C++ Fall 2019 Notes & Documentation:

Basics:

* #include <iostream> 🡪 allows you to read & write standard the standard output streams
* using namespace std; 🡪 allows us to use objects and variables from the standard library
* int main() {} 🡪 where everything is executed
* #include “pch.h” 🡪 ??
* Return 0; 🡪 ??

Variables/Data Types:

* Int
* Double
  + Float vs double?
  + Float – allows 7 digits after the decimal
  + Double – allows 15 digits after the decimal
* Char
* String
  + #include <string>
  + string greeting = “Hello”;
  + cout greeting[0]; //outputs H
* Bool
* Const 🡪 read only

Input/Output:

* cout 🡪 display to the screen – uses insertion operator <<
* cin 🡪 get user input – uses extraction operator >>
* Example:
  + string firstName;
  + cout << “What is your name?”;
  + cin >> firstName;
  + cout << “Your name is:” << firstName;
* \*cin considers a space/tab as a terminating character
  + string fullName;
  + cout << “What is your full name?”;
  + getline(cin, fullName);
  + cout << “Your full name is: ” << fullName;

Binary Operators: operate on 2 operands 🡪 \* / % + -

Unary Operators: operate on only 1 operand 🡪 - 🡪 flips sign of current value?

**Chapter 3**:

The CIN Statement:

* Cin automatically converts the data read from the keyboard to the data type of the variable used to store it
* Requires <iostream> header file, enter key is pressed after data is entered
* Can be used to gather multiple values (separated by spaces)
* EX:

Arithmetic operators and their precedence:

* When 2 operators share an operand, the operator with the highest precedence works first.
* Highest 🡪 \* / %
* Lowest 🡪 + -

Set Precision Statement:

* Sets the number of significant digits
* Specify where floating point will go, counting from right
* Stream manipulator

Fixed Manipulator:

* stream manipulator
* same as setprecision, with force decimal notation
* must be used with setprecision???

ShowPoint:

* adds zeros to equate to the number of places specified

The **>>, <<** operators:

* >> 🡪 stream extraction operator: gets characters from the stream object on its left, and stores them in the variable on its right
* In a statement that uses COUT – the << operator always points towards cout. This indicates that data is flowing from a variable or a literal to the cout object.
* In a statement that uses CIN – the >> operator always points towards the variable that is receiving the value. This indicates that the variable

The header file <iostream>:

* Ok

Associativity:

* the order in which an operator works with its operands
* An operator’s associativity is either: left 🡪 right OR right 🡪 left
* Unary negation - 🡪 right to left
* \* / % 🡪 left to right
* + - 🡪 left to right

When C++ is working with an operator, it strives to convert operands to the same type. This is known as:

* Automatic type conversion **🡪 Type Coercion**
* Promoted: when a value is converted to a higher data type
* Demoted: when a value is converted to a lower data type

When a variable is assigned a number that is too large for its data type, what happens?

* The variable overflows
* Attempting to store a number too large for the given variable type

setw manipulator:

* specifies the field width for the value immediately following it
* the field width is the minimum number of character positions, or spaces on the screen to print the value in it.
* Cout << “(” << setw(5) << value << “)”; // value = 23
* ( 23) //output
* Requires <iomanip>

The function **pow(x, y)**, requires which header file?

* <cmath>
* Unlike many programming languages, C++ does NOT contain an exponent operator
* This requires a library function

Rand() Function:

* Needs header file – <cstdlib>

Hand Tracing:

I will give you snippets of code and you should be able to identify the answer. Here is an example.

