of the rectangle
Read the width of the rectangle
Compute the area of the rectangle as length times width.
```

You should write pseudocode whenever you are addressing problems in computer science. This allows you space to determine how to solve a problem without worrying about syntax and formatting, instead of having to figure out everything all at once.

2 PreQuiz

Problem 2.1. What is a variable?

Problem 2.2. Consider a variable tracking changing cost of an item on Ebay. What would be a good name for this variable?

Problem 2.3. Consider a variable that represents the roll of a 6-sided die. What possible values could this variable hold?

Problem 2.4. Consider a variable that tracks the temperature outside. What data type should you use? Are there multiple types that would work?

3 Recitation

3.1 Hello, Name!

You have recently learned how to accept user input and store it in a variable. Write a program where you ask the user for their name in the terminal, and then print `Hello, [name]!` in the terminal. You may find it helpful to start with the Hello World program from recitation.

A few example runs are shown below. In these examples, red text represents user-provided text (or text you would have to type in the terminal while running the program).

Sample Run 3.1.1

```
What is your name?
Mike
Hello, Mike!
```

Sample Run 3.1.2

```
What is your name?
samantha
Hello, samantha!
```

Problem 3.1.a. What steps would your program need to follow? Write these steps below.

Problem 3.1.b. Assume you have stored the user's name in a string variable called `name`. How would you write the cout statement to print `Hello, [name]!`?

Problem 3.1.c. Write out the steps above as a complete C++ program. Test your code. Does it work as expected? Keep testing until it does.

3.2 Converting Seconds to Days:Hours:Minutes:Seconds

A day has 86,400 seconds ($24 \times 60 \times 60$). Given a number of seconds in the range of 0 to 1,000,000 seconds, your program should print the time as days, hours, minutes, and seconds for a 24 hour clock. For example, 70,000 seconds is 0 days, 19 hours, 26 minutes, and 40 seconds. Your program should accept user input for the number of seconds to convert and then use that number in your calculations. Format your output as follows:

```
The time is W days, X hours, Y minutes, and Z seconds.
```

Problem 3.2.a. Explicitly list the variables you will need and their data types.

Problem 3.2.b. What arithmetic operators will be most useful for this problem?

Problem 3.2.c. Write out the steps you would use to solve this problem by hand as pseudocode.

Problem 3.2.d. Pick a random number between 0 and 1,000,000 for a sample run. Follow the steps you wrote from part c for this number to find your end result, and verify it. You can use your calculator, but make sure to write out each step.

Problem 3.2.e. Translate your pseudocode into a c++ program to solve the above code.

4 Homework

4.1 Identify and Correct the Errors

There are several snippets of code below where there is one error. You will need to identify and correct the error so that the code compiles. This code is also available directly in CodeRunner.

Problem 4.1.a. Spot the error:

```
#include <iostream>
using namespace std;

int Main()
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

Problem 4.1.b. Spot the error:

```
#include <iostream>
using namespace std;

int main
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

Problem 4.1.c. Spot the error:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, World! << endl;
    return 0;
}
```

Problem 4.1.d. Spot the error:

```
#include <IOstream>
using namespace std;

int main()
{
    cout << "Hello, World!" << endl
    return 0;
}
```

Problem 4.1.e. Spot the error:

```cpp
#include <iostream>
using namespace;
int main()
{
    cout << "Hello, World!" < endl;
}
```

4.2 Fahrenheit to Celsius Converter

Create a program that convert temperatures from Fahrenheit to Celsius.

$$Celsius = (Fahrenheit - 32) * (5.0/9.0)$$

The answer-box in coderunner is pre-loaded with following solution template for this question.

```cpp
#include <iostream>

using namespace std;

int main(){
    // declare all the variable
    double fahrenheit, celsius;

    // prompt the user & get their input
    cout << "<add question>" << endl; // EDIT THIS LINE TO PROMPT USER
    cin >> fahrenheit;

    // temperature calculation
    celsius = <add equation>; // EDIT THIS LINE TO CALCULATE TEMPERATURE
    // hint: use (5.0/9.0) instead of (5/9)

    cout << "The temperature is " << celsius << " degree Celsius." << endl;
    return 0;
}
```

Here are a few sample runs of the program, with user input shown in red:

Sample Run 4.2.1

```
What is the temperature in Fahrenheit?
32
The temperature is 0 degree Celsius.
```

4.3 Calculate the Falling Time of an Object

Create a program that calculates the time for a falling object to hit the ground based on the height from which the object fell. The equation for this is:

$$time = \sqrt{\frac{(2 * height)}{9.8}}$$

- **time** is the amount of time the object fell in seconds.
- **height** is the height the object was dropped from in meters.

Hint: cmath library contains a square root function that you can utilize.

You should print the time to two decimal places. You can accomplish this by using the iomanip library and the setprecision function. Here is a sample run of the program:

Sample Run 4.3.1

```
How far did the object fall in meters?
4
The object fell for 0.90 seconds.
```

4.4 Volume Of A Fish Tank

Write a program that calculates the volume of a fish tank. Ask the user for three values: the length, the width, and the height of the tank in inches. Use these values to calculate the total volume of the tank in cubic inches. Then, translate cubic inches into gallons and tell the user how many gallons can fit in the fish tank. Print this capacity to one decimal place.

Hint: 1 cubic inch is 0.004329 gallons.

Sample Run 4.4.1

```
What is the length of the fish tank in inches?
20
What is the width of the fish tank in inches?
20
What is the height of the fish tank in inches?
20
This fish tank has a 34.6 gallon capacity.
```

4.5 Pool Water Management

You're in charge of maintaining a swimming pool with a minor leak. The pool has some water in it already and needs to be filled up. However, due to the leak, the water level decreases slightly every hour at a constant rate. Write a program takes the hour as an input (as an integer) and predicts the pool's water level over time.

Pool	**Initial water level (inches)**	**Fill rate (inches/hour)**	**Leakage rate (inches/hour)**
Indoor Pool	19	0.6	0.4
Outdoor Pool	22	0.3	0.1

Sample Run 4.5.1

```
How many hours have passed?
22
The indoor pool has 23.4 inches of water, and the outdoor pool has 26.4 inches of water.
```

4.6 Barter System

Bartering is the exchange of goods and services between two or more parties without the use of money. Below is the table of conversion values:

Items	**Values**
1 chicken	6 avocados
1 avocado	2 watermelon
1 watermelon	4 potatoes

1 chicken is equal to 6 avocados, 1 avocado is equal to 2 watermelon, and 1 watermelon is equal to 4 potatoes.

Your program should take the number of potatoes as an input(integer) and converts its value to the maximum number of chickens, avocados, watermelons, and potatoes that can be bought.

Sample Run 4.6.1

```
Enter the number of potatoes:
200
Maximum number of chicken(s) 4, avocado(s) 1, watermelon(s) 0, potato(es) 0
```

Sample Run 4.6.2

```
Enter the number of potatoes:
100
Maximum number of chicken(s) 2, avocado(s) 0, watermelon(s) 1, potato(es) 0
```

Week 3: Booleans and Conditions

Learning Goals

This week you will:

1. Understand the Boolean data type and boolean operators
2. Understand relational operators
3. Be able to implement decisions using if statements

1 Background

1.1 Booleans

Booleans are a special data type that stores only "true" or "false". This true or false value can be stored in a boolean variable, or it can be the result of evaluating different expressions.

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.

Operator	Meaning	Example
>	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! = 6 is TRUE

Logical Operators

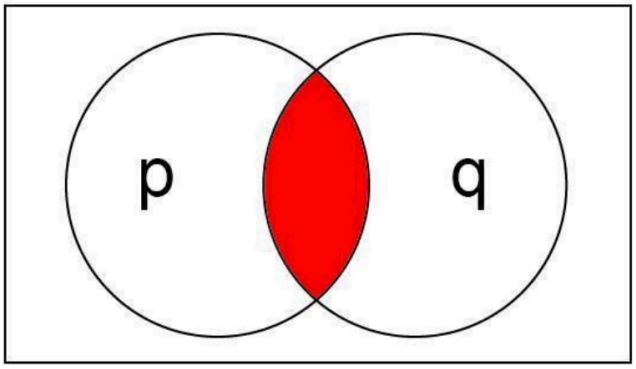
Logical operators are used to compare the results of two or more conditional statements, allowing you to combine relational operators to create more complex comparisons. Similar to relational operators, logical operators will evaluate to True or False based on whether the given rule holds for the operands. 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 the operand is false and false if the operand is true

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 truth tables for each of the logical operators specified above.

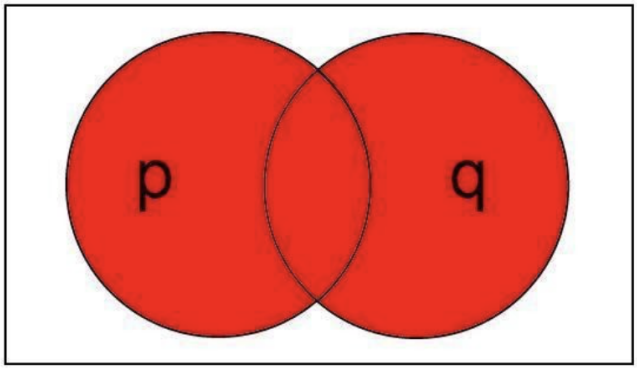
AND (`&&`): These operators return true if and only if both operands are True. This can be visualized as a venn diagram where the circles are overlapping.

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



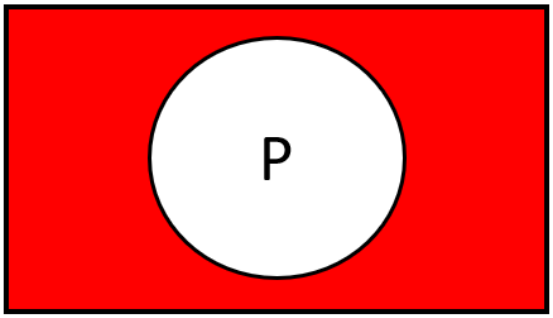
OR (`||`): These operators return True if one or both of the operands are True. This can be visualized as the region of a venn diagram encapsulated by both circles.

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



NOT (!): This operator returns the opposite of the operand. This can be visualized as the region of a venn diagram outside the circle. Unlike AND and OR, the NOT operator has only one operand.

p	!p
True	False
False	True



You can create truth tables for more complicated expressions by combining elements of these tables. You should begin with columns of the basic variables representing each possible combination of those variables, and then add columns to represent their modified values. For example, if you wanted to create a truth table for `!p && q` you could make a column for `p` and a column for `q` representing all possible combinations of true/false between the two variables. You can then create a third column for `!p`, and then perform the `&&` operation between the `!p` and `q` columns instead of the `p` and `q` columns, like this below:

p	q	!p	!p && q
True	True	False	False
True	False	False	False
False	True	True	True
False	False	True	False

For simple expressions, you can often work through the truth table in your head. However, knowing how to make truth tables will be helpful when you need more complicated expressions.

Using Booleans

There are two main ways you can use booleans: you can either assign them to a boolean variable, or you can use them directly as a condition (such as in an if statement). If you would like to evaluate a boolean expression and store it in a variable, you can do it like this:

```
bool myNewBoolean = (4 < 5); // this will evaluate to true
bool mySecondBoolean = (5 == 6); //this will evaluate to false
bool myFinalBoolean = (myNewBoolean && mySecondBoolean); //this will evaluate to false
```

You can string together increasingly complicated boolean equations either as a combination of boolean variables or as a combination of relational/logical expressions.

Booleans can also be represented using integers, and will print that way by default in C++. As an integer representation, 0 is false and 1 is true.

