

C Programming For Beginners - Master the C Language

Section 1: Introduction

1. Welcome to Class!

- Focus on writing high quality code

- Overview of C:

- efficient
- portable
- power
- flexibility
- programmer oriented

- Language features:

- imperative language
- top down planning
- structured programming
- modular design

- Advantages of using C:

- small
- fast programs
- reliable
- easy to learn and understand

- We're gonna use the VScode IDE

- write
- edit
- debug

- Basic C concepts:

- structure of a program
- comments
- output using printf

- Makefiles:

- how to build
 - a tool to build and compile code

- Variables:

- declaring
- using

- variables is the idea of associating data and memory for your program with a name

- Data types:

- int
- float
- double
- char

- later in the course we're gonna learn more advanced data types like:

- enums
- and providing your own data types by using the **type def** keyword

- Basic operators:

- logical
- arithmetic
- assignment

- Conditional statements:

- making decisions
- if switch

- Repeating code:

- looping
- for
- while
- do-while

- Arrays:

- defining
- initializing
- multi-dimensional

- Functions:

- one of the most important concepts related to the C programming language
- everything should be used and organized around a function, this refers to as module type programming
 - declaration
 - use
 - arguments
 - parameters
 - call by value vs. call by reference

- Debugging:

- ◊ finding and fixing your errors
 - call stack
 - can show you if your program crashes, where it crashed.
 - common mistakes
 - understanding compiler messages
 - the compiler checks if your code is syntactically correct and if not it'll give you error messages or warnings.
 - you need to understand what the compiler is telling you

- Structs:

- ◊ similar to arrays in that it stores data
- ◊ they're kind of like objects
 - initializing
 - nested structures
 - variants

- Character Strings:

- ◊ this is thought of as an array in C
 - basics
 - arrays of chars
 - character operations

- Pointers:

- ◊ this is gonna be our most important concept in the course
- ◊ we're gonna spend a lot of time on this topic
 - a concept in C where you have to manage your own memory
 - in C you have to worry about managing your own memory
 - definition
 - use
 - using with functions and arrays
 - malloc
 - pointer arithmetic

- The preprocessor:

- ◊ this step happens before the compiler
 - #define
 - #include

- Input and output:

- getchar
- scanf

- File Input/Output:

- ◊ reading and writing to a file
- ◊ file operations

- Standard C library:
 - ◊ this library comes with the programming language
 - string functions
 - math functions
 - utility functions
 - standard header files

- Course outcomes:
 - ◊ **you will be able to write beginner C programs**
 - understand the fundamental aspect of the C language
 - know how to provide input and output
 - do file I/O
 - drill down the concept of pointers
 - ◊ **you will be able to write efficient, high quality C code**
 - **this affects the maintenance phases of the software life cycle**
 - modular
 - things are grouped into functions
 - if you ever wanna add new functions or add new code to a program, it's much easier to do that if it's modulized, if it's organized.
 - low coupling
 - we're gonna make sure that in our different files we have low dependancies, so that they don't affect one another
 - when you have a lot of dependancies in your code, it's much easier to find and fix your bugs
 - it's much easier to fix errors if you have high quality code
 - we're not just gonna get things to work, we're gonna do it the right way
 - ◊ **you will be able to find and fix your errors**
 - an essential part of becoming a good programmer is **problem solving**
 - we're gonna focus on problem solving techniques and how to debug your code
 - and how to become an efficient problem solver
 - this includes understanding the compiler messages
 - how to use the debugger
 - how to use the call stack
 - ◊ **you will understand fundamental topics of the C language**
 - ◊ **we will have fun!**

2. Class Organization

- The class is organized around explaining the why and the how
 - ◊ Understand the whole programming language as a whole
- Powerpoint slides
- Demonstrations of code (IDE)
 - ◊ code examples (concrete, real world)
 - ◊ More practical/hands-on learning
- Challenges
 - ◊ coding assignments/projects
 - A way for you to assess your own learning
 - Code alongs - walking through the code of a solution for a particular challenge
 - All source code provided in class
- A walkthrough → code along