Which of the following will allow the user to input the values **15** and **20** and have them stored in variables named **base** and **height**, respectively?

|  |  |
| --- | --- |
| a. | **cin << base << height;** |
| b. | **cin base, height;** |
| c. | **cin >> base >> height;** |
| d. | **cin base >> cin height;** |
| e. | None of these |

**Chapter 4: Making Decisions**:

Relational Operators:

* ==, >=, <=, !=
* Used to compare numeric data
* All relational operators are binary – need to operands to operate on
* All relational operators have left to right associativity
* Each relational operator determines whether a specific relationship exists between 2 values
* Relational Expressions are also known as Boolean expressions – value can only be true or false

(ex: x > y 🡪 used to determine if x is less than y)

Logical Operators:

* Connects two or more relational expressions into one, or reverse the logic of an expression
* Typically used with Boolean values
* && (logical AND) – takes 2 expressions as operands and creates an expression that is true only when both subexpressions are true.
* || (logical OR) – takes 2 expressions as operands and creates an expression that is true when either of the subexpressions are true. \*performs short-circuit evaluation
* ! (logical NOT) – takes 1 operand and reverses its truth or falsehood. Must first evaluate the expression, then apply the “!” – flop the value.

\*C++ does not allow you to check numeric ranges with expressions such as: 5 < x < 20. Instead you must use a logical operator to connect the two relational expressions: x >= 5 && x <= 20.

sequence structure: statements are executed in a sequence, without branching off in another direction

decision structure: a specific action is taken only when a specific condition exists

If Statement:

* if the value of the expression inside of the parenthesis is true, the very next statement/group of statements inside the containing brackets is/are executed. Otherwise, the statement(s) is skipped.
* Braces needed if more that one statement should be executed upon true condition/expression

Comparing floating point numbers?

If/Else Statement:

* Will execute one group of statements if the expression is true, and a different group of statements if the expression is false.

Nested If Statement:

* Used to test multiple conditions, separately.
* Could also be done by using if/else if statements
* Could also be done by combining conditions & using logical operators in same expression.

If/Else If Statement:

* Tests a series of conditions, like nested if statements.
* Often a simpler way to test multiple conditions than using nested ifs.
* Each expression is tested sequentially, until a true evaluation is found. All other expressions/statements are skipped and the if statement is over.
* If none of the expressions are true, then the last else clause is executed.

Flag:

* A Boolean or integer variable that signals when a condition exists
* If the flag variable is set to true – the condition exists
* If the flag variable is set to false – the condition does not exist
* Must set the flag variable based on a certain condition, then you can later test that variable and use it to determine a conditionally executed statement.

(ex: if (flagVar) { “you have reached your goal sale!” }

Integer Flags: 0 is false, any non-zero value is true. Works same as Boolean

Input Validation:

* The process of inspecting data given to a program by the user & determining if it is valid.
* A good program should give clear instruction as to the kind of input that is acceptable, while also assuming that the user has not followed those instructions.

Comparing Characters & Strings:

* Relational operators can also be used to compare character & string objects.

The Conditional Operator:

* Shorthand for if/else statements
* expression ? expression : expression
  + 1st: expression to be tested
  + 2nd: executes if true
  + 3rd: executes if false
* Takes 3 operands – TERNARY

Switch Statement:

* Lets the value of a variable or an expression determine where the program will branch
* Similar to if/else statement, but it tests the value of an integer expression and then uses that value to determine to which set of statements to branch
* An optional default section comes after all the case statements. The program branches to this section if none of the case expressions match the switch expression.
* Selects the execution of the statement often based on a keyboard command VS.
* Allows only the testing of a single expressing against a list of discrete values (less flexible that if/else)

Blocks & Variable Scope:

* The scope of a variable is limited to the block in which it’s defined.