You can build if statements using boolean variables or boolean expressions.

1.2 Conditionals

Conditional statements, also known as decision statements or branching statements, are used to make a decision based on condition. A condition is an expression that evaluates to a boolean value, either true or false. Conditional Execution in C++ is a good online resource for learning about conditionals in C++.

You have seen one type of conditional expression already: switch statements. If statements, If/Else statements, and If/Else If/Else statements are a more complicated but also more dynamic way to make decisions in your code.

IF Statements

An if statement in C++ is composed of a condition and a body. The body is executed only if the condition is true. The condition appears inside a set of parentheses following the keyword "if" and the body appears within a set of curly brackets after the condition:

The general format for if statements is:

```
if ( <CONDITION> ){
    <BODY>
}
```

It is good practice to vertically align the closing curly bracket with the start of the if statement, and to indent the body.

The condition is interpreted as a boolean value, either true or false. Be careful, most expressions in C++ have a boolean interpretation. For instance, non-zero numeric values are true. Assignment operations (single equal sign) are interpreted as true as well. A common mistake is to use a single equals sign inside a condition when a double equals sign is intended.

Example 1.2.1. Here is an if statement that will check if a number is negative and change it to positive (i.e., find the absolute value):

```
if (num < 0){
    cout << "Changing sign" << endl;
    num = -1 * num;
}
```

IF-ELSE Statements

If statements may be paired with else statements in C++. If the condition associated with the if statement is false, the body associated with the else statement is executed. The else statement body is enclosed in a set of curly brackets:

```
 if ( <CONDITION> ){
    <BODY>
    // executed when CONDITION is true
}
else{
    <BODY>
    // executed when CONDITION is false
}
```

An if statement does not need an else statement, but there must be an if statement before every else statement.

Example 1.2.2. Here is an if/else statement that will check if a number can be a divisor before performing division:

```
if (num == 0) //notice the double equals!{
    cout << "Can't divide by 0!" << endl;
}
else{
    num = 1000 / num; //integer arithmetic
}
```

1.3 ELSE-IF Statements

Finally, an if statement may also be associated with any number of else-if statements. These statements each have an associated condition and an associated body. The body is executed if the condition is true and the conditions for all preceding if- and else-if statements in the same group are false. An else statement may be included at the end of the group but is not required. The else statement will be executed if all the previous conditions are false.

```
if ( <CONDITION> ){
    <BODY>
}
else if ( <CONDITION> ){
    <BODY>
}
else if ( <CONDITION> ){
    <BODY>
}
else{
    <BODY>
}
```

This is **not** logically the same as having multiple sequential if statements.

Example 1.3.1. These two if statements:

```
if ( <CONDITION A>){
   //do X
}
if ( <CONDITION B>){
   //do Y
}
```

are NOT logically the same as this if/else-if statement:

```
if( <CONDITION A>){
   //do X
}
else if ( <CONDITION B>){
   //do Y
}
```

In the first code section, both if statements are evaluated. If both CONDITION A and CONDITION B are true, we will do **both** X and Y. Meanwhile, in the second code block, if CONDITION A is true we will never evaluate CONDITION B, and therefore never do execute that code; here, we will **only** do X. Therefore, we need to use "else if" only when we want the two conditions to be mutually exclusive.

Example 1.3.2. Here is an if/else if/else statement to tell you if a number is positive, 0, or negative:

```
if ( num > 0 ){
    cout << "Positive" << endl;
}
else if ( num == 0 ){
    cout << "Zero" << endl;
}
else{
    cout << "Negative" << endl;
}
```

Nested If Statements

You can put if statements inside of other if statements (or if/else, or if/else if/else). The meaning of logical expressions can change when you are nesting if statements, so you should think through the truth tables for your if/else statements carefully.

```
if (booleanExpression1){
   //anything here will evaluate if booleanExpression1 is true
   if (booleanExpression2){
       //we will only evaluate this if statement if booleanExpression1 is true,
       //and then will only execute this statement if booleanExpression2 is ALSO true
   }
}
```

Nested if statements are essentially performing a logical "AND" operation on the two boolean expressions for the innermost if statement, but if only the first if statement is true you can still do other things.

1.4 Common Errors

Unintended behavior when accidentally using assignment operation (= instead of ==) in conditional statements:

Example 1.4.1. Here is some (incorrect) code:

```
int x = 5;
if (x = 1){ // one equal sign: changes value of x, will always evaluate to true

    cout << "The condition is true." << endl;
}
cout << "x is equal to " << x << endl;
```

The output of this would look like this:

Sample Run 1.4.1

```
The condition is true.
x is equal to 1
```

What you would ACTUALLY want is:

```
// CORRECT CODE
int x = 5;
if (x == 1) // two equal signs, performs comparison
{
    cout << "The condition is true." << endl;
}
cout << "x is equal to " << x << endl;
```

Which would output:

Sample Run 1.4.2

```
x is equal to 5
```

Remember, "=" is for assignment and "==" is for checking equality.

2 PreQuiz

Problem 2.1. What is the difference between arithmetic and relational operators?

Problem 2.2. What is the correct way to write a condition to determine if a variable `num` stores a number between 0 and 4, inclusive?

Problem 2.3. Fill in the blank in the code below with a condition that accounts for temperature in range of 25 (exclusive) to 50 (inclusive) degrees:

```
#include <iostream>
using namespace std;

int main()
{
    int temperature = 50;

    if (temperature > 85) {
        cout << "It's a hot day!";
    }
    if (temperature <= 85 && temperature > 50){
        cout << "It's a pleasant day.";
    }
    else if (_____________________________________________________) { //FILL IN THIS LINE
        cout << "It's a cool day.";
    }
    else {
        cout << "It's a cold day.";
    }

    return 0;
}
```

Problem 2.4. Complete the following truth table to evaluate the expressions for (a && !b).

a	b	!b	a && !b
T	T		
T	F		
F	T		
F	F		

Problem 2.5. Complete the following truth table to evaluate the expressions for (a || (!a && b)).

a	b	!a	!a && b	a || (!a && b)
T	T			
T	F			
F	T			
F	F			

3 Recitation

3.1 Spot The Error

Problem 3.1.a. The code snippet below is supposed to determine if a variable stores a value that is greater than, less than, or equal to 8. Identify the error(s) in the code below, and write the correct line(s).

```
#include <iostream>
using namespace std;

int main()
```

```
{
    int num = 6;

    if (num > 8) {
        cout << "The number is greater than 8." ;
    }
    else if (num = 8) {
        cout << "The number is equal to 8.";
    }
    else {
        cout << "The number is less than 8.";
    }

    return 0;
}
```

Problem 3.1.b. The code snippet below is supposed to determine if a variable stores a value for an angle that is obtuse, right, or acute. Identify the error(s) in the code below, and write the correct line(s).

```
#include <iostream>
using namespace std;

int main()
{
    int angle = 120;
    if (x>90) {
        cout<<"It is an obtuse angle." ;
    }
    elif(x=90) {
        cout<<"It is a right angle.";
    }
    else{
        cout<<"It is an acute angle.";
    }
}
```

Problem 3.1.c. The code snippet below is supposed to determine if a variable stores a value that is equal to zero or not. Identify the error(s) in the code below, and write the correct line(s).

```
#include <iostream>
using namespace std;

int main()
{
    int num = 7;

    if (num) {
        cout << "The number is zero.";
    }
    else {
        cout << "The number is not zero.";
    }

    return 0;
}
```

Problem 3.1.d. The code snippet below is supposed to determine if a variable stores a value that is equal to zero or not. Identify the error(s) in the code below, and write the correct line(s).

```
#include <iostream>
using namespace std;

int main()
{
    int num = 0;

    else {
        cout << "This is the 'else' block.";
    }
    if (num == 0) {
        cout << "The number is zero.";
    }
    else {
        cout << "The number is not zero.";
    }

    return 0;
}
```

Problem 3.1.e. The following code snippet is expected to accept a user provided integer and then state whether that number is even or odd. Identify the error(s) in the code below, and write the correct line(s).

```
#include <iostream>
using namespace std;

int main()
{
    int num;
    cout << "Provide an integer:" << endl;
    cin >> num;

    if (num/2){
        cout << "The number is even." << endl;
    }
    else {
        cout << "The number is odd." << endl;
    }

    return 0;
}
```

Problem 3.1.f. The following code snippet is expected to accept a user provided character and then state whether the corresponding grade passes or not. Identify the error(s) in the code below, and write the correct line(s).

```
#include <iostream>
using namespace std;

int main()
{
    char grade;
```

```
    cout << "Provide a grade (A, B, C, D, or F):" << endl;
    cin >> grade;

    if (grade == 'A' || 'B' || 'C'){
        cout << "This is a passing grade." << endl;
    }
    else if (grade == 'D'){
        cout << "This grade passes with conditions." << endl;
    }
    else {
        cout << "This is a failing grade." << endl;
    }

    return 0;
}
```

3.2 Step Tracking App

Your goal is to walk 10,000 steps every day but you aren't great at remembering to do it! So you decide to create a step tracking app that tracks your steps every day and will alert you based on how much you walked for the day. The program first asks how many steps you walked that day and then displays a message based on whether you have hit your goal for the day. Next, it will also tell you how many steps you have left to walk.

The following are the possible messages you will get based on your intake:

- If you've walked 5,000 steps or less, then you get:

  `You have not walked much today! Get those steps in! You have X steps left to walk.`

- If you' ve walked more than 5,000 steps but less than 10,000 steps, you get:

  `You're doing great, over half way there! You still have X steps left to walk.`

- If you've walked 10,000 steps or more, you get:

  `You've hit your goal for the day! Great job getting exercise!`

Note that **X** is the amount of steps left after subtracting how far you have walked.
Here is a sample run:

Sample Run 3.2.1

```
How many steps have you taken today?
3000
You have not walked much today! Get those steps in! You have 7000 steps left to walk.
```

Problem 3.2.a. Write an algorithm in pseudocode for the program above.

Problem 3.2.b. Imagine how a sample run of your program would look like. Write at least two examples.

Problem 3.2.c. Identify the values that you must test for. We call these values "boundary conditions".

Problem 3.2.d. Implement your solution in C++ using VS Code. Revise your solution, save, compile and run it again. Are you getting the expected result and output? Keep revising until you do. Make you sure you test for the values used in your sample runs, and for the boundary conditions.

4 Homework

4.1 Preparing for the Heat

You are planning to go for a run outside, but it's a hot day. Write a C++ program to determine whether you need to carry extra water based on the temperature.

The program should prompt the user to enter the temperature in degrees Fahrenheit. If the given temperature is above 85°F, display "You need to carry extra water." as it is considered hot weather for running, and then terminate. If the temperature is 85°F or below, the program will display "You don't need extra water." as carrying extra water may not be necessary, and then terminate.

Additionally, the program should perform input validation. If the user inputs a non-positive value for temperature (i.e., 0 or a negative number), the program should display "Invalid temperature." and terminate.

Sample Run 4.1.1

```
What is the temperature?
75
You don't need extra water.
```

Sample Run 4.1.2

```
What is the temperature?
90
You need to carry extra water.
```

Sample Run 4.1.3

```
What is the temperature?
-5
Invalid temperature.
```

4.2 Ordering Pizza

You've decided to order a pizza for dinner. The restaurant you're ordering from has three sizes of pizza: S, M, and L. Each size has a different base price and price per topping. The prices are indicated in the table below. Create a program to calculate the total cost of your pizza. The program should prompt you for the size of the pizza and the number of toppings, then output the cost.

