Chapter 2

January 20, 2019 7:31 AM

- History
  - **1960s - ALGOL**
    - C is based off that starting point
  - **1967 - BCPL** (Basic Combined Programming Language)
    - Created by **Martin Richards**
  - **1970 - B Language**
    - Created by **Ken Thompson**
  - **1972 - C Language**
    - Created by **Dennis Ritchie**
    - A direct derivative of the B programming language
  - **1989 - ANSI C standardizatio**n
  - **1999 - Major C standard update**
  - **2011 - Minor C standard update**
- Basic Steps
  a. **Understand**
  b. **Plan**
  c. **Code ***
  d. **Compile ***
  e. **Test ***
  f. **Implement**
- Don't forget documentation
- Errors
  - **Syntax error**
    - Prevents the program from compiling because the language is wrong
    - Error messages may help to solve syntax errors
    - Often times syntax errors are misspellings
  - **Logic error**
    - Prevent a program from running properly
    - Logic errors are more subtle than other errors
    - A program can still run even with logic errors
    - Testing is very important
- **Five key components** to the program
  - **Preprocessor directives**
  - **Global declarations**
  - **Local declarations**
    - Goes inside main
  - **Statements**
    - Goes inside main
  - **Other functions** as required
- Comments
  - Describes a function or code block's purpose
  - Describe **why** you are doing something not **how** you are doing something
  - Used to communicate what the program does to other and yourself
  - Can be
    - **Bock comments**, which use /* and */
    - **Line comments**, which use //
  - Don't overuse comments
  - Can appear anywhere a space can, but cannot be nested
- Preprocessor directives
  - **#include <stdio.h>**
  - Tell the computer to do something before compiling
  - Appear at the beginning of the code
  - Begin with a '#'
    - #include includes a header file or library
    - Appendix G - pg. 1071
  - Line the # all the way to the left
  - No space between # and include

- Libraries
  - Additional features (mostly functions), which can be available the program
  - C is a small language - libraries extend it as needed
  - Standard libraries available
    - Appendix F - pg. 1059
  - Programmers can create their own
- The main() function
  - Int main()
    {
        Declaration section
        Statement section
    }
  - A main() is required
  - The program begins and ends with main()
  - It calls other functions
  - Every main() will have at least one action
  - Will **always end with return 0**
- The printf() Function
  - Printf("Hello World!\n");
  - Printf() is a standard library function from <stdio.h>
    - This library is used for most programs
- Printf() Control Code
  - "Escape sequences" used for special things
  - Control codes begin with a '\'
    - List of codes - pg. 48
  - **#include <stdio.h> allows the use of printf()**
- Variables Identifiers
  - Generally have a name referring to a place in memory where data or objects can be stored
  - **Variable parts**
    - **Name**
    - **Type**
    - **Value**
    - **Address**
  - Using a variable name accesses the value stored at that memory location
  - Variables are dynamically allocated to a program in C
  - **Three areas of memory**
    - **Literal Pool**
    - **Code/Instructions**
    - **Variable Pool/Memory**
  - Every time you ask for a variable the Variable Pool grows
- Identifier Naming
  - Syntax rules
    - **First Character must be alphabetic character or underscore**
    - **Must consist only of alphabetic characters, digits, or underscore**
    - **First 63 characters of an Identifier are significant**
    - **Cannot duplicate a keyword**
  - Use meaningful names
  - C is a case-sensitive language
  - Use 'camel' casing for this class
- Data Types
  - Defines a set of values and a set of operations that can be applied on those values
  - Common types
    - **Decimal numbers** - primitive
    - **Non-decimal numbers** - primitive
    - **Characters** - primitive
    - **Strings** - advanced
  - When a variable is created, the computer reserves memory to hold the maximum value for that data type
- Void
  - A void type has no values and no operations
  - This cannot be used for variables
  - Functions the same way you would use a postcard in real life
- Integral Types
  - Integral types are whole numbers

- Three integral types
  - **Boolean** (**bool**)
    - Represents a true or a false value
    - True is 1 false is 0
    - Takes up 1 byte of data (8 bits)
    - **The library <stdbool.h> must be included to use this data type**
    - Added for the C99 standard
  - **Character** (**char**)
    - Are symbols that we look up with a number
    - Takes up 1 byte of data
    - ASCII 48: 0, 65: A, 97: a
  - **Integer** (**int**)
    - Value without a fraction part
    - Four types:
      - **Short int, int, long int, long long int**
      - Short: **2** bytes
      - Int: **4** bytes
      - Long int: **4** bytes
      - Long long int: **8** bytes
    - A signed integer means you can have both positive and negative numbers
- **Technically 0 is false and anything other than 0 is true in the C language**

❖ Floating-Point Types
- Values containing a fractional part
- Three types:
  - **Float**
    - 4 bytes
  - **Double**
    - 8 bytes
    - More precise than a float
  - **Long double**

❖ Declaring Variables
- Syntax lists the type then the name

❖ Assignment Operator
- Data is assigned to a variable after it has been created
- Assignments are done using the **=** operator
  - "gets"

❖ Important Point!
- **What happens if you declare a variable and it doesn't get a value?**
  - **The variable will have "junk" in it**
  - **Every variable should be initialized before being used as an output in a program**
- **P. 45**

❖ Initializer
- When you initialize a variable upon creation
  - int myVariable = 77;

❖ Important items
- In a block, variables and created - then used
- A block is surrounded by { }
- The main block has two sections
  - **Declarative**
    - Variables are defined
  - **Executable**
    - Variables are used
- Variables can be declared in the executable section