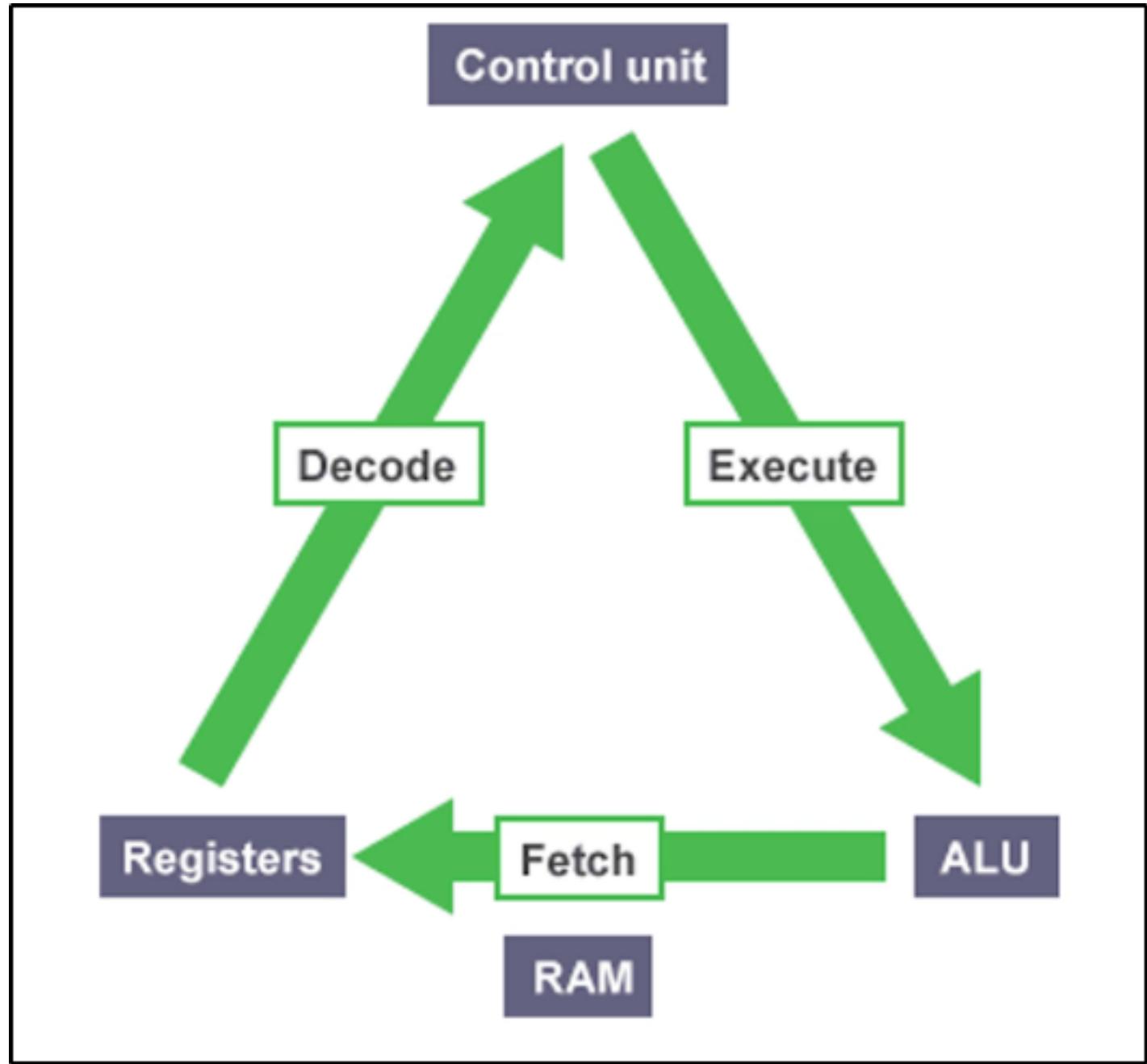
3. Fundamentals of a Program

- Computers are very dumb machines
 - ◊ they only do what they are told to do
- The basic operations of a computer will form what is known as the computer's instruction set
 - ◊ instruction set → how it's gonna do something
 - ◊ instruction set → they're basically the instructions on the CPU
- To solve a problem using a computer, you must provide a solution to the problem by sending instructions to the instruction set
 - ◊ a computer program sends the instructions necessary to solve a specific problem, this is done via commands to the CPU
- The approach or method that is used to solve the problem is known as an algorithm
 - ◊ So, if we were to create a program that tests if a number is odd or even
 - the statements that solve the problem become the program
 - the method that is used to test if the number is even or odd is the algorithm
- To write a program, you need to the instructions necessary to implement the algorithm
 - ◊ these instructions would be expressed in the statements of a particular computer language, such as Java, C++, Objective C or C.