Size	Base Price	Price Per Topping
S	8.00	0.99
M	10.00	1.99
L	14.00	2.99

The input should be a character (for size) and a non-negative integer (for the number of toppings). The program should accept both lowercase and uppercase inputs for pizza sizes (e.g., 's', 'm', 'l' or 'S', 'M', 'L'), and the output should be double.

Note: You can utilize the `toupper()` or `tolower()` functions to convert the input into either upper or lower case before comparing.

Note: The total cost should be formatted with a two-digit precision. You can use the `setprecision()` function with the fixed manipulator from `<iomanip>` library to do so.

Bad formatting: 10.8
Good formatting: $10.80
You will also need to perform input validation on the size of the cake – i.e., XL is not valid.

Sample Run 4.2.1

```
What size pizza would you like to order?
S
How many toppings do you want?
3
Your total is $10.97
```

Sample Run 4.2.2

```
What size pizza would you like to order?
XL
How many toppings do you want?
2
Invalid pizza size.
```

Sample Run 4.2.3

```
What size pizza would you like to order?
M
How many toppings do you want?
-2
Invalid number of toppings.
```

Sample Run 4.2.4

```
What size pizza would you like to order?
R
How many toppings do you want?
-2
Invalid pizza size and number of toppings.
```

4.3 Travel

You want to write a program that tells you if you can afford a road trip. In order to do this, you need to know your budget, how far you are driving, and how many nights you will stay.

To calculate the gas money, you estimate the road trip will cost you 16 cents per mile. After computing the gas money, you can determine your budget for each night. If you have less than $20 a night, you cannot afford to go. If you have at least $20 a night, you can afford to go camping during this trip. If you have at least $50 a night, you can afford a cheap motel. If you have at least $100 a night, you can afford a nice hotel.

You should perform basic input validation for all inputs by checking that they are non-negative numbers. If any of the inputs are invalid, the program should display "Invalid input(s)." Otherwise, the program should proceed with the calculations based on the provided values.

Sample Run 4.3.1

```
What is your budget?
500
How many miles will you drive?
800
How many nights do you want to spend there?
3
You can afford to stay in a nice hotel on this trip.
```

Sample Run 4.3.2

```
What is your budget?
500
How many miles will you drive?
800
How many nights do you want to spend there?
6
You can afford to stay in a cheap motel on this trip.
```

Sample Run 4.3.3

```
What is your budget?
100
How many miles will you drive?
800
How many nights do you want to spend there?
3
This trip is outside your budget.
```

Sample Run 4.3.4

```
What is your budget?
300
How many miles will you drive?
1200
How many nights do you want to spend there?
4
You can afford to go camping on this trip.
```

Sample Run 4.3.5

```
What is your budget?
-300
How many miles will you drive?
1200
How many nights do you want to spend there?
4
Invalid input(s).
```

4.4 Temperature Changes

You've decided to test the temperature outside every morning. Create a program that will tell you if the temperature over the last three days has increased, decreased, or neither. If the temperatures are increasing, print out "It's getting warmer!". If the temperatures are decreasing, then print out "It's getting cooler!". If the temperatures are not in any order or if two or more temperatures are the same, then print "The temperature is changing unpredictably."

The user should input 3 non-negative numbers (double) separated by spaces.

Sample Run 4.4.1

```
Enter temperatures over the last three days:
75.9 79.3 81.2
It's getting warmer!
```

Sample Run 4.4.2

```
Enter temperatures over the last three days:
45 30 18
It's getting cooler!
```

Sample Run 4.4.3

```
Enter temperatures over the last three days:
79 75 90
The temperature is changing unpredictably.
```

Sample Run 4.4.4

```
Enter temperatures over the last three days:
-79 75 90
Invalid temperature input.
```

4.5 Car Rental

You have decided to go on a weekend road trip with your friends. So, you plan to rent a car for this trip. The car rental company offers many options and has 4 categories. Each type of car has its specific base rate and price per day to rent the vehicle. Write a C++ program that calculates the total cost based on the car type and number of days.

Car Type	Base Price	Price per day
A	$80	$15
B	$110	$25
C	$160	$35
D	$220	$45

The total bill is calculated based on this formula:

$$\text{Total} = 1.23 \times (\text{base price} + \text{no. of days} \times \text{price per day})$$

The input should be a character (for car type), a non-negative integer (for the number of days you want to rent the car), and the output should be double.

Ensure you are doing input validation. The user should input car type from one among A, B, C, or D, and the minimum number of days to rent a car is 1. If the car type or the number of days is invalid, display "Please enter valid input." and exit the program.

Note: The total cost should be formatted with a two-digit precision. You can use the `setprecision()` function with the `fixed` manipulator from the `<iomanip>` library to do so.
Bad formatting: 10.8
Good formatting: $10.80

Sample Run 4.5.1

```
Which car type (A, B, C, or D) would you like to rent?
A
How many days would you like to rent this car?
6
Your total is $209.10
```

Sample Run 4.5.2

```
Which car type (A, B, C, or D) would you like to rent?
C
How many days would you like to rent this car?
11
Your total is $670.35
```

Sample Run 4.5.3

```
Which car type (A, B, C, or D) would you like to rent?
E
How many days would you like to rent this car?
1
Please enter valid input.
```

Sample Run 4.5.4

```
Which car type (A, B, C, or D) would you like to rent?
C
How many days would you like to rent this car?
-1
Please enter valid input.
```

Week 4: Functions

Learning Goals

This week you will:

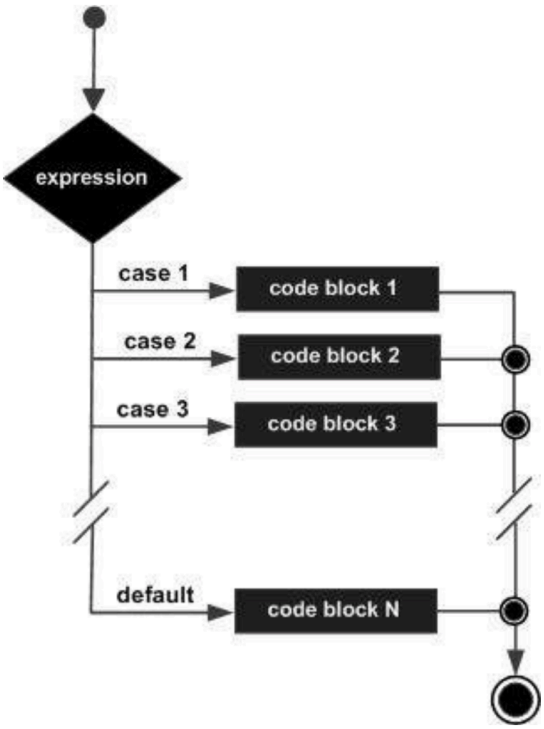
1. Understand switch statements
2. Understand basic functions:
   - Return types
   - Parameter lists
   - Function calls
3. Be able to identify the scope of a variable

1 Background

1.1 Switch Statements

Switch statements are an easy way to make decisions in a program. We can execute different sections of our code based on the value of a character or integer variable.

- If we are building a switch statement around an `int` variable, all of the cases must be defined using numbers.
- If we are building a switch statement around a `char` variable, all of the cases must be defined using characters. This means they must also use single quotes.



With the switch statement, the variable name is used once in the opening line. The **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.

An example of a simple switch statement:

Example 1.1.1. Switch statement syntax:

```
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 when using switch statements :

- The expression provided in the switch should result in a constant value otherwise it would not be valid.
  - `switch(num)`
    - allowed (`num` is an integer variable)
  - `switch('a')`
    - allowed (takes the ASCII Value)
  - `switch(a+b)`
    - allowed, where `a` and `b` are **int** variables, which are defined earlier

- 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.
- 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.
- The default statement is optional. Even if the switch case statement does not have a default statement, it would run without any problem.

Switch statements are a simple way to make decisions based exclusively on the equivalence of some characters or numbers. They are a simple form of **conditional statement**.

1.2 Functions

"Functions" in the context of math may be familiar, especially when formatted like this:

$$f(x) = x^2 + 5$$

Here we have labeled our function f, and it is a function of the variable x. You might have seen some where the functions are functions on multiple variables:

$$g(x, y) = x^2 + y^3$$

Functions in computer science are similar, but we can abstract them further so they apply to more than just numbers.

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.

There are two types of functions:

- Library functions
- User-defined functions

Library functions refer to pre-existing functions that you can use but did not write yourself. In order to use a library function, you must include the library that contains the function. For example, the C++ math library provides a `sqrt()` function to calculate the square root of a number. To use the `sqrt()` function, you must include the cmath library at the top of your program, e.g. `#include<cmath>`. 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 return type. A function may also return nothing, in which case its return type is void.

Here is the syntax for a function definition:

```
returnType functionName(parameterList)
{
    //function body
}
```

- The `returnType` is the data type that the function returns
- `functionName` is the actual name of the function

- `parameterList` 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. Note: this can be a list of multiple items separated by commas.
- `//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.

This may not immediately look like the functions you have seen in math, but they are actually quite close. Here is an example of the functions above written in C++:

Example 1.2.1. The function $f(x) = x^2 + 5$ written in C++:

```
double f(double x){
    return (x*x+5);
}
```

Example 1.2.2. The function $g(x, y) = x^2 + y^3$ written in C++:

```
double g(double x, double y){
    return (x*x+y*y*y);
}
```

I used `double` for the parameters and the return type because these functions in math would probably use decimals, but you could also use integers if you only wanted to use whole numbers. We can also use functions inside of other functions. Instead of just multiplying variables by themselves to do the exponents, we could use the `pow` function from `cmath`:

Example 1.2.3. The function $f(x) = x^2 + 5$ written using cmath:

```
#include<cmath>

double f(double x){
    return pow(x, 2)+5;
}
```

Example 1.2.4. The function $g(x, y) = x^2 + y^3$ written using cmath:

```
#include<cmath>

double g(double x, double y){
    return pow(x, 2) + pow(y, 3);
}
```

If you wanted to then use these functions to perform some calculations, you could use any value of x and y. You could use constant values, or you could use variables. Here are examples of ways you could use the function:

Example 1.2.5. Examples of calling the function $g(x, y)$:

```
#include<cmath>
```

```cpp
#include<iostream>

using namespace std;

double g(double x, double y){
    return pow(x, 2) + pow(y, 3);
}

int main(){
    double firstValue = g(4.0, 5.0);
    double secondValue = g(2, 3);
    double myX, myY;

    cout << "What value would you like for X?" << endl;
    cin >> myX;
    cout << "What value would you like for Y?" << endl;
    cin >> myY;
    cout << "The result would be " << g(myX, myY) << endl;

    return 0;
}
```

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. We can illustrate the order in which code executes with this example:

Example 1.2.6. An example with a simple function:

```cpp
#include <iostream>
using namespace std;

//funtion to add two numbers
int sum( int num_one, int num_two)
{
    int result = num_one + num_two;
    return result;
}

//main function
int main()
{
    //declare parameter value
    int parameter_var = 1;