❖ Literal constants
- A literal constant is a set value that never changes and it is what the programmer has literally typed in their code
- **Boolean**
  - Two possible values 1 (true) and 0 (false)
  - Include the **stdbool.h** library
- **Character**
  - Enclosed between two single quotes (apostrophes)
  - Notice control characters (table 2-6, pg. 48)
  - 1 byte
- **Integer** (defaults to int type)

        - Written as is, can use **U, L, LL**
            - **LL** long long integer
            - **U** unsigned integer
    - **Floating-point** (defaults to double type)
        - Written as is, can use **F, L**
        - We will mainly be using **F**
    - **String**
        - Zero or more characters enclosed in double quotes
- More Constants
    - **Defined constants**
        - #define *identifier value*
        - #define is the best type of constant to use in the C language
    - Memory constants (**don't use**)
        - const *type identifier* = *value*;
        - Follows the same syntax as defining a variable
    - Points
        - Must appear at the top of the program
        - We will write constants in ALL CAPS
        - We will use the underscore to separate constants
- Rules for Constants
    - Limit using literal constants, especially when they could be more understandably replaced by defined constants
- Input/Output Streams
    - Data is input to and output to a stream
    - A stream is a source or destination for data
    - Two types are available: **text** and **binary**
        - **Text streams -** all data is sent via characters
            - Less efficient for the computer but much easier to work with
        - **Binary streams -** all data is sent via 1s and 0s
            - More efficient for the computer but more difficult to work with
    - A **monitor** can only display a text stream
        - Called **standard output**
    - A **keyboard** can only send a text stream
        - Called **standard input**
- Using printf()
    - Used to display text and formatted data
    - Requires one parameter but has many
        - One **format control string**
            - Surrounded by double quotes
        - Zero or more **conversion specifications**
            - Each begin with a %
    - Each specification is replaced with the data that has been formatted during runtime
- Conversion specifications
    - Consists of a **%**, three optional modifiers, and a **conversion code**
    - Three major conversion codes
        - **%c**, **%d**, **%f**
        - **%c** - character
        - **%d** - integer
        - **%f** - floating-point
            - Floating-point will default to 6 decimal places
    - Four different modifiers
        - **Size**
            - Defines the size of the data type
            - **h**, **ll**, **L**
                - Lowercase **l** for a double - most common
                - Uppercase **L** for a long double
        - **Precision**
            - Specifies how many decimal places are displayed for a number
            - Only applies to floating-point data types
            - Starts with a period and ends with a number
                - %**.2**d
        - **Minimum width**
            - Specifies at minimum how many spaces the conversion will take up
            - %**2**d without a period

        - %**2**.2d with a period
    - **Flag**
        - Only works if there is a minimum width
        - Generally works with int, char, and float
        - **Justification**
            - This will change the alignment to the left
            - %**-2**d
        - **Padding**
            - Only works if you are right aligned
            - %**02**d
        - **Sign**
            - Can be used with justification or padding
            - %**+2**d
- Using scanf()
    - Used to take text from the keyboard, format that data, and store it into variables
        - One format control string, with one or more conversion specifications
            - Surrounded by double quotes
        - Zero or more variable addresses
            - Each begin with the **&** (address operator)
                - %2d, **&***variable*
                - **Ampersand**
                - This allows us to "read in" data
    - Must have at least one conversion specification
        - **Size** (Modify the type)
        - **Flag** (assignment suppression = **"%*d"** we will never use this)
        - **Maximum width**
            - Reads to the maximum number or whitespace
        - **Precision cannot be used in a scanf()**
    - scanf() ceases to execute code until the user has entered in data
- Reading values
    - scanf() can be used to read in any type
    - When the user enters ints or floats, a space must separate each
        - Spaces, tabs, or enters
- Prompting the user
    - Be clear on the type of data needed
    - Usually used before a scanf() statement
- What if...
    - The program asks for 3 numbers, yet the user only enters 2?
        - The program will just wait until another number is entered in
    - The program asks for 3 numbers, yet the user enters 4?
        - The extra number typed in will be left on the text stream and will be automatically used in the next scanf() statement if there is one in the program
    - The program asks for a character, yet the user enters and integer?
        - The first integer is used and any other numbers or letters are left on the input stream
- Characters and scanf()
    - The enter character is a valid character so it can be left on the input stream
    - When reading numeric input, white space is skipped and only numeric values are read
    - Solution
        - **Add a space in front of the %c in a scanf()**
        - It skips all whitespace characters

```
scanf(" %c", &firstChar);
```

Chapter 3

February 11, 2019 8:00 AM

- Expression
  - A sequence of operands and operators that reduces to a single value
  - Expressions are generally thought of actions to perform
  - **Operator**
    - A token or symbol that requires an action
  - **Operand**
    - An object (value) on which an operation is performed
      - 1 + 2 - the numbers are operands and the "+" is the operator
  - Expression categories (pg. 94)
- Statements
  - An action performed by the program, which generally ends with a semicolon (pg. 121)
  - Statements use expressions to perform actions
    - A **compound statement** does not use a semicolon
  - The value of an expression can be assigned to a variable
    - Often expressions are stored for later use
      - **Variable** = **expression**;
    - **The assignment operator is an expression that returns whatever value is on the right**
- **Binary** Expressions
  - Means **two** operands with **one** operator
  - Sub categories
    - **Multiplicative expressions**
      - **Multiplication, division,** and **modulus**
      - All multiplicative expressions are binary expressions
    - **Additive expressions**
      - **Addition** & **subtraction**
- Math Operations
  - Operations in C are related to data types
    - Int + int = int
    - Int + float = float
  - Same for **subtraction**, **multiplication**, and **division**
  - Decimals are **truncated** not rounded when converting data types
- **Division** - the Same (but different)
  - Float / int = float
  - Int / int = int
- **Modulus** (**%**) Operator
  - The modulus operator only works with two integer operands; cannot use floats
  - Its used to give the remainder from division
- Expression Types
  - **Simple expression** contains only **one operator**
  - **Complex expression** contains more than one operator
- **Order of Precedence**
  - Complex expressions can reduce to simple expression
  - Operators are evaluated by 'priority'
    - See front cover
- **Associativity**
  - Direction used to evaluate operators of the same precedence
    - 8 / 4 * 2