• Terminology:

- CPU:
 - ◊ does most of the computing work
 - ◊ instructions are executed here
- RAM:
 - ◊ stores the data of a program while it is running
 - ◊ all the data in a program is stored in memory
 - ◊ the reason for storing in memory instead of a hard drive is because memory is much faster
- OS:
 - ◊ developed to help make it more convenient to use computers
 - ◊ controls all the operation of a computer
 - ◊ it makes life easier when using a computer because at a very low level a computer is just doing what it's told, and it's just using an instruction set, it'd be very difficult to use if you had to understand all of this instructions.

- ◊ a program that controls the entire operation of a computer:
 - all input and output
 - manages the computer's resources and handles the execution of programs
 - Windows, Unix, Android, etc.
- Fetch/Execute Cycle (life of a CPU)
 - ◊ fetches an instruction from memory (using registers) and executes it (loop)
 - ◊ a gigahertz CPU can do this about a billion times (instructions) a second



- ◊ on this image what's going on is that:
 - you're grabbing data from RAM
 - you're grabbing this data from RAM for instructions
 - you're constantly decoding that data which is stored in registers and RAM and executing it on different parts of the CPU, either:
 - control unit
 - ALU

- **Higher Level Programming Languages**

- Make it easier to write programs

- ◊ They are the opposite of Assembly language

- ◊ C is a high programming language that describes actions in a more abstract form

- ◊ the instructions (statements) of a program look more like problem solving steps

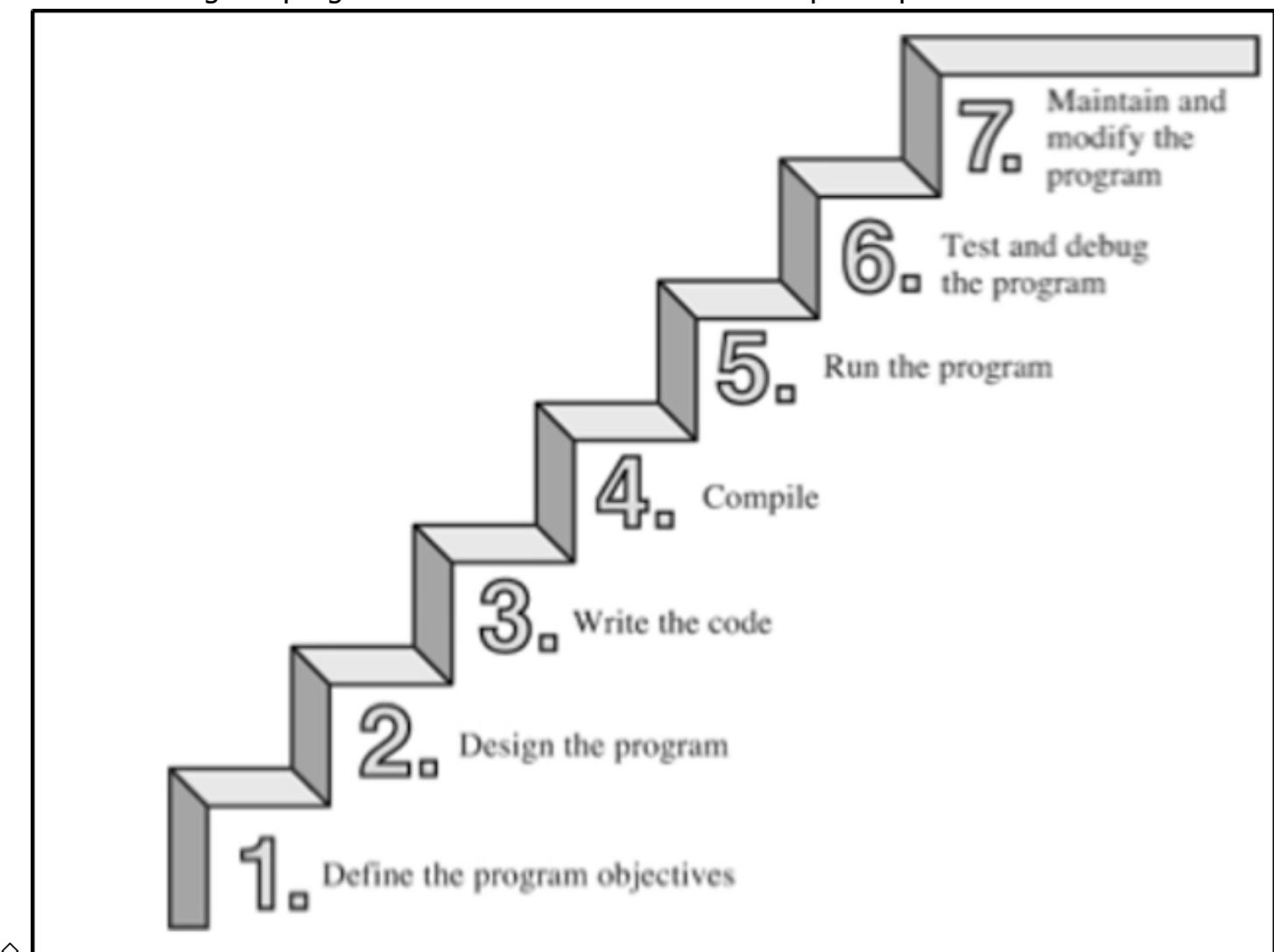
- Compilers:

- ◊ will check that your program has valid syntax for the programming that you are compiling

- finds errors and it reports them to you and doesn't produce an executable until you fix them

- **Writing a Program:**

- the act of writing a C program can be broken down into multiple steps:



- Steps in writing a program:

- 1) Define the Program Objectives

- understand the requirements of the program

- get a clear idea of what you want the program to accomplish

2) Design:

- decide how the program will meet the above requirements
- what should the user interface be like? (if it's required)
- how should the program be organized?
 - are you gonna create different functions and try to create different algorithms...

3) Write the code:

- start implementation, translate the design in the syntax of C
- you need to use a text editor to create what is called a source code file

4) Compile:

- translate the source code into machine code (executable code)
- consists of detailed instructions to the CPU expressed in a numeric code

5) Run the program:

- the executable file is a program that you can run

6) Test and Debug:

- just because a program is running, doesn't mean it works as intended
- need to test, to see that your program does what it's supposed to do (may find bugs)
 - debugging is the process of finding and fixing program
 - making mistakes is a natural part of learning

We'll be performing steps 1 through 6 over and over and over again

7) Maintain and Modify the Program:

- programs are used and released by many people
- have to continue to fix new bugs or add new features

For the above steps, you may have to jump around steps and repeat steps

- ◊ e.g. when you are writing code, you might find that your plan was impractical

• Steps in writing a program:

- ◊ Many new programmers ignore step 1 & 2 and go directly to writing code
 - a big mistake for larger programs, may be ok for very simple programs
 - the larger and more complex the program is, the more planning it requires
 - should develop the habit of planning before coding
- ◊ Also while you are coding, you always want to work in small steps and constantly test (divide

and conquer).

- example: write 5 lines of code, test it, write another 5 lines, test it...
 - this is gonna make much easier to find a fix your errors

4. Overview

- Overview of the C programming language:
- C is a general-purpose, imperative computer programming language that supports structured programming
 - ◊ uses statements that change a program's state, focuses on how
 - ◊ general purpose means that you can write all sorts of applications in C
 - ◊ imperative means that it's organized around statements, variables...
- Currently, it's one of the most widely used programming languages of all time
- C is a modern language
 - ◊ has most basic control structures and features of modern languages
 - ◊ designed for top-down planning
 - ◊ organized around the use of functions (modular design) structured programming
 - ◊ a very reliable and readable program
 - reliable: means that when you write the code it's gonna do what you told it to do, it's not just gonna crash for no reason
 - readable: means that when you look at the source code and statements, you can kind of understand what's going on
- C is used on everything from minicomputers, Unix/Linux systems to PC's and mainframes.
- C is the preferred language for producing word processing programs, spreadsheets and compilers.
- C has become popular for programming embedded systems
 - ◊ used to program microprocessors found in automobiles, cameras, DVD players, etc.
 - embedded means the program has to be efficient, it's not running on as much memory, not as much high level hardware
 - embedded can also be real time programming
- C has and continues to play a strong role in the development of Linux
 - ◊ Linux is the base for the macOS operating system.
- C programs are easy to modify and easy to adapt to new models or languages
- In the 1990s, many software houses began turning to the C++ language for large programming projects
- C++ is a subset of C with object-oriented programming tools added