    //call the function
    int sum_result = sum(parameter_var, 99);

    cout << "The sum is " << sum_result << endl;

```

```
22      return 0;
23  }
```

The code will begin executing in the main function – in this case, the first step is declaring a variable in line 15. When the program execution reaches line 18, the main function will pause, and the first line in the body of the `sum()` function will begin running. After the line `return result;` is reached, `sum()` will stop running, and the main program will resume execution where it left off. In this case, when `main()` resumes execution, the return value of `sum()` will be stored in `int sum_result`, and then the last two lines of the main function will run.

Functions can do more than just perform calculations; they can also perform operations on other variables, or they can be used to prevent yourself from copy/pasting the same code multiple times in your program. They do not even always need parameters. For example, if you wanted to write out a selection menu, you could write a function to print the menu so you do not have to type it out every time. In these cases, the parenthesis can be empty, like they are for our main functions.

As a general note: the function names f and g are not very good function names. You should generally pick better (clearer) names, and you should choose a naming style to be consistent. The naming style you choose should be different from how you name variables, so it is easier to read your code.We recommend using camel case to name your functions, like so:

Examples of function names we might use: `circleArea(), sumList(), findCapitalLetters()`

1.3 Testing Code

You must naturally test your code to make sure that it works correctly, and that it works correctly **all** the time. This may seem like a very difficult task, but there are several steps you can take to make sure you start correctly.

- Come up with a handful of test cases to use on your program.
- Consider the ways a user could use your program incorrectly. Use this to develop additional test cases.
- Consider extreme values, or "boundary conditions". Use these boundary conditions to develop yet more test cases.
- Test each part of your program independently. This is called *Unit Testing*. Do you have individual functions? Test each of them. Do you have major steps in your main function? Test each of them.

Boundary conditions are significant for many complex problems, and should test the extreme limits of what your problem may be applied to. Example: Are you supposed to examine a string? What happens if that string is empty? What happens if that string is thousands of digits long?

You should develop test cases for each boundary condition you can come up with. If there are several components to your program, you may need to identify boundary conditions for each of these components. Testing these boundary conditions of these pieces individually – whether they are functions or just blocks of code in your main function – is often easier than testing every independent combination of them.

Consider a program where you have 4 functions, and each function has 3 boundary conditions. To test each boundary condition for each of the 4 functions, you would need 12 tests. If however you only tested the program as a whole, you might need to check each combination of boundary conditions, which would end up becoming $3^4 = 81$ different tests. This is part of why **unit testing** is valuable.

As you develop extra functions, you should start by using your `main` function to test these functions.There will be 3 different types of test cases you should be expected to write depending on the return type of the function. Listed below is how we expect you to test different types of functions. The process will be different for if you are testing a `void` function, non-void functions that return an `int` or `bool`, and non-void functions that return a `double`.

Testing Void Functions

For void functions that have printed output (i.e. functions that use cout to print to the terminal), call the testing function in the main function. Your tests should include the expected output in comments. For these functions you will want to make sure that all expected outputs are successfully printed.

See the example code below:

Example 1.3.1. This is testing a function that prints whether a grade is passing or not.

```
void checkGrade(char grade){
    switch(grade){
        case 'a':
        case 'b':
        case 'c':
            cout << "You passed!" << endl;
            break;
        case 'd':
            cout << "You did not pass, but you were close." << endl;
            break;
        case 'f':
            cout << "You failed." << endl;
            break;
        default:
            cout << "That is not a valid grade." << endl;
    }
}

int main(){
    checkGrade('b'); //Should output "You passed!"
    checkGrade('d'); //Should output "You did not pass, but you were close."
    checkGrade('f'); //Should output "You failed."
    checkGrade('m'); //Should output "That is not a valid grade."
}
```

Testing Integer/Boolean Functions

For non-void functions that return a `bool` or `int`, use an `assert` statement from the `cassert` header (`#include <cassert>`) with a conditional expression.

Assert tests contain a conditional expression which will be true unless there is a bug in the program. If the conditional expression evaluates to false, then your program will terminate and show an error message.

For immediate purposes, functions that return a `bool` or `int` can be compared to a specific value using the equality operator ==.

Your test will look something like this:

`assert(<function call> == <value to compare to>);`

- `<function call>` is where you will call the function you want to test with its function parameters.
- `<value to compare to>` is the value you expect the function to return.
- == is the equality operator, and it compares the equality of both sides of itself.

See the sample code below:

Example 1.3.2. The below code shows examples of how to test integer functions with a simple addition function:

```
#include <iostream>
#include <cassert>
using namespace std;

int addInts(int num1, int num2)
{
    // add num1 and num2 before returning
    return num1 + num2;
}

int main()
{
    // test 1 for addInts
    assert(addInts(5, 6) == 11);
    // test 2 for addInts
    assert(addInts(10, 10) == 20);
}
```

Testing Double Functions

For non-void functions that return a double, use an `assert` statement from the `cassert` header (`#include <cassert>`) with a conditional expression and include the following function in your program:

Example 1.3.3. This is a required function to successfully test Double functions in C++:

```
/**
 * doublesEqual will test if two doubles are equal to each
 * other within two decimal places.
 */
bool doublesEqual(double a, double b, const double epsilon = 1e-2)
{
    double c = a - b;
    return c < epsilon && -c < epsilon;
}
```

Because the **double** type holds so much precision, it will be hard to compare the equality of a function that returns a **double** with another **double** value. To overcome this challenge, we can compare **double** values within a certain range of precision or decimal places. The function above compares the equality of two values `a` and `b` up to two decimal places. This function returns `true` if the values `a` and `b` are equal with each other up to two decimal places.

You will be expected to use this function in conjunction with assert statements to test functions that return the type double.

Your test will look something like this:

`assert(doublesEqual(<function call>, <value to compare to>));`

- `<function call>` is where you will call the function you want to test with its function parameters
- `<value to compare to>` is the double value you expect the function to return.

See the sample code below:

Example 1.3.4. This is code to test a function that finds the reciprocal of a value (i.e., divides 1 by that number).

```
#include <iostream>
#include <cassert>
using namespace std;
/**
 * doublesEqual will test if two doubles are equal to each other within
 * two decimal places.
 */
bool doublesEqual(double a, double b, const double epsilon = 1e-2)
{
    double c = a - b;
    return c < epsilon && -c < epsilon;
}
/**
 * reciprocal returns the value of 1 divided by the number
 * passed into the function.
 */
double reciprocal(int num)
{
    return 1.0 / num;
}
int main()
{
    // test 1 for reciprocal
    assert(doublesEqual(reciprocal(6), 0.16));
    // test 2 for reciprocal
    assert(doublesEqual(reciprocal(12), 0.083));
}
```

General Testing Tips

You will certainly come to a time in your coding career when your code does not work, and you just cannot figure out *why*.

In these times, there are a few possible options. First, your algorithm may be incorrect. If that is the case no amount of code testing will help you, and you will need to go back and think through how to solve the problem. If your algorithm is correct but your code is not, here are three tips:

1. If your code does not compile, start commenting out sections of your code. Keep going until it compiles, even if you have to go all the way back to an empty main function. Then, you can uncomment sections of your code until it fails to compile again. This will help you pinpoint *where* the issue is in your code, and once you know where it is you will be able to see *what* it is.
2. If your code compiles but has unexpected runtime errors, add output statements periodically throughout your code. When your program fails, you will know where your code stopped running based on which output statements failed to print.
3. If your code compiles and runs completely but the output is incorrect, go back through your code and print out the significant variables at each step in your code. You can then compare this to the test cases you worked out by hand and see where the code output differs from your algorithm.

2 PreQuiz

Problem 2.1. Select True or False:

A) **T/F:** When writing switch statements in C++ you must include a default case, otherwise your switch statement is not valid.

B) **T/F:** Switch statements can be built on integers, floats, and characters.

C) **T/F:** There is no difference between an if/else-if/else-if chain and an if/if/if chain in C++.

Problem 2.2. Given two conditions, `cond1` and `cond2`, the expression if `(cond1 && cond2)` will evaluate to true only if _____`cond1` and `cond2` are true. On the other hand, the expression if `(cond1 || cond2)` will evaluate to true if _____`cond1` or `cond2` is true.

Problem 2.3. How can you write a switch statement where multiple cases will execute the same block of code? Provide an example.

Problem 2.4. Fill in the blank(s) for the code below:

```
int choice;
cout << "Help our adventurer discover what to do next! Enter 1, 2, or 3." << endl;
______ >> choice;

switch(____________){
    case ________:
        cout << "You found a hidden treasure!" << endl;
        break;
    case 2:
        cout << "You discovered a secret passage!" << endl;
        __________
    case 3:
        cout << "You found a mystical artifact!" << endl;
        break;
    ________:
        cout << "That's not a valid case. Please try again." << endl;
}
```

3 Recitation

3.1 Spot The Error

Problem 3.1.a. Below is code that asks the user for the day of the week as a number (Monday is 1, Sunday is 7) and then prints a corresponding statement. Identify the error(s):

```
int day;
cout << "What number day of the week is it?" << endl;
cin >> day;
```

```
switch (day) {
  case '6':
    cout << "Today is Saturday";
    break;
  case 7:
    cout << "Today is Sunday";

  default:
    cout << "Looking forward to the Weekend";
}
```

Problem 3.1.b. Below is code with the same goal as the previous question, but different error(s). Identify the error(s):

```
int day = 4;
switch (day)
  case 6:
    cout << "Today is Saturday";
    break;
  case 7:
    cout << "Today is Sunday";
    break;
  default
    cout << "Looking forward to the Weekend";
```

Problem 3.1.c. The code below is meant to determine if an angle is acute, obtuse, or right. Spot the error(s):

```
#include <iostream>
using namespace std;

int main()
{
    int angle =40;
    if (x<90) {
        cout<<"It is an acute angle." ;
    }
    else if(x=90) {
        cout<<"It is a right angle.";
    }
    els{
        cout<<"It is an obtuse angle.";
    }
}
```

Problem 3.1.d. The code below implements an exclusive OR logical operation, which means that only one of the conditions may be true. Spot the error(s):

```
// This program implements XOR
#include iostream
using namespace std;

//Set the variable value to 1 when x or y is 1
int main(){
    int x = 1,y=0,value;
```

```
    if (x == 1){
        if(y==0)
        value = 1;

        else
        y == 0;

    if(x==0){
        if(y==0)
        value = 0;

        else
        value = 1;
    }

    cout < value < endl;
    return 0;
}
```

3.2 Final Velocity of a Rocket

Write a C++ program that will calculate the final velocity of a rocket after 20 seconds. The program will ask the user for the initial velocity (m/s) and the fuel type (A, B, C). The rate of acceleration will depend on the type of fuel and the initial velocity.

- If initial velocity is less than 10, then the acceleration rate for each fuel type is as follows
  - Fuel type A → 5 (m/s) per second
  - Fuel type B → 10 (m/s) per second
  - Fuel type C → 20 (m/s) per second
- If initial velocity is greater than or equal to 10 and less than or equal to 40, then the acceleration rate for each fuel type is as follows
  - Fuel type A → 6 (m/s) per second
  - Fuel type B → 12 (m/s) per second
  - Fuel type C → 24 (m/s) per second
- If initial velocity is greater than 40, then the acceleration rate for each fuel type is as follows
  - Fuel type A → 3 (m/s) per second
  - Fuel type B → 6 (m/s) per second
  - Fuel type C → 9 (m/s) per second

Below are some sample runs. User input is shown in bold.

Sample Run 3.2.1

```
Enter the initial velocity:
70
Enter the fuel type:
C
The final velocity is 250 m/s.
```

Sample Run 3.2.2

```
Enter the initial velocity:
5
Enter the fuel type:
A
The final velocity is 105 m/s.
```

Problem 3.2.a. Write out the steps you would use to solve this problem by hand as pseudocode.

Problem 3.2.b. Pick possible inputs for your program. Follow the steps you wrote for these values to find your result, and verify it.

Problem 3.2.c. Identify two possible values that are "boundaries" in this problem that you will have to test. What should happen for these values?

Problem 3.2.d. Translate your pseudocode into a c++ program to solve the above code.

4 Homework

4.1 Car switch

Write a program that accepts a single character representing an automobile manufacturing company as input from the user. Then, the program should print out the text for the appropriate company.

- Your program prompts the user with "Enter the first letter of the company: ", which asks for a character input.
- The input must be case sensitive, e.g. if the user enters 'b' instead of 'B', the output should be invalid and not "BMW".
- Your program prints output according to the following
  - If the input is 'B', print "BMW"
  - If the input is 'V', print "Volkswagen"
  - If the input is 'H', print "Honda"
  - If the input is 'T', print "Tesla"
  - Any input value that is not 'B', 'V', 'H', or 'T' print "Invalid"

Note: Code should NOT contain any if-else statements and should only utilize Switch Statements

Sample Run 4.1.1