        - 2 * 2
    - An expression always reduces to a single value
- **Primary Expressions**
    - One operand with no operator
    - Includes
        - **Names**
            - Variables, functions
        - **Literals**
        - **Parenthetical**
            - Any set of values or expressions within parentheses
    - There is no associativity with primary expressions
    - Functions do have associativity
- **Postfix** Expressions
    - One operand followed by one operator
    - Includes
        - **Postfix increment** "**++**"
        - Can be a statement or part of an expression
            - **Variable++;**
        - **The value is determined before the variable is incremented** (the side effect)
        - **Postfix decrement** "**--**"
            - **Variable--;**
- **Prefix** Expressions
    - Includes
        - **Prefix increment**
            - **++variable;**
        - **The value is determined after the variable is incremented** (the side effect)
        - **Prefix decrement**
            - **--variable;**
- The Difference is Timing
    - With prefix and postfix, the result is the same... To that variable
    - When it happens becomes critical
- Used as a Statement
    - It is common to see a single statement
        - Called an **expression statement** (pg. 122)
    - In this situation, prefix and postfix are the same; **normal use is postfix**
- Additional notes
    - Be careful in complex expressions

      myC = myX++ * --myZ
    - Cannot use post/pre-fix on multiple variables in parentheses or constants and literals
    - If it is confusing or unclear, make it clear
        - Correctness overrides conciseness
    - Do not attempt to modify a variable more than once in an expression -- undefined
- **Unary** Expression
    - An expression that has one operator and one operand
    - The **sizeof()** operator
        - Gives the size, in bytes, of a type or expression
    - The minus operator
        - Changes the sign of a value (+ - or - +)
            - It does NOT change the value of the variable itself
    - The address operator (ampersand)
        - Gets the memory location of a variable
- **Implicit** Type Conversion
    - When working with two values of different types, one type is automatically converted the higher-level type

        - An int + a float results in the calculation being that of a float type
    - Implicit type conversion rank
- **Explicit** Type Conversion
    - **Type cast** operator
        - Int x = 5,
          y = 2;
          float z = **(float) x** / y;
    - Casting overrides the type of the value for that calculation only
    - Both implicit and explicit conversions take extra processing
- Compound assignments
    - Common to see simple assignments
        - A = A + 2;
          B = B * 5;
    - Compound assignments can be used
        - A += 2
          B *= 5;
    - Operators **+ - * /** can be used
    - Note that it has a low order of precedence
- Statement Types
    - **Null** statement
        - Line with just a semicolon
            - **;**
            - Seems odd, but some situations call for it
    - **Expression** statements
        - Expression with a semicolon
            - Complete side effects and discard the expression value
                - **1 + 2;**
    - **Return** statement
        - **Terminates** a function
    - **Compound** statement
        - Zero or more statements surrounded by curly braces, needs no semicolon
- Final comments
    - Limit the use of expressions in printf() statements (pg. 10 style guide)
    - Know the order of precedence and prefix, postfix
    - When the precedence is equal for two operators, associativity takes over
        - A few operators associate right to left

Chapter 14

February 15, 2019 8:37 AM

- **Logical Bitwise** Operators
  - **&** - And
  - **|** - Inclusive Or
  - **^** - Exclusive Or
- **One's Complement** Operator
  - **~** - Flips the bits
- **Shift** Operators
  - **<<** - Left Shift
    - Multiplies by 2
  - **>>** - Right Shift
    - Divides by 2
- Final Comments
  - Why are these used?
    - Allows low-level data manipulation
    - Increase storage potential
    - Masking data

Chapter 4

February 21, 2019 8:01 AM

- **Functions**
  - Are a critical part of programming
  - Breaking big tasks into smaller tasks makes sense and is more manageable
  - Every C program is a collection of functions
    - Must have main()
- Why use functions?
  - Allow the breaking up of programs
  - Makes programming easier
  - Promote abstraction
  - Can be reused and shared
  - Permit multiple programmers
  - Permit customizing to fit one's needs
- Structure/Hierarchy Charts
  - Top-down design is a popular method in programming and system development
- Function calls
  - When main() calls a function that code is loaded and then executed
- Creating a Function
  - A function exists in 3 places (p.155)
    - **Declaration**
      - Appears in the global declaration section
    - **Definition**
      - Appears after main()
      - Actual code to perform the function's task
    - **Function call**
      - Inside main()
      - Appears when using the function
- Function **Declaration**
  - Appears in the global declaration section
  - Consists of
    - Function **return type**
    - Function **name**
    - **Parameters**
    - **Semicolon** (null)
    - Example:
      - void clearScreen();
        int calculateSum(int one, int two); - one and two are needed for style not syntax
        return; - must return nothing in a void function
- Function **Definition**
  - Appears after main()
  - Consists of:
    - Function **return type**
    - Function **name**
    - **Parameters**
    - **Curly braces** and **body**
- Function **Calls**
  - Appears in main() or in another function
  - Consists of:

        - Function **name**
        - **Parameters**
- A Process to Writing Functions
    - **Decide** what the function **does**
    - **Choose** a **name** for the function
    - **See** what data the function **needs**
    - **Determine** the **type** of data to return
- Return values
    - Just because a function returns a value does not mean the program must use the value
    - printf() and scanf() both return a value
    - Sometimes, the value returned by a function is important sometimes it is not
- Miscellaneous Comments
    - Automatic return type
        - A function without explicit return type is an int
            - Although previous standards did not, the C99 standard requires an explicit return type
    - Return
        - Causes a function to stop at the statement and immediately return
        - Avoid printing in functions that perform a calculation
- **Function Documentation**
    - Generally, program documentation should consist of:
        - Comments at the beginning about the entire program
            - Comments before each function definition
        - Basics would be:
            - Explain what the function is supposed to do
            - Describe the parameters
- Math Functions
    - Used by including the **#include <math.h>** library
        - When compiling, include an **-lm**
        - **c99 -Wall myProg.c -o myProg.exe -lm**
    - Power & square root functions (p. 191)
    - Absolute values (p. 187)
        - Int versions part of **#include <stdlib.h>**
    - Ceiling functions (p. 188)
        - Always round **up**
    - Floor functions (p. 189)
        - Always round **down**
    - **Truncate & round functions (p. 190)**
- Random Numbers
    - It's a pseudo random number
    - First, seed the random number
        - **srand(value);**
            - For example:
                - **srand(time(NULL));**
    - Second, get a random number
        - **rand()**
        - Will give a random integer value
    - To create a **range** of random numbers
        - **rand() % range + minimum**
        - **rand() % 10; 0-9**
        - **rand() % 10 + 1; 1-10**
    - The library **stdlib.h** is needed for **rand()** and **srand()**
    - The library **time.h** is needed for **time(NULL)**
    - Use **srand()** only once for each random number series

- Parameters & Variable Scope
  - Each function is its own **self-contained unit**
  - Functions can declare variables within themselves
    - Called local variables
    - Only visible within the block where created
  - When one function needs values from another function, parameters must be used
    - DON'T USE GLOBAL VARIABLES
    - Visible within any part of the program
  - Parameters are *similar* to local variables
  - Parameters match up in left to right order
    - **Type** is **important**
    - **Name** is **unimportant**
  - Functions can accept and return any type
    - A function can only return one value at a time
  - The **void** type is used to indicate 'none'
    - **void displayValue(int number);**
- Pass by Value
  - by default, parameters are passed to a function by value
    - Technically, all parameters in C are pass by value
  - In other words, a copy of the **value** is made and given to the function
  - The original value in memory is left untouched
- Pass by Reference
  - Parameters can also be passed by reference
    - This works by passing an address value
      - A copy of the **location** is sent
    - Sometimes called **pass by address**
    - The function can change the original
- How/Why is this done?
  - First
    - Pass the variables **address** to the function
    - Use the **address (&) operator**
      - **&x**
  - Second
    - Make the parameter a pointer in the declaration and definition
    - Use the asterisk (*) in front of the parameter
    - Style guide:
      - **void change(int *param);**
  - Third
    - Use the asterisk (*) in the parameter in the definition code to access the variable
    - **Dereference:**
      - ***param = 8;**
    - Never forget the asterisk (*)
- Why?
  - If the function must modify 2 values
  - Only do this if it's necessary
  - Sometimes a function can be broken into two sub-functions to avoid this

Chapter 5

March 1, 2019 8:25 AM

- Logical Data
  - **Logical data** conveys either true or false
    - boolean
  - In the past, C had no logical data type
    - False was 0; all other values were true
  - The **boolean** type is now available
    - 0 is still false; all other values are true
  - To display a boolean value, use **%d**
- Comparative Operators (p. 236)
  - **Relational operators**
    - Greater than - **>**
    - Less than - **<**
    - Greater than or equal to - **>=**
    - Less than or equal to - **<=**
  - **Equality operators**
    - Equal to - **==**
      - Must use 2 equal signs
    - Not equal to - **!=**
  - **Logical Operators**
    - NOT **(!)**
    - AND **(&&)**
    - OR **(||)**
  - All three result in a logical value **(1 or 0)**
  - AND and OR allow a test condition to evaluate more than one item at a time
- The '!' (NOT) Operator
  - Works the same way we use it in English
  - Changes true to false; false to true
    - 1 to 0 and 0 to 1
    - !(a > b) == **(a <= b)**
    - !(b <= 7) == **(b > 7)**
- The '&&' and '||' Operators
  - Gives one logical value from two values
    - (money > 50 && amount < 100)
- Short-Circuit Evaluation
  - In a logical operation, evaluation stops once the outcome is known
  - The following evaluates to **0** once any value is found to be **false**
    - **W && X && Y && Z**
  - The following evaluates to **1** once any value is found to be **true**
    - **W || X || Y || Z**
  - **Y** gets evaluated first, then **Z**, then **X**
    - **X || Y && Z**
- Conditional Statements
  - Decision making deals with choosing to execute one set of statements over another
  - C's simple **if** structure:
    - If (test condition)
      {
      Statement;

Statement;
}

- If you put a semicolon after the parentheses or after curly braces is a **logic error**