Counter:

Accumulator:

* Keeps track of the sum of numbers that accumulates with each iteration of the loop

**Chapter 5 – Loops & Files**:

**++** and **–** operators:

* Increment or Decrement operators
* Prefix & Postfix ?test? (order)
* Post: evaluates the variable, & then adds or subtracts
* Pre: adds or subtracts, & then evaluates

Counter:

* Variable that is regularly incremented or decremented each time a loop iterates

While loop:

* Executes statements while the condition is true
* Pretest loop – tests its expressions before each iteration
* Good for input validation – repeats while predicted bad data is entered, until acceptable data is entered.
* Good for reading lists of data terminated by a sentinel value.
* EX: while (x < 10 ) { do something; x++; }

Do-while loop:

* Always executes action at least once
* Posttest loop – executes its expression after each iteration
* If condition evaluated is true, action continues while it remains true
* MUST BE TERMINATED WITH A SEMICOLON
* Good for when you always want the loop to iterate at least once.
* Good choice for a repeating menu
* EX: do { statements } while ( condition );

Conditional Loops: executes as along as a particular condition exists – While & Do While loops

Count-Controlled Loops: a loop that repeats a specific number of times – FOR LOOP

User Controlled Loop:

* Lets the user decide the number of iterations; ex: asks the user to enter Y or N
* Programmer has no way of knowing the number of times the loop would iterate

For Loop:

* Ideal for performing a known number of iterations
* Pretest loop
* Count-controlled loop
* Steps: initialize counter variable to a starting value, test against max/min value, ++/--
* Can use multiple statements in the initialization and update statements
* FORMAT: for ( initialization; test; update ) { statement }
* EX: for ( int x; x < 10; x++ ) { statements }
* It is possible to execute more than one statement in the initialization & update expression; separate with commas

Break statement:

* Terminates the loop early & program jumps to next line after the loop
* Can be used to give the user the option to terminate the loop by entering in a value.

Continue statement:

* Causes a loop to stop its current iteration and begin at the next iteration.
* All other statements in the body of the loop after the continue will be ignored.
* While loop – program jumps to test expression at top of loop
* Do While loop – program jumps to test expressions at bottom of loop
* For loop – program jumps to update expression, and then test expression.

Infinite loop:

* When the conditional statement/Counter is absent

Sentinel:

* A special value that marks the end of a list of values.
* Often used by asking a user to enter a sentinel at the end of a list of values.
* Cannot be mistaken for one of the values & signals that there are no more values to be entered.
* When a user enters a sentinel, the loop terminates.
* EX: prompt user to enter list of points their team has earned this season. Enter -1 when finished.

GIGO:

* Garbage in garbage out
* Input Validation – checks for integrity/reasonableness
  + Display error & provide more specific instruction

**Chapter 6 – Functions**:

Modular Programming:

* Uses functions/modules to break programs down into smaller manageable pieces

Modules can be compiled and tested separately, and can be imported into other projects as well.

* Improves maintainability & simplifies process of writing programs
* Can be made available for other modules by interfaces

Function name:

Parameters / arguments

* Arguments: values sent into a function; listed in the parenthesis of a function call
* Parameters: variables that receive arguments; defined in the parenthesis of a function definition
* When passing a variable as an argument, simply rite the variable name; do NOT include the variable type
* If you pass an argument that is not the same as the parameter’s type, the argument will be promoted or demoted automatically
* Any argument listed inside the parenthesis of a function call is copied into the function’s parameter value - in essence parameter variables are initialized to the value of their corresponding arguments

Function call:

* Statement that causes a function to execute

Return statement:

* Causes a function to end immediately

Value returning function:

* Function that sends a value back to the part of the program that called it
* Contains a return type in the function header – the type of the variable that is returned to the program
* It is possible to return multiple values, but they must be packaged in such a way that they are treated as 1 value

Function definition:

* Contains the statements that make up the function
* Contains the function header?

Function header:

* Declares the function return type, name, & parameter list

Function prototype: (function declaration)

* Eliminates the need to place a function definition before the call to the function – BUT \*Must place prototype ahead of the call to the function
* Same as the function header, but it has a semicolon at the end – tells compiler the return type and what parameters if any are needed
* Also known as a function declaration
* You must place either the function definition or the function prototype ahead of all calls to the function
* So that main() can be the first function in the program – prototypes, main(), all other functions.