```
Enter the first letter of the company:
B
Automobile manufacturer chosen: BMW
```

Sample Run 4.1.2

```
Enter the first letter of the company:
V
Automobile manufacturer chosen: Volkswagen
```

Sample Run 4.1.3

```
Enter the first letter of the company:
A
Automobile manufacturer chosen: Invalid
```

Sample Run 4.1.4

```
Enter the first letter of the company:
T
Automobile manufacturer chosen: Tesla
```

4.2 Instrument Price

You want to learn to play an instrument, but you need to know how much it will cost to buy it. The music store has the following table on their website:

Category	Instrument	Price
Brass	Trumpet	$570
	Trombone	$500
Woodwind	Flute	$425
	Saxophone	$225
Percussion	Snare Drum	$275
	Cymbals	$350

Write a menu-driven program that asks the user to input an instrument category and then an instrument. The program should give the user the price.

The user should input an integer in the range of the choices you give them (for example, a user cannot input 3 if you only have 2 choices). If the user inputs the correct range, the program should be prompted for the next set of choices. Once they make the final selection, the total should be printed to them as an integer with proper formatting, as shown in the sample run.

Note: Code should NOT contain any if-else statements and should only utilize Switch Statements

Sample Run 4.2.1

```
Select a category: (1)Brass (2)Woodwind (3)Percussion
1
Select an instrument: (1)Trumpet (2)Trombone
2
Your instrument will be $500
```

Sample Run 4.2.2

```
Select a category: (1)Brass (2)Woodwind (3)Percussion
3
Select an instrument: (1)Snare Drum (2)Cymbals
2
Your instrument will be $350
```

Sample Run 4.2.3

```
Select a category: (1)Brass (2)Woodwind (3)Percussion
5
Please enter a valid input.
```

Sample Run 4.2.4

```
Select a category: (1)Brass (2)Woodwind (3)Percussion
2
Select an instrument: (1)Flute (2)Saxophone
1
Your instrument will be $425.
```

4.3 Movie Night

You are preparing a movie night with your friends. Create a C++ program that helps you determine which movie to select.

Based on the given table, you, as the programmer, should give step-by-step choices to the user to proceed

further and make a movie selection. Once the user has selected the movie, the program should display a message: "You have selected the movie: <movie title>" where "<movie title>" is the movie the user has selected.

Additionally, ensure that the user's input is present in the range of choices. If the user inputs an invalid option, print "Please enter a valid input" and terminate the program.

Genre	Directors	Movies
(1) Animation	(1) Pete Docter	(1) Monsters, Inc.
		(2) Inside Out
	(2) Brad Bird	(1) The Incredibles
		(2) Ratatouille
	(3) Andrew Stanton	(1) Finding Nemo
		(2) WALL-E
(2) Adventure	(1) Steven Spielberg	(1) E.T. the Extra-Terrestrial
		(2) The BFG
	(2) Jon Favreau	(1) The Jungle Book (2016)
		(2) Elf
	(3) Robert Zemeckis	(1) Back to the Future
		(2) Who Framed Roger Rabbit

Sample Run 4.3.1

```
Select the genre: (1) Animation (2) Adventure
1
Select the director: (1) Pete Docter (2) Brad Bird (3) Andrew Stanton
1
Select the movie: (1) Monsters, Inc. (2) Inside Out
1
You have reserved the movie: Monsters, Inc.
```

Sample Run 4.3.2

```
Select the genre: (1) Animation (2) Adventure
2
Select the director: (1) Steven Spielberg (2) Jon Favreau (3) Robert Zemeckis
1
Select the movie: (1) E.T. the Extra-Terrestrial (2) The BFG
1
You have reserved the movie: E.T. the Extra-Terrestrial
```

Sample Run 4.3.3

```
Select the genre: (1) Animation (2) Adventure
4
Please enter a valid input
```

Sample Run 4.3.4

```
Select the genre: (1) Animation (2) Adventure
2
Select the director: (1) Steven Spielberg (2) Jon Favreau (3) Robert Zemeckis
2
```

```
Select the movie: (1) The Jungle Book (2016) (2) Elf
2
You have reserved the movie: Elf
```

4.4 Area of a room

You work in a construction company and have been tasked to develop a reusable C++ function that will allow engineers to calculate the area of the room they wish to renovate. For simplicity, we'll imagine every room as a rectangle.

The engineers will be asked to input the room's length and width in feet. Then, the dimensions will be passed into the function `calculateRoomArea()`, which will compute and return the area of the room.

Complete and submit the `calculateRoomArea()` function and `main()` to coderunner. You do not need to include the header.

```
double calculateRoomArea(double length, double width){

}
```

Function: `calculateRoomArea(double, double)`	`double calculateRoomArea(double length, double width)`
Purpose:	To calculate the area of the room.
Parameters:	**double** `length` - the length of the room. **double** `width` - the width of the room.
Return Value:	If successful, it returns the area of the room.
Error Handling:	- If `length` or `width` is non-positive, -1 is returned. Then, in `main()`, the program should display "Length or width is invalid. Area cannot be calculated."

Table 4.1: Function Details for calculateRoomArea

Sample Run 4.4.1

```
Enter the length of the room in ft:
4.5
Enter the width of the room in ft:
2
The area is: 9 sq ft.
```

Sample Run 4.4.2

```
Enter the length of the room in ft:
30
Enter the width of the room in ft:
40
The area is: 1200 sq ft.
```

Sample Run 4.4.3

```
Enter the length of the room in ft:
2
```

```
Enter the width of the room in ft:
0
Length or width is invalid. Area cannot be calculated.
```

Sample Run 4.4.4

```
Enter the length of the room in ft:
-10
Enter the width of the room in ft:
2
Length or width is invalid. Area cannot be calculated.
```

4.5 Estimate Sowing Time

Write a C++ program with a function to estimate the time it takes to sow seeds on farmland. The farming company provides 4 different kinds of sowing machines. Each machine can sow seeds at a different speed per square foot, as shown below.

Sowing Machine	Time taken per square foot
W	8 sq ft per 12 minutes
X	3 sq ft per 10 minutes
Y	2 sq ft per 7 minutes
Z	7 sq ft per 8 minutes

Your program should ask the user the area of the farm and the type of sowing machine the user intends to use in `main()`. Once the information is obtained, call the `calculateSowingTime()` function and pass in the appropriate values.

For submission, please paste in the whole program into coderunner. Please make sure `calculateSowingTime()` function exists and takes in the two parameters in the same order. Below is the function specification for `calculateSowingTime()`:

Function: `calculateSowingTime(double, char)`	`double calculateSowingTime(double area, char machine_type)`
Purpose:	To calculate the time taken to plant seeds all over the farmland.
Parameters:	**double** `area` - the area of the farmland. **char** `machine_type` - the type of machine selected.
Return Value:	If successful, it returns the time taken to plant seeds using a particular machine across the entire farmland.
Error Handling:	- If `area` is non-positive, 0 is returned. - If `machine_type` is invalid, 0 is returned.

Table 4.2: Function Details for calculateSowingTime

Sample Run 4.5.1

```
Enter area of the farmland in sq ft:
1200
Enter the type of sowing machine to be used:
W
The time taken is: 1800 minutes.
```

Sample Run 4.5.2

```
Enter area of the farmland in sq ft:
3244
Enter the type of sowing machine to be used:
X
The time taken is: 10813.33 minutes.
```

Sample Run 4.5.3

```
Enter area of the farmland in sq ft:
5000
Enter the type of sowing machine to be used:
A
Area or machine type is invalid. Time cannot be calculated.
```

Sample Run 4.5.4