- Curly Braces...Required?
  - If there are **2 or more** statements use curly braces
  - **STYLE GUIDE:** ALWAYS USE CURLY BRACES NO MATTER WHAT!
- If-else Statement
  - Every if statement has two paths
    - A simple **if** statement, true path does something; false path is empty
    - An **if-else** statement, both paths perform different tasks
  - C's if-else statement (p.238)
  - Syntax rules (p.239)
  - Conditions only go with if
  - Nested if statements
    - True path (p.243); false path (p.261)
  - **Mutually exclusive statement**:
    - If (x < 0 && x > 10)
  - **Overlapping statement**:
    - If (x > 0 || x < 10)
- Conditional Operator
  - This operator is used for simple tests (p. 247)
  - Basic syntax:
    - <condition> ? <true value> : <false value>
      value = A > B ? 25 : 19
- Switch statement (**nested if's could always be used instead**)
  - Sometimes, multiple tests need to be performed on the same variable
  - The test value must be an integral type
    - No floats
    - This function can be a variable, function
  - Ranges **cannot** be done in C
    - This **can** be done in nested if statements
  - The default case is taken if no match is found in the switch
    - Is not required
    - Does not have to be the last item
    - Is required for our class style
- The "break" statement
  - The break statement jumps to the **end** of the switch
  - Break statements are important but are **not** required
  - However, without break statements, control falls through the remaining cases
    - Sometimes this may be desired (p. 258, 260)
- The **goto** statement
  - C has them but **DON'T** use them
  - Allows the control to jump to another part of the program
  - Use of **goto** tends to create "spaghetti code" that is difficult at best to follow
    - Adding comments doesn't make bad code good
- Final comments
  - One of the biggest benefits of a switch statement is it's great for **menu-driven** programs
    - Don't forget the curly braces
    - Use single quotes around characters
    - Remember the break statements
    - STYLE GUIDE: do not indent like in the book
  - Classifying and conversion functions (p.266)
  - Exiting functions (p.268)

- Don't ever use them
- **exit(1);**

Chapter 6

March 13, 2019 8:02 AM

- Looping
  - Repeats a section of code
  - Common in algorithms and program logic
  - Types
    - **Definite iteration**
      - Count from 1-100
    - **Indefinite iteration**
      - Prompt the user for a value from 1 to 10; continue until the user does so
      - **User** data and **Sensor** data are usually indefinite
    - **Infinite iteration**
      - Loops forever (this is generally a bad thing)
- Looping Constructs in C
  - **Pre-test loop**
    - Test at the **beginning** of the loop
    - Like asking permission before you do something
  - **Post-test loop**
    - Test at the **end** of the loop
    - Like asking forgiveness after you've done something
  - C provides three loops:
    - Indefinite, pre-test (**while** loop)
    - Indefinite, post-test (**do while** loop)
      - Both are **event-controlled** loops
      - Do-while is the only post-test loop
    - Definite, pre-test (**for** loop)
      - **Counter-controlled** loop
- Initialization & Updating
  - Initialization
    - Setting loop variables to starting values
    - Can be explicit or implicit
      - **Explicit** - start at a value
      - **Implicit** - start at a value read-in from the user
  - Updating
    - Something inside the loop to change the condition, called a **loop update**
    - Performed each iteration (time through the loop)
- The while loop (p. 310)
  - Repeat a section of code from 0 to possibly infinite times based on a test condition
  - Test condition evaluated before entering the loop
    - If true, enter the loop; false, skip the loop
  - A **priming read** is often needed (implicit initialization)
  - Once control enters the while loop, the entire body of the loop is executed
  - At the end, control returns to the top and re-evaluates the test condition
    - True, loop again; False, exit the loop
  - At some point, the test condition must evaluate to false
- The do while loop
  - Repeat a section of code at least once
  - The difference between while and do while
    - While is a pre-test loop

        - Do while is a post-test loop
    - It is the least used loop
    - Syntax: (p.320)
- The for loop
    - Repeat a section of code a definite number of times
        - **for (x = 1; x < 10; x++)**
    - **Initialization** (starting conditions)
        - Executed only once when the loop begins
    - **Test condition** (limit-test condition)
        - Must be a logical condition
        - Evaluated before each loop iteration
        - Must evaluate to TRUE to continue looping
    - **End of loop expression** (update expression)
        - Executed <u>at the end</u> of each iteration
        - It will happen the same number of times as the loop body
- Also note
    - Can have more than 1 initialization and more than 1 end of loop expression
        - **for (j = 2, k = 4; j < k; j += 2, k++)**
    - Parts can be left "blank" if not needed
        - **for (; j < k; j += 2)**
    - It is best to always to have a condition and update
    - The control variable can be changed within the loop body
- Loop interchangeability
    - Any looping structure can be rewritten with either of the other two structures
    - How to pick the correct loop?

<u>Missing notes</u>

- Nested loops
    - Any loop can be placed inside another
- Exiting a loop
    - All C loops end when the test condition is false
    - Sometimes, the loop must end before reaching the test condition
    - The **break;** exits the current loop
        - Use a flag value instead (p. 339, 340)
        - Does not exit nested loops
    - The **continue;** skips directly to the test condition
        - End of loop expression is still executed in a for loop
- **Recursion**
    - WARNING! An advanced topic (briefly discussed)
        - -**T**he **T**tp **P**roject
    - Recursion is a repetitive process in which a **function calls itself**
        - This includes a function calling a second function which in turn calls the first function
    - Primary limitation
        - **Eats up memory**; lack of efficiency
    - Keep in mind that the recursion **must end** sometime; infinite recursion is not good
        - Every recursion function call must get closer to solving the problem
        - The TTP Project is infinite recursion
        - CSS 227 will talk more about this
        - **Do not use recursion in this class**

Chapter 7

March 22, 2019 8:02 AM

- Introduction
  - File input/output is a fundamental process
  - A **file** is an external collection of related data treated as a unit
  - Two classes of files:
    - **Text** variety
      - All the data is stored as **character** values
      - Easier to work with
      - **WE WILL BE USING TEXT FILES FOR THIS CLASS**
    - **Binary** variety
      - All the data is stored as **binary** values
      - More efficient
  - This chapter deals with text files
    - Contain human-readable characters
- Steps to working with a file
  - **Declare a file variable** (points to the file)
    - Create a stream
  - **Open the file**
    - Associate the stream with the file
    - The **operating system** prepares the file
  - **Use the file**
    - Input or output
  - **Close the file**
    - Tells the operating system to **release the resources**
- **Declare a file variable**
  - Uses the **<stdio.h>** library
  - A variable is used to store a value
  - A file variable is used to refer to a collection of values
  - Declare the variables