Calling/Passing data by Value:

* In call by value, only a copy of the value being passed to the function. It is locally stored by the function parameter in stack memory location. Changes to the value are only for the current function. It will not change the variable value inside of the caller method – int main() for example.
* When only a copy of an argument is passed to a function, it is said to be “passed by value”. This is because the function receives a copy of the argument’s value and does not have access to the original argument.
* Original value is not modified in call by value

Reference Variable:

* When used as a function parameter, it allows access to the original argument
* Is an alias for another variable
* Must use “&”
* The “&” must appear in both the prototype and the header of any function that uses a reference variable as a parameter. It does not appear in the function call.
* EX: void doubleNum(int &refVar) { refVar \*= 2; }
* The parameter value points to the value variable in the function main(). When a program works with a reference variable, it is actually working with he variable it references, or to which it points.

Calling/Passing data by Reference:

* In call by reference, the original value is modified because we pass the reference (address)
* In call by reference, the address of the value is being passed to the function. Actual & Formal arguments share the same address space. Changes to the value inside of the function are reflected outside of the function.
* When a reference parameter is used, it’s said to be PASSED BY REFERENCE
* Original value is modified in call by reference

Using functions in menu driven programs:

* Functions are ideal for use in menu-driven programs. When the user selects an item from a menu, the program can call the appropriate function.

Scope:

Local variables: defined inside a function and is not accessible outside of that function

Global variables:

* Defined outside of all functions and is accessible to all functions in its scope
* Can be accessed by all functions that are defined after the global variable is defined, without being passed as arguments
* Should be used in moderation: make debugging difficult,
* Because of this, you should not use global variables for the conventional purposes of storing, manipulating, & retrieving data. In most cases, you should declare variables locally and pass them as arguments to the functions that need to access them

Global Constant:

* A named constant that is available to every function in a program
* Since the value cannot be changed, don’t have to worry about the potential hazards that are associated with global variables
* Typically used to represent unchanging values that are needed throughout the program

Local & Global Variables with the same name:

* Cannot have 2 local variables with the same name
* Can have a local and a global variable with the same name

Static Local variables:

* If a function is called more than once in a program, the values stored in the function’s local variables do not persist between function calls.
* Local variables are destroyed when the function ends, and recreated when the function starts again
* Sometimes it’s desirable for a program to remember what value is stored in a local variable between function calls
* Static local variables are not destroyed when the function returns, the exist for the lifetime of the program even thought their scope is only the function in which they are defined

Overloaded functions:

* Two or more functions may have the same name as long as their parameter lists are different

Exit() Function:

* Causes a program to terminate, regardless of which function or control mechanism is executing
* Requires <cstdlib>

Stubs & Drivers:



Functions/Parameters/Arguments

Static/local/global variables

Default arguments

Return statement

Exit function/Prototype function

Stub/dummy function

**Chapter 7 – Arrays & Vectors**:

Array:

* Allows you to store and work with multiple values of the same data type
* EX: int days[6];

Array Size Declarator:

* The number inside the brackets of an array definition
* Must be greater than 0; can be a literal(6), or named constant(ARRAY\_SIZE)

Subscript:

* The number inside the brackets of an assignment statement, or any statement that works with the contents of an array.
* Can be stored as variables

Accessing Array Elements:

* Even though an entire array has only one name, the elements may be accessed and used as individual variables – because each element is assigned a number known as a subscript.
* Subscript numbering always starts at 0 – therefore – the subscript of the last element in the array is always one less than the total elements in the array
* The individual elements of an array are assigned unique subscripts, which are used to access the elements in the array.
* EX: days[0] – first elements in the array

Initialization List:

* The series of values inside the braces and separated with commas
* Allows you to initialize an array’s elements when you declare/create the array

Partial Array Initialization:

* Can initialize only part of the list; the uninitialized elements will be set to 0 or empty string
* CANNOT skip elements

Implicit Array Sizing:

* Can define an array without specifying the size, as long as you prove an initialization list
* EX: double scores[] = { 2, 5, 10 };

Reading Writing contents/data to/from a file

No Bounds Checking