```
Enter area of the farmland in sq ft:
1000.45
Enter the type of sowing machine to be used:
Z
The time taken is: 1143.37 minutes.
```

Appendix A: Software Installation Guide

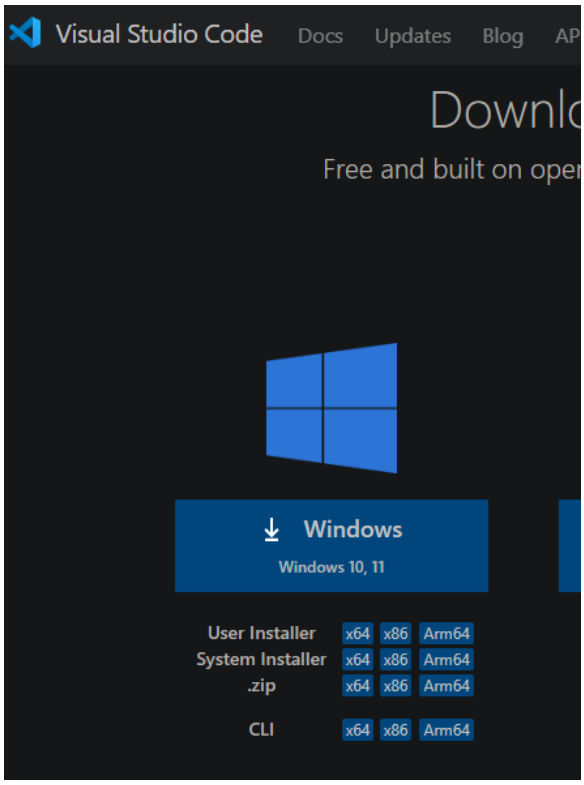
1 VS Code Set Up

1.1 Windows Installation

Important: Before proceeding with this document, make sure that you have run Windows Update within your Windows 10 or 11 environment. You must have the latest updates installed.

Step 1: Install VS Code

- Go to the VS code download page, and download for Windows.



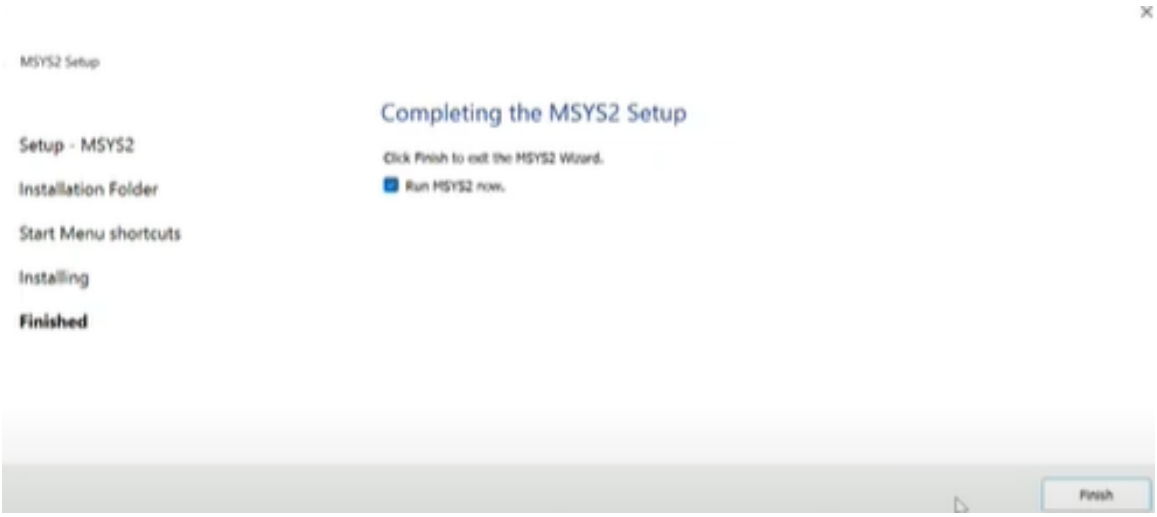
- Run the installer and accept all of the default settings.
- Click on Install and wait for Visual Studio Code to finish installing, then close the installer.

Step 2: Installing MinGW

This section is based on this guide from Microsoft: https://code.visualstudio.com/docs/cpp/config-mingw

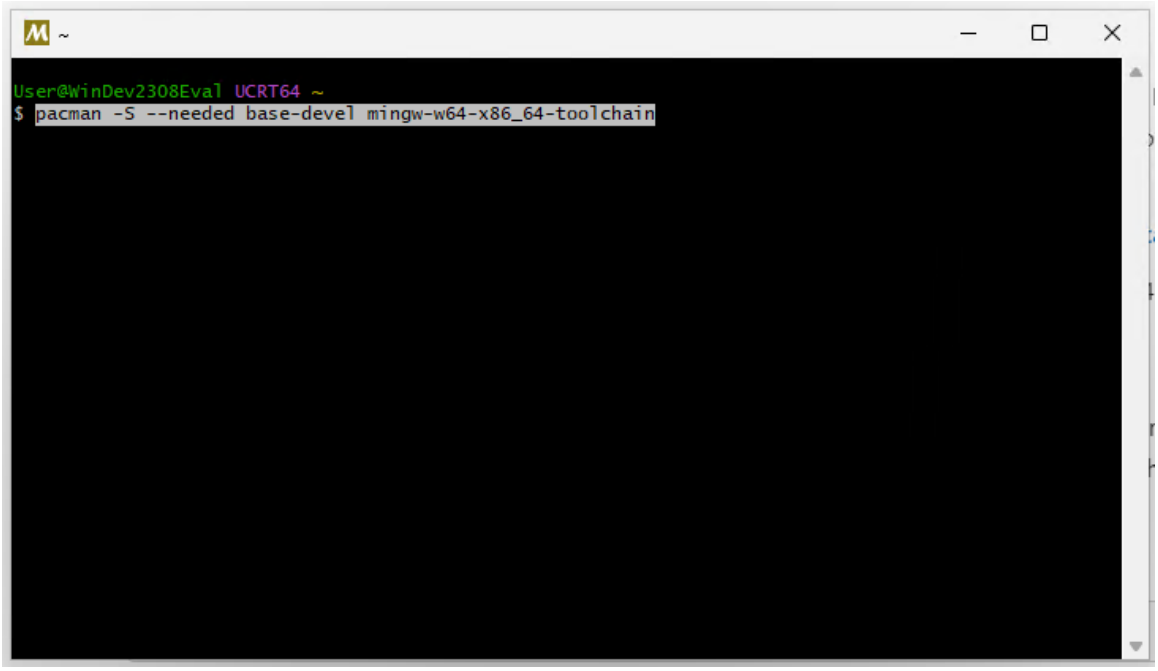
MinGW is a Windows C/C++ compiler tool set that will allow us to compile our C/C++ code into a .exe file.

- First Install MinGW from this link
- Open the installer and choose the Defaults for all settings.
- At the end of the installation run msys2 and then run the following command:



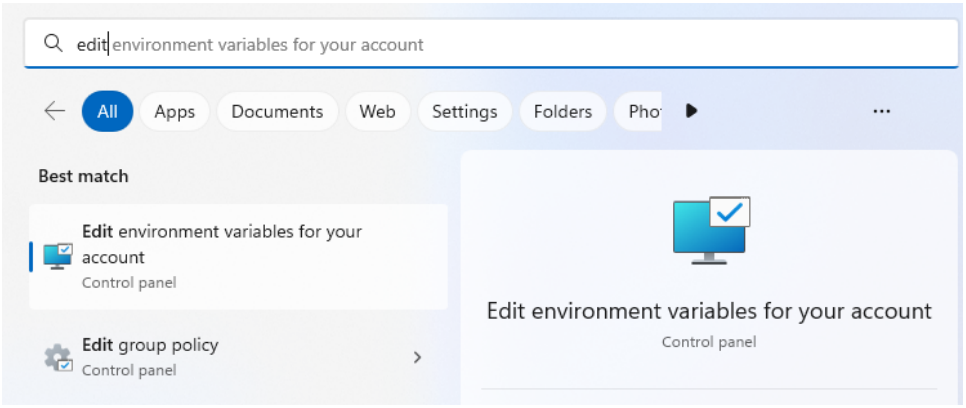
(Shift + Insert is the paste shortcut in MSYS2' s terminal)

```
pacman -S --needed base-devel mingw-w64-x86_64-toolchain
```



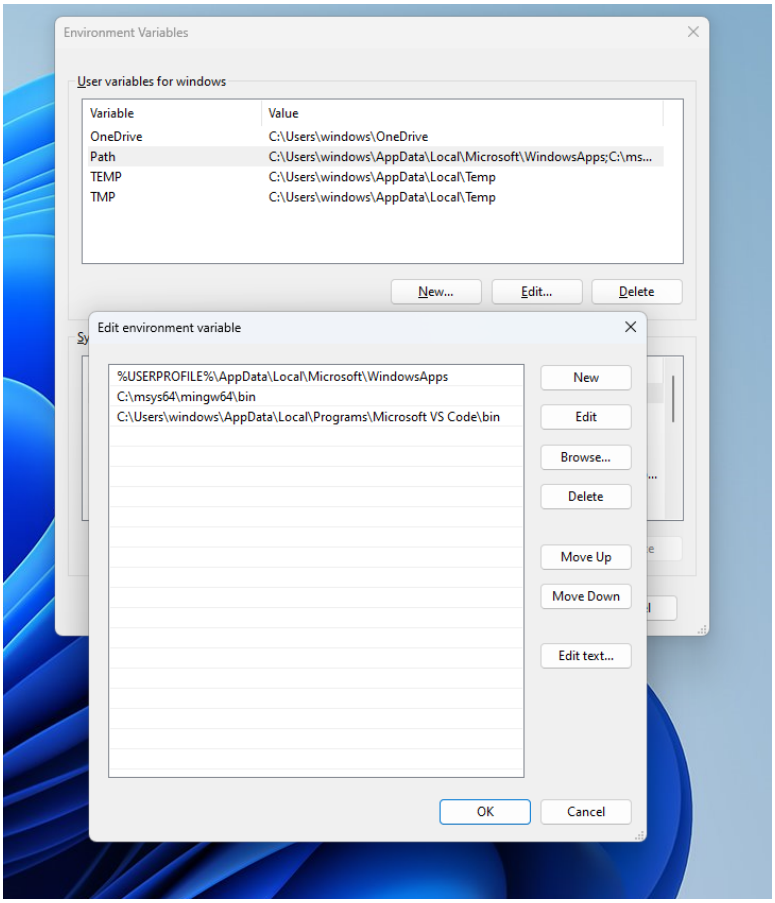
- Press enter when prompted to install all of the default packages, then press Y to confirm the install. This will take 1 to 5 minutes to finish. Once the install completes you can close msys2.

- Now we need to add msys2 to window' s PATH variable. Press the Windows key and begin typing "Edit environment variables for your account" until you see this option.



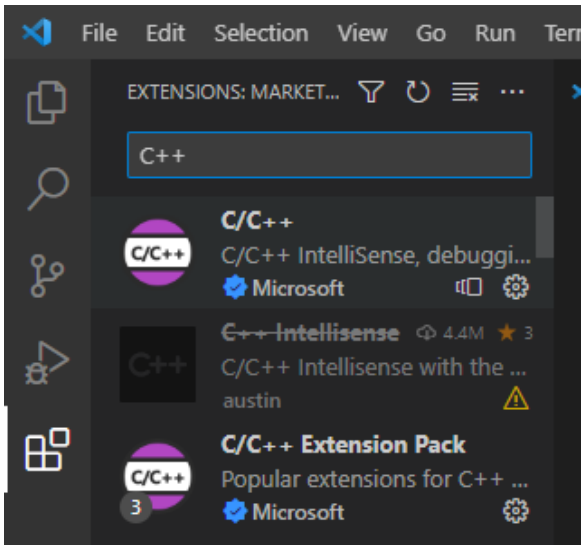
Now Select the "PATH" variable and click edit, in the window that opens click "New" and enter the following path for the default installation location of Msys2.