    FILE *fileIn;
  - The **FILE** type is in all caps
  - There is a star **(asterisk)** in front of the file's variable name
  - ALWAYS PUT THE **ASTERISK** ON THE **VARIABLE NAME** NOT THE DATA TYPE
- Open the file
  - Associate the FILE reference variable with a file on the computer
  - For this, use the **fopen()** function which is part of the <stdio.h> library
    - fopen("filename", "mode")
      - Filename is the variable we are trying to access
    - This function returns as a return value
  - **Keep files in the same location as the program**
  - Whenever you call **fopen()** you will always **store the result**

    fileIn = fopen("text_in.txt", "r");

    fileOut = fopen("data_out.dat", "w");
- Modes for fopen (it is a string - use double quotes)
  - **"r"** (**read**) - **input** - **load** - open to read from a file
    - **Error if there is no file**
    - The only mode where we have to worry about a problem
  - "**w**" (**write**) - **output** - **save** - **store** - open to write to a file

        - Always starts from an **empty file**
        - If the file exists, **delete** and **replace**
        - If the file does not exist, **create** it
    - "**a**" (**append**) - opened at the **end** of a file
        - If the file does not exist, **create** it
- Close the file (p. 398 - 401)
    - **Close the file** (when finished) **to free system resources**
        fclose(fileIn);
        fclose(fileOut);
    - Files can only be open in **one instance at a time**
- Reading a file
    - Use the **fscanf()** function to read a file
    - Similar to scanf(), with 1 difference - the first argument is the **FILE** reference variable
        fscanf(fileIn, "%d", &valueIn);
            - No asterisk
    - The **fscanf()** can only be used for files that have been opened for reading
    - "Obviously" the format/contents of the file must be known to read correctly
        - Know your data!
- Writing a file
    - Use **fprintf()** to write to a file
    - Similar to printf(), with 1 difference - the first argument is the **FILE** reference variable
        fprintf(fileOut, "%d", count);
    - The **fprintf()** can only be used for files that have been opened for write or append
- Error Checking
    - Page 402 shows file error checking
    - We will be using a modified technique than what the book shows
    - Only error check when you are in "read" mode
        - fopen() - returns a **pointer** or **NULL**
        - fclose() - returns **0** or **EOF**
- Read Until EOF
    - The **fscanf()** function returns a value; used in **indefinite** situations
    - The value is **EOF** if it encounters the end of a file when it reads from the file
        while (fscanf(fileIn, " %d", &value) != EOF)
    - The above while loop will read integers over and over from a file until it reads the end of the file
- Control Breaks
    - This is a break in the loop to perform extra processing
- Special File Variables (p.397)
    - **stdin** - scanf
    - **stdout** - printf
    - **stderr** - printf
    - They are already declared and open, so never open them, and never close them
- FILE Variables as Parameters
    - Pass the variable like pass by reference
        - But in the call there is **no ampersand** before the file variable
- Additional Functions (p. 433)
    - The process one character at a time and are found in <stdio.h>
        - **getchar()**
        - **putchar()**
        - **fgetc()**
        - **fputc()**

Chapter 8

April 3, 2019 8:02 AM

- New Data Structure - Array
  - **Sequenced collection** (order) of elements of the same data type
  - Instead of one name for one value, **an array is one name that references a number of cells**

    int singleValue;
    int manyValues[100];
    float singleTemp;
    float manyTemps[31];
- Visualizing an Array (p. 462 & 464)
  - Think of an array like rooms on a hallway

    int ageArray[10];
    An integer can be placed into each of the cells
- Array indexes
  - The **array index** [**subscript**] refers to the slot or location in an array
  - The actual value within the slot is called an **element**
  - The first box of all array subscripts always start at 0
  - An array of 10 elements looks like this:

    [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
- Declaring an Array
  - Arrays are **derived types**
  - Two type of arrays:
    - **Fixed-length** (in the book)
      - Data type, name, and [*constant*] size are required
      - Choose a max size that will be needed
    - **Variable-length** (in the book)
      - Data type, name, and [*variable*] size are required

        int bigGradebook[students]
      - most people call all of these examples fixed-length arrays
      - **C only supports fixed-length arrays**
- Initializing Arrays (**p. 466**) - **memorize picture examples**
  - Types (works with any data type)
    - **Basic**
      - Fills every box with a value
      - The order of numbers matters

        int x[5] = {3, 7, 12, 24, 25};
    - **Without size** (bad practice)
      - Same amount of values as there are boxes, indefinite

        int x[ ] = {3, 7, 12, 24, 25};
    - **Partial**
      - Defaults the rest of the array boxes to 0 if there is no value given

        Int x[ ] = {3, 7}; (3, 7, 0, 0, 0};
    - **All zeros**
      - Will set all the values to 0

        int x[1000] = {0}; (1000 zeros)
  - You can only declare and initialize the array once (on creation)
  - Specifying more values than elements int the array is a compiler error
- Assigning values
  - A single array index acts as a variable
    - The index must be an integral value (whole numbers)
  - For example:

    scores[0] = 23;
    scores[counter] = 8;
    scanf("%d", &scores[index]);
    fscanf(fileIn, "%d", &scores[ctrl]);
    scores[x] = scores [x + 1]; - as long it evaluates to an integral value