- ◊ any C program is a valid C++ program
- ◊ by learning C, you also learn much of C++

- C remains a core skill needed by corporations and ranks in the top 10 of desired skills

- C provides constructs that map efficiently to typical machine instructions and thus is used by programs that were previously implemented in Assembly language
 - ◊ provides low level-level access to memory (has many low-level capabilities)
 - ◊ requires minimal run-time support
 - this is because it easily maps to assembly instructions
 - ◊ a lot of drivers are written in C

- **History:**
 - ◊ was invented in 1972 by Dennis Ritchie of Bell Laboratories
 - Ritchie was working on the design of the UNIX Operating System
 - UNIX is an antedecessor to the Linux OS
 - UNIX runs on different hardware
 - Linux runs on PC hardware

 - ◊ the fact that it's older means that it's been tried, tested and it's reliable

 - ◊ was created as a tool for working programmers
 - main goal is to be a useful language
 - easy readability and writability

 - ◊ C initially became widely known as the development language of the UNIX operating system
 - virtually all new major operating systems are written in C and/or C++

 - ◊ C evolved from a previous programming language named B
 - uses many of the important concepts of B while adding data typing and other powerful features
 - B was a "typeless" language--every data item occupied one "word" in memory, and the burden of typing variables fell on the shoulders of the programmer
 - typeless: means that data didn't have types

 - ◊ C is available for most computers

 - ◊ C is also hardware independent
 - it means it's more portable
 - you can take the program and you can run it on different Operating Systems

- it means that it may work on many other systems
- ◊ by the late 1970s, C had evolved into what is now referred to as “traditional C”
- ◊ the rapid expansion of C working on many different hardware platforms led to many variations that were similar but often incompatible
 - a standard version of C was created (C89/C90, C99, C11)
- ◊ A program only written only in Standard C and without any hardware-dependent assumptions will run correctly on any platform with a standard C compiler
 - non-standard C programs may run only on a certain platform or with a particular compiler
 - there used to be many variations of C and we started standardizing in 1989
 - by standardizing we can now run it basically anywhere
- ◊ C89 is supported by (almost all) current C compilers
 - most C code being written today is based on it
 - this was the first standard
- ◊ C99 is a revised standard for the C programming language that refines and expands the capabilities of C
 - has not been widely adopted and not all popular C compilers support it
 - we'll be mostly focusing on this standard
- ◊ C11 is commonly referred as the current standard
 - some elements of the language as defined by C11 are optional
 - also possible that a C11 complier may not implement all of the language features mandated by the C11 standard
 - this class will base its examples and concepts off C11
- ◊ C has grown because people try it and like it.
- ◊ in the past decade or two, many have moved from C to languages such as C++, Objective C and Java
 - C is still an important language in its own right, as well a migration path to these others

5. Language Features

- **Overview:**

- C produces compact and efficient programs
- The main features of the C language are the following:

- ◊ **Efficient**
- ◊ **Portable**
- ◊ **Powerful and Flexible**
- ◊ **Programmer Oriented**

- **Efficiency:**

- ◊ C is an efficient language
 - takes advantage of the capabilities of current computers
 - C programs are compact and fast (similar to Assembly language programs)
 - C programmers can fine-tune their programs for maximum speed or most efficient use of memory

- **Portability:**

- ◊ C is a portable languages
- ◊ C programs written on one system can be run on other systems with little or no modification
- ◊ a leader in portability
- ◊ compilers are available for many computer architectures
- ◊ Linux/UNIX systems usually come with a C compiler as part of the package
 - compilers are available for personal computers