```
C:\msys64\mingw64\bin
```

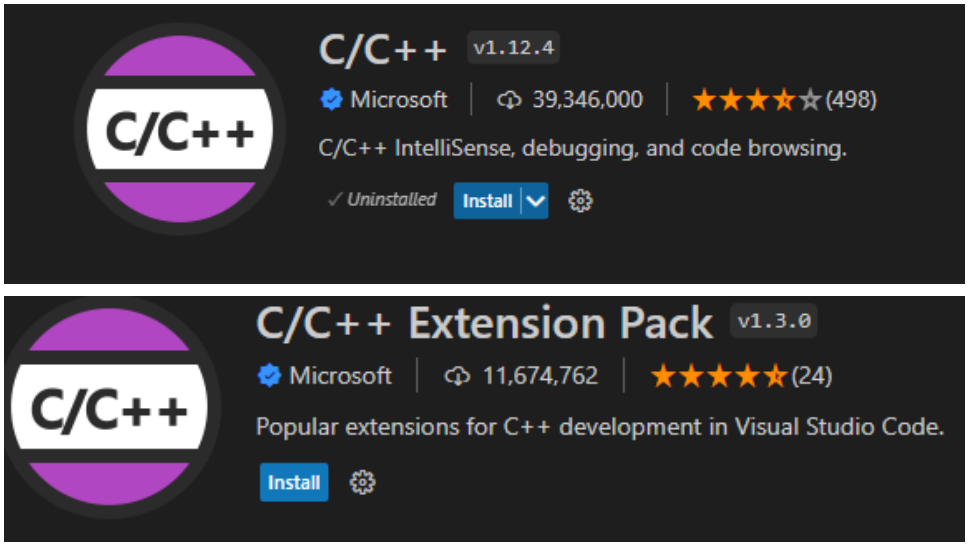


Step 3: Adding VS code extensions

- After you Reboot open VScode and select the extensions tab. (5th from the top), and search for "C++". We need to install the "C/C++" and "C/C++ Extension Pack" both from Microsoft.



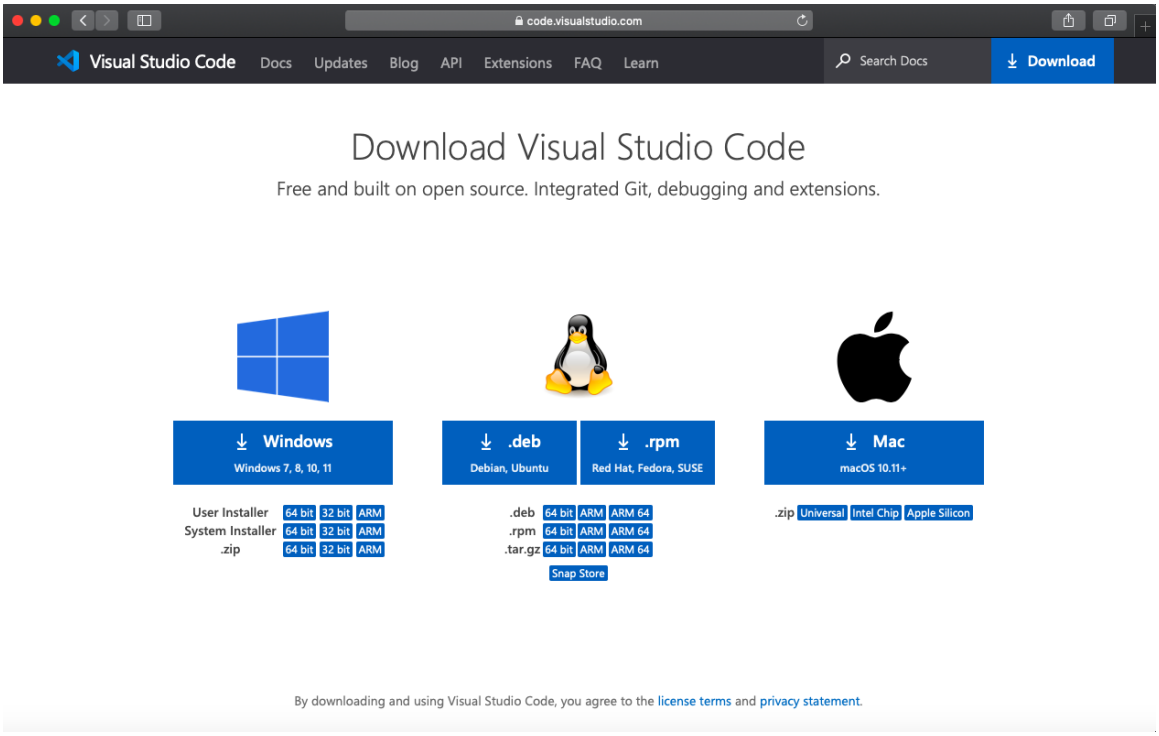
- Select the extension then click on install, these will provide Syntax Highlighting and other useful tools when working in C++



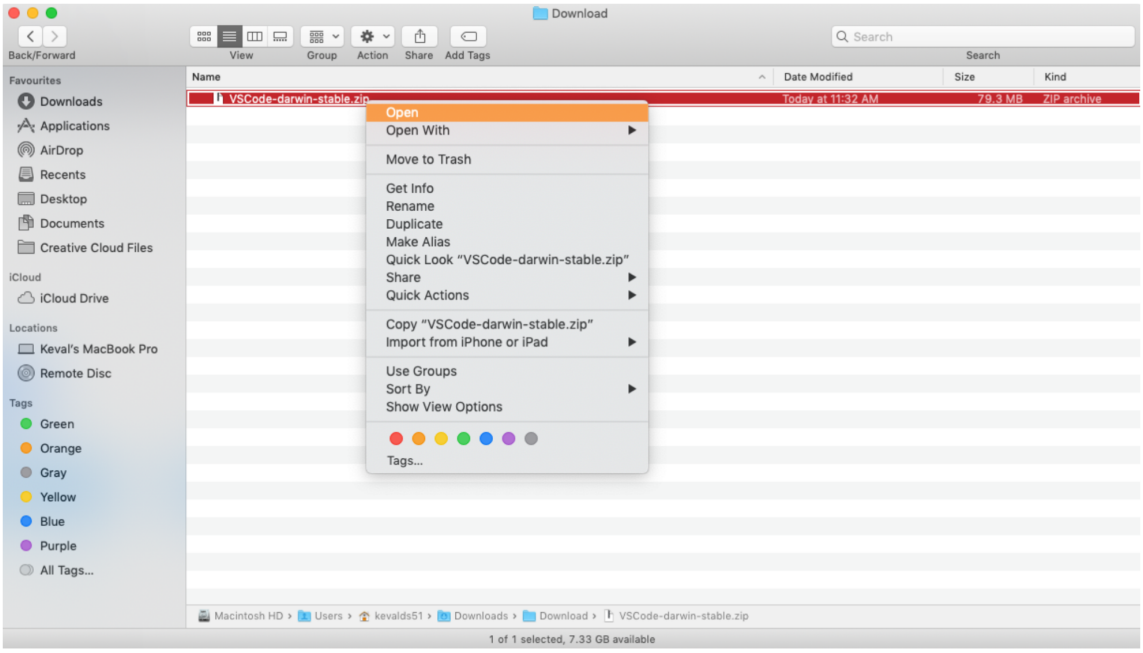
1.2 Mac Installation

Step 1: Installing VS Code

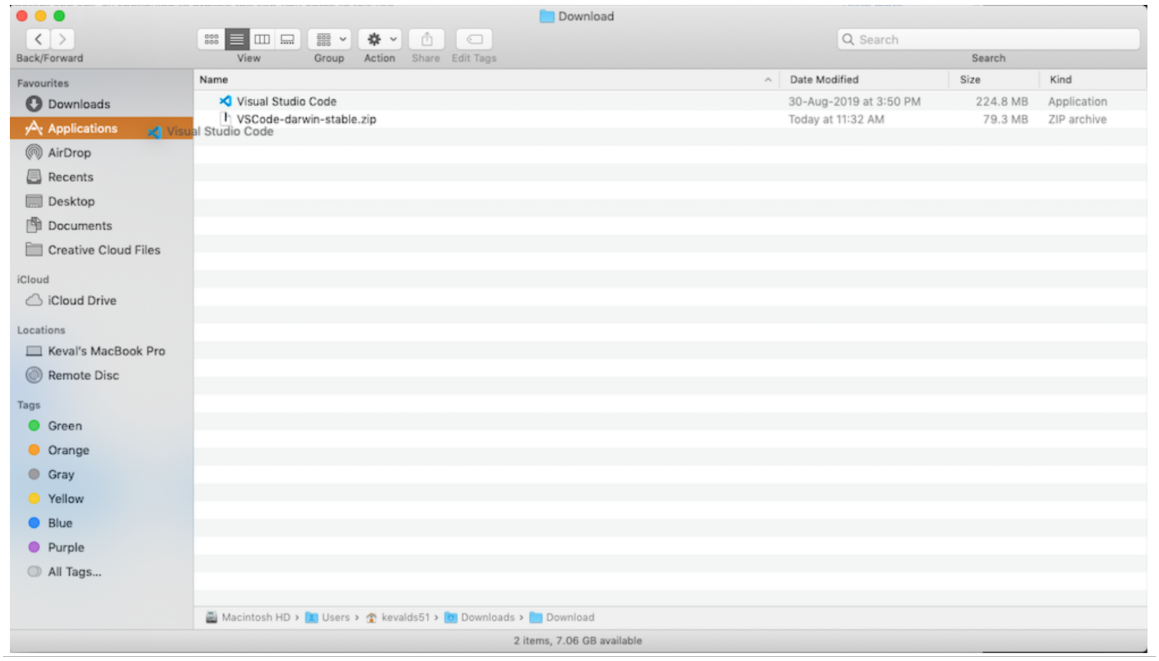
- Go to https://code.visualstudio.com/Download , and download for Mac.



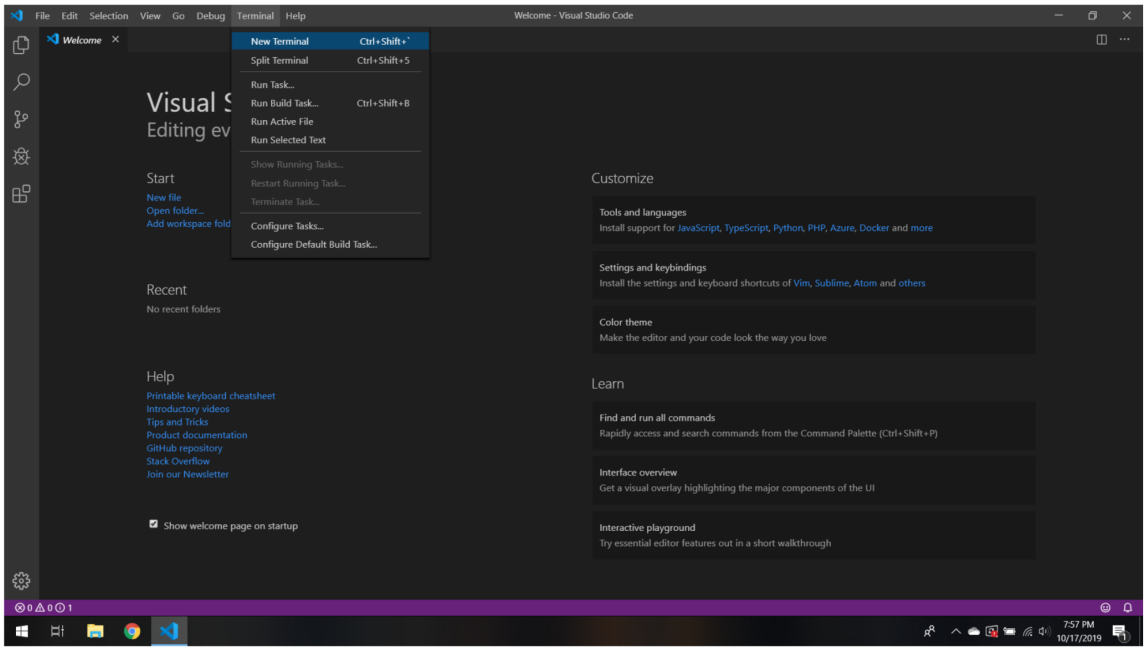
- After the download has finished, unzip the folder by double-clicking on it.



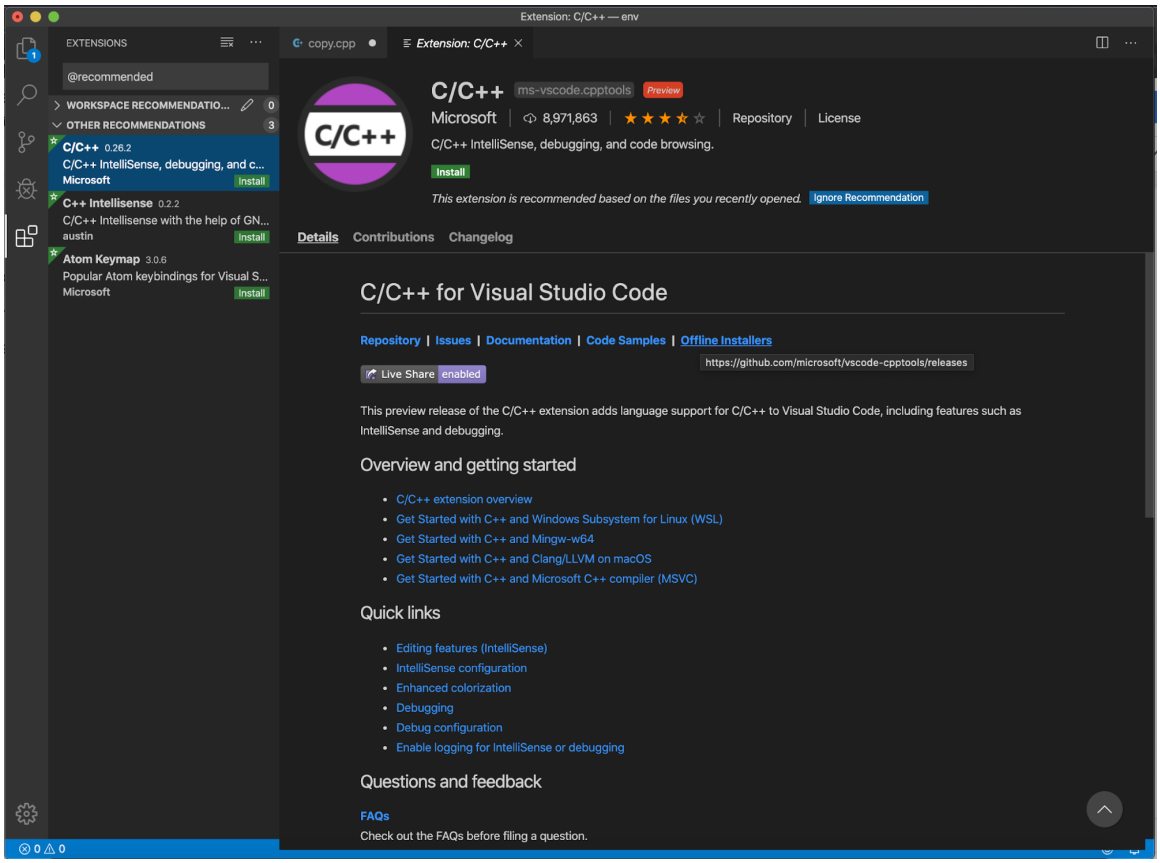
- Now you can see the "Visual Studio Code" application. Drag and drop this icon to the "Applications" folder of your computer.



- Double click on the " Visual Studio Code" icon to launch the application. (You might need to right click and select "open" if you cannot launch the program). Next, select the " New Terminal" option to open the terminal window.

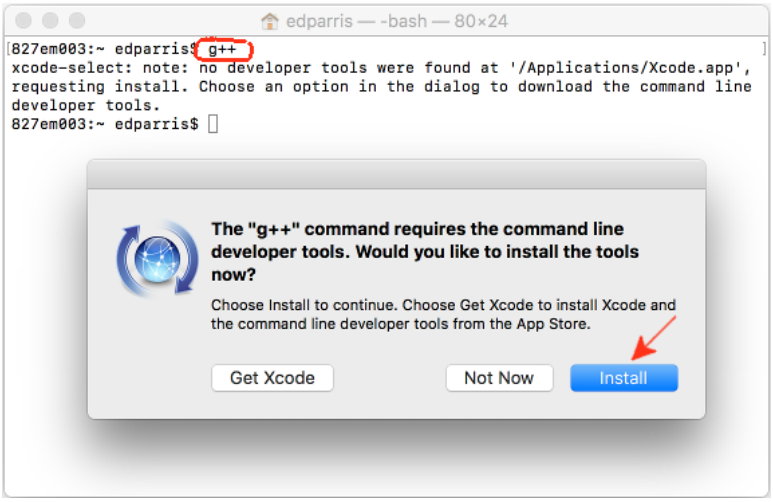


- Install C/C++ extension. In the toolbar on the left hand side of the screen click on the bottom icon for Extensions. Search for C/C++ and click install.



Step 2: Install g++

- Open a Terminal window.(One way is to press Command+Space, type Terminal in the search field, and press the Return key.)
- In the Terminal window type `g++` and press the Return key. We will see an alert box like this:

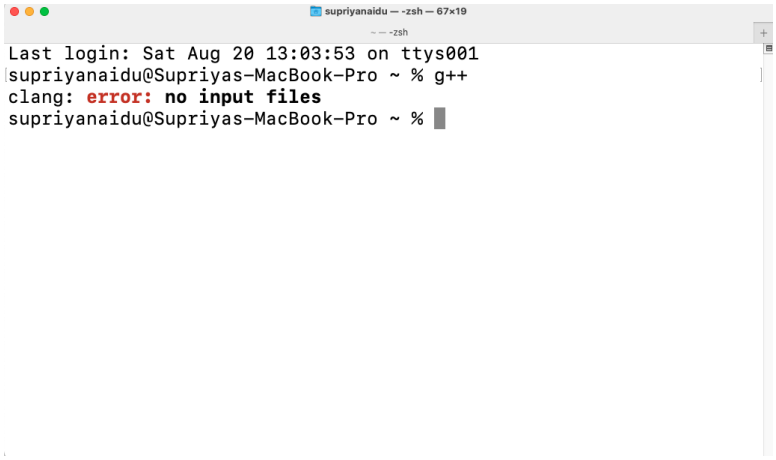


- Choose Install to get only the command line tools unless you want to learn Xcode.

  Xcode can be installed later from the App Store.

  After installation, type g++ in the Terminal, press the Return key, and verify the terminal prints the message, "no input files".