- Be careful of the **upper array bound!**

❖ Arrays and Addresses
- The name of an array is a **constant address**
- Assume the address for the following array is 10000
  int arrayName[10];
- When using an array name and subscript
  arrayName[5] = 25;
- C takes the base address
  10000
- And adds to that address the subscript times the data type size
  10000 + (integerSize * 5)
- Resulting in the address
  10020 and index **[5]**

❖ Processing Arrays
- An index must be used when getting a value from or assigning a value to an array
  int testScores[20];
  testScores[0] = 95.3;
  testScores = 88.7; - **not correct**
- Arrays must be processed element by element
  char firstArray [3] = {'a', 'b', 'c'};

❖ Code Examples
- It's common practice to **print all the array values** using a for loop
- You can also **swap values between each array index** as needed
- You can also **reverse all the values** as needed

❖ Array Elements as Parameters
- Since an array name and index is just a variable, pass it to a function the same as a variable
  - Pass by value:
    funFunction(myArray[]);
  - Pass by reference:
    funFunction(&myArray[]);
- The formal parameter is declared as before (**p. 474-475**)

❖ Arrays as Parameters
- **Entire arrays are passed by reference only**
- The function receiving the array knows the type of values, but not the array size
- When the function operates on that array, the actual values in the original array are affected
- To prevent a function from possibly changing array values (when it should not), use const before the array formal parameter

❖ How is This Done...
- First, the function prototype/heading
  - The formal parameter is an "**anonymous arra**y"
    void loadArray(float arrayTemps[]) - leave the square brackets empty
  - The parameter is the same type to be passed to it without a size given
  - If you need the size in the function, do this:
    void loadArray(float arrayTemps[], int size)
- Second, the function call
  - Just pass the array name
  - No type and no square brackets
    loadArray(arrayTemps);
  - Remember, the name of an array is an address; passing an address of a structure is a pass by reference

❖ Array Applications
- **Frequency array**
  - Shows how often a particular value occurred
  - Example:
    - Numbers of A's, B's, C's, D's, and F's in a class
- **Histogram**
  - Visual representation of a frequency array
  - Example:
    - **A's ******
      B's *******
      C's **********

D's ****
F's **

- Multidimensional Arrays
  - So far, only **one-dimensional arrays** have been discussed
  - C can also have **multidimensional arrays**
- Creating and Using These
  - Declaration examples:

```
int    scoreTable[3][5];
float arrayTemps[12][31];
```

  - Two dimensions:

```
scoreTable[0][0] = 15;
scoreTable[0][1] =  0;
scoreTable[1][0] = -3;
scoreTable[2][3] = 22;
```

  - Three dimensions:

```
Int x[2][3][4];
x[1][1][1] =  5;
x[0][2][3] = -2;
```

- Passing These Arrays
  - Passing the entire array
    - Because of the multiple dimensions, the formal parameter must specify the 2nd dimension's size
    - Example:

```
void loadBigArray(int scores[][5])
```

    - 3-dimensional:

```
void loadBigArray(int scores[][5][3])
```

    - The program can also pass just one row of a multidimensional
- Common Use for Arrays
  - A common use of arrays is for sorting data
  - Let us look at some sort algorithms: **(p. 491-498**)
    - **Bubble sort** - sorts by switching values side by side
    - **Selection sort** - sorts by switching specific values no matter their order
    - **Insertion sort -** sorts by putting numbers in the correct order as it reads them
- Another common Use of Arrays
  - Another common use is for searching a set of data
  - Let us look at some search algorithms: (**p. 501-509**)
    - **Sequential search** - starts from 1
    - **Binary search** - starts from halves

Chapter 11

April 15, 2019 8:00 AM

- Introduction - **program11 (p.495-509)**
  - **A series of characters treated as a unit**
    - C has no string data type
  - Strings by nature **vary in length**
    - People's names
  - A string in C is a **variable-length array of characters** that is **delimited** by the **null character: '\0'**
  - A string in C is an array of characters
    [H][E][L][L][O][\0]
- Null Character Importance
  - **The null character ('\0') is vital to a string**
    - It literally ends the string
  - Without it, C would not know the end of the string
    - Strings can vary in length
  - Remember, when an array is passed to a function, the function knows:
    - The array type
    - The beginning of the array (address)
    - *But it does not know the length
  - Most of the time, the null character is handled for us; only rarely does the programmer need to worry about it
  - Lastly, keep in mind that the null character **takes up 1 slot in the array**
  - The **numeric value** of the null character is **"0"**
- Declaration and Initialization
  - Simply declare an array of chars
    char lastName[10]; - can have a last name of 9 characters
    - You can also enter in each character individually
      char lastName[10] = {'T','e','i','c' . . . };
    - Or use double quotes (**shortcut**)
      char lastName[10] = "Teichroeb";
    - You can also use Without Size but this can never change after creation
  - **The string must be initialized on creation**
  - **A string is an array**
- String Input and Output
  - Using **scanf()** and **printf()** will work for strings
  - The field specification is **%s**
  - For example:
    scanf("%s", lastName);
    printf("%s", lastName);
  - Be sure to make the array large enough
- Problems with **scanf()**
  - Two problems arise when reading strings using **scanf()**
    - It reads all characters until white space
      - **Could overflow the array**
    - It stops at any white space
      - **May not read the entire string as it starts**
  - Solutions:
    - Place a width on the field specification
      scanf("%20s", name);
      - Solves only 1 problem - stops array overflow
    - Use the **gets()** function
      gets(name);

        - Solves only 1 problem - allows white space
    - Use the **fgets()** function
      fgets(name, sizeof(name), stdin);
      or
      fgets(name, [constant], stdin);
        - Solves both problems but does still read spaces
- More on String Output
  - Already saw **printf()**
    printf("%s", name);
  - Also have **puts()**
    - **puts(name);**
    - Note: forces a return '\n' after the string
    - The same as:
      printf("%s\n", name);
  - As well as **fputs()**
    fputs(name, stdout);
    - No size needed here
    - Also no return '\n' forced after the string
  - As well as **sprintf()**
    sprintf(outputString, "Data %d", x);
    - Allows you to take data and print it to a character array
- What Cannot Be Done
  - Assume the following
    char lastName[10];
  - Each of the following are illegal
    lastName = "Johnson";
    lastName = newLastName;
    If(lastName == newLastName)
  - Because strings are not basic data types in C, string manipulation functions are needed
    - Header file to include is **<string.h>**
- **strlen()**
  - Returns the length of a string (number of characters (**int**)) less the null character
    - If the string is empty, zero is returned
- **strcpy()** and **strncpy()**
  - **strcpy()** copies the contents of one string into another
  - **strncpy()** copies a specified number of characters of one string into another
    char lastName[10] = "Johnson"
    char newName[10];
    strcpy(newName, lastName);
    strncpy(newName, lastName, sizeof(newName) - 1);
  - It is the programmer's responsibility to be sure that the receiving array is large enough
    - BTW, note **page 695**
- **strcmp()** and **strncmp()**
  - **strcmp()** compares two strings until unequal characters are found (if any)
  - **strncmp()** compares a specified number of characters of two strings until unequal characters are found (if any)
  - Both return an integer
    - **Zero** - two strings are equal - **0**
      If(strcmp(name, "Jill") == 0)
    - **Less than zero** - first string is less than second **< 0**
      If(strcmp(name, "Jill") < 0)
    - **Greater than zero** - first string greater than the second **> 0**
      If(strcmp(name, "Jill") > 0)
- **strcat()** and **strncat()**
  - **strcat()** appends one string to the end of another string

- **strncat()** - appends a specified number of characters of one string to the end of another string

```
char title[5] = "Mrs.";
char lastName[10] = "Johnson";
char wholeName[15];
strcpy(wholeName, title);
strcat(wholeName, lastName);
strncat(wholeName, lastName, sizeof(wholeName) - strlen(wholeName) - 1);
```

❖ Arrays of Strings
- What would an array of strings be?
  - A 2-dimensional array of characters

```
char wholeName[4][10] = { "Jack", "Jill", "Bob"};
```

- This concept is important because it allows a program to operate on multiple strings like people's names

❖ Creating and Using These
- Declare the array

```
char csTeachers[6][10];
```

- The 6 is the number of rows; 10 is the number of columns
  - Last names can be max length of <u>?</u>
- To read the names

```
scanf("%9s", csTeachers[num]);
```

- Notice, that the array plus a subscript is used
  - In a 2-dimensional array, only using one subscript still yields an address

❖ Additional String Functions
- Character in string
  - Searches for the first occurrence of a character within a string; returns a pointer

    strchr(lastName, 'a');

    strrchr(lastName, 'a'); - the extra 'r' is for reverse searching
  - If it **does not** find any characters that it is searching for it will **return NULL**
  - If it **does** find it, it will **return** the **location in memory**

❖ String in string
- Searches for the **first occurrence** of a string within a string; **returns a pointer**

  strstr(lastName, searchString);

❖ Data in string (p.713)

```
sscanf(lastName, "%s", first_part);
sscanf(someString, "%d %f", first_part, second_part)
```

❖ Additional Strings Functions
- String to long (p.705 lines 15, 18, 21)

  strtol(someString, ,) - will take a string and convert it into a long type
- String to double

  strtod() - always base 10

Chapter 9

April 29, 2019 8:00 AM

- Pointer Understanding
  - A pointer is a variable that **points to**, or **refers to**, another variable
    - A pointer stores a **memory address**
  - A pointer variable of type "pointer to **int**" is a ***reference*** to another variable (of type **int**)
  - Pointers provide a way to refer to (get access to) a **cell of memory**
  - Why have a variable that refers to another variable, why not just use the variable itself directly?
    - Using pass by reference
    - Sorting of large structures of data (arrays)
- Pointer Declarations
  - Making pointers is easy
    <type> *<variable_name>;
  - For example:
    int *pCount;
    int *pTemp;
  - When a pointer is declared, the compiler must know the type of data to which it points
  - **All** pointers **hold an address**, but the ***type*** of what they point to is a key issue
  - **All** pointers take up **4 bytes** of data
- Item of Note
  int *pCount;
  - Creates a variable called **pCount** which **points to an integer** - it is NOT an integer
  - Right now, **pCount** is not pointing to anything
    - Technically, the pointer variable will contain **garbage**
- Pointer Initialization
  - To point a pointer at something, assign that pointer the **address** of the item
  - For example:
    int count;
    int *pCount;
    pCount = &count;
    - **pCount** is now pointing to the location of **count**
  - To initialize a pointer to point to nothing
    int *pCount = NULL;
- Pointer Usage
  - To get the **value** that the pointer is pointing to, use an **asterisk** and the **pointer name**
    int count = 17;
    int *pCount;

    pCount = &count;
    count = count / *pCount;
    printf("%d", *pCount);

    count will be **1**
  - To change what the pointer points to, use the pointer name without the asterisk
    pCount = &value;
- Important...
  - When creating a pointer, use the ***
    int *pToSomething; - stores a location to this data type
  - When pointing it, don't use the *

pToSomething = &something;

- Whether using or modifying that value
- For example:
  ***pToSomething = *pToSomething * 2;**
  printf("%d", *pToSomething);

❖ Pointer Usage

- Pointers do not do normal calculations
- Addition and subtraction on pointers is different then normal math
- Adding one to a pointer move to the next address of that data type
- Use **%p** to print out the location of a variable or array