```
$ g++
clang: error: no input files
```



Appendix B: Formatting Guide

1 Naming Conventions

Give variables descriptive names, such as `first_name` or `homework_score`. Avoid one-letter names like `a` or `x`, except for loop counter variables such as `i`.

Example 1.0.1. Poor variable names would be:

```
int thing = 16;
double a = 2.2;
string x = "Michael";
```

Better variable names would be:

```
int cups_per_gallon = 16;
double lbs_per_kilo = 2.2;
string first_name = "Michael";
```

2 Whitespace

"Whitespace" refers to all of the empty space you can use to organize your code including new lines, indentation, and extra spacing. Good whitespace usage helps reduce the strain on the reader's eyes. The compiler ignores whitespace, which allows you can place things anywhere and format them however you want. To maximize readability, there are a few guidelines for whitespace usage.

Indentation: Increase your indentation by one increment of each brace "{" and decrease it once on each closing brace "}". Use Tab to increase indent and Shift+Tab to decrease indentation. You can also increase indent for multiple lines by highlighting them and pressing CTRL+] on Windows and CMD+] on Macs, and decrease the indent for multiple lines by highlighting them and pressing CTRL+[on Windows and CMD+[on Macs.

New Lines: You should use one blank line to separate blocks of code. You can separate sections of code based on the tasks they are meant to complete. You should also put any brackets on their own lines to help visually break up your code.

Example 2.0.1. Here is a poor example of whitespace usage:

```
int main(){int number = 0;
if(number < 5){
cout << "Less than 5" << endl;
}else{
cout << "Greater than 5" << endl;
}
number++;
}
```

Here is a better example of whitespace usage:

```
int main()
{
    int number = 0;
    if(number < 5)
    {
        cout << "Less than 5" << endl;
    }
    else
    {
        cout << "Greater than 5" << endl;
    }
    number++;
}
```

3 Commenting

Your code should be well-commented. Use comments to explain what you are doing, especially if you have a complex code section. These comments are intended to help other developers understand how your code works. Comments can also help you remember your own code if you have to go back and read it a week or a month after writing it. Single-line comments should begin with two forward slashes (//). Multi-line comments begin with one forward slash and an asterisk (/* ... comments here ... */).

Example 3.0.1. Here is a single line comment:

```
// CSCI 1300 Fall 2024
```

Here is a multi-line comment:

```
/*
Algorithm:
Input: two numbers
Output: sum of input numbers

1. Ask the user to enter a number
Save in variable number_1
2. Ask the user to enter a number
Save in variable number_2
3. Compute sum
sum = number_1 + number_2
4. Display sum to user
*/
```

Appendix C: Syntax Guide

This syntax guide was designed with heavy inspiration from the Coding with Harry C++ syntax guide.

1 Basics

The basic start for a C++ program is:

```
#include <iostream>
using namespace std;

int main() {
    return 0;
}
```

Declaring variables:

```
bool my_bool;
bool my_bool_initialized = false;
char my_char;
char my_char_initialized = 'A';
int my_int;
int my_int_initialized = 1;
double my_double;
double my_double_initialized = 3.5;
float my_float;
float my_float_initialized = -2.1;
string my_string;
string my_string_initialized = "Hello!";
```

Terminal output:

```
cout << [statement to print];
cout << variable_contents;
cout << "Hello!";
```

Terminal input:

```
cin >> variable_input;
getline(cin, string_input);
```

Comments:

```
// It's a single line comment

/* It's a multi-linecomment*/
```

Compiling:

```
g++ -Wall -Werror -Wpedantic -std=c++17 file1.cpp file2.cpp ...
```

2 Decisions

Decisions are decided based off of boolean values. In this guide, any time `< condition >` appears, it should be replaced fully (including the angle brackets) with something that expresses to a boolean value.

If statements:

```
if ( < condition > ){
    //code
}
```

If-Else statements:

```
if ( < condition > ){
    //code
}
else {
    //more code
}
```

If-Else-If statements:

```
if ( < condition > ){
    //code
}
else if ( < condition2 > ){
    //more code
}
```

Integer switch statements. The case values can vary:

```
switch(integer_variable){
    case 1:
        //commands
        break;
    case 2:
        //commands
        break;
    //any further cases
    default:
        //commands
}
```

Character switch statements. The case values can vary:

```
switch(character_variable){
    case 'A':
        //commands
        break;
    case 'B':
        //commands
        break;
    //any further cases
    default:
        //commands
}
```

3 Functions

Functions are small packages of code that can be called multiple times. The general format of a function requires a return type (which is a data type), a function name, and a list of parameter inputs, like so:

```
<return type> FunctionName(<parameter 1>, <parameter 2>, ...);
```

There can only be one return type and it must be a data type. It can be any data type you have seen so far, or it can also be "void" which means the function returns nothing. There can be as many parameters as you like, and they must be written as <parameter i> = <data type of parameter i> <name of parameter i>.

Function prototypes are just the function declaration followed by a semicolon.Here are a few function prototype examples:

```
int AddNumbers(int numOne, int numTwo);
void PrintMenu(string menu);
double calcCircum(double radius);
```

When implementing the function, it is instead written with brackets:

```
<return type> FunctionName(<parameter list>){
    /* code for function */
    return <something of return type>;
}
```

When calling the function, you provide the list of arguments:

```
int sum = AddNumbers(firstVal, secondVal);
sum = (4, 5);
```

4 Strings

Declaring strings:

```
string my_string = "Hello World";
```

Getting the length of strings:

```
int string_length = myString.length();
```

Appending two strings:

```
string first_name = "Harry ";
string last_name = "Potter";
string full_name = first_name.append(last_name);
```

Accessing or changing characters:

```
char first_character = my_string[0];
my_string[1] = 'a'; //my_string now stores "Hallo World"
```

5 Loops

While loops repeat code as long as a condition is true:

```
while (< condition >)
{
/* code block to be executed */
}
```

Do-While loops will execute the block of code, and then execute it again as long as a condition is true:

```
do
{
/* code block to be executed */
} while (< condition >);
```

For loops will execute a block of code a given number of times. This is done by creating a variable, modifying that variable and go until a particular condition is false.

```
for (<create and initialize variable>; <condition>; <modify variable>){
    /* code block to be executed */
}
```

For example, to count from 0 to 9:

```
for (<create and initialize variable>; <condition>; <modify variable>){
    /* code block to be executed */
}
```

The keyword break terminates a loop:

```
    break;
```

The keyword continue skips the rest of the current iteration of the loop:

```
    continue;
```

6 Objects

To define a class:

```
class Class_Name {
    public: // Access specifier
        // member functions

    private:
        //data members
};
```

To create an object of that class:

```
Class_Name object_name;
```

Constructors describe how a new object's data members should be initialized. It does not have a return type. It must match the class name.

```
class Class_Name {
    public: // Access specifier
        Class_Name(); //default constructor
        Class_Name(<parameter list>); //parameterized constructor

    private:
        //data members
};
```

To call a member function on a particular object:

```
    object_name.functionName();
```

7 Libraries

To use a library you need to know the library name and include it.

```
#include<LibraryName>
#include "libraryFileYouWrote.h"
```

7.1 Math

The library is called <cmath>.

The square root function:

```
double square_root = sqrt(169);
```

The power function (returns the value of x raised to the power y):

```
double power = pow(x, y);
```

7.2 File I/O

The library is called <fstream>.

Creating and writing to a text file:

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
// Create and open a text file
ofstream my_file("filename.txt");

// Write to the file
my_file << "File Handling in C++";

// Close the file
my_file.close();
}
```

Opening a file and reading one line:

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
// Open a file to read from
ifstream my_file("filename.txt");

string file_line;

getline(my_file, file_line); //stores the first line in the file in file_line

// Close the file
my_file.close();
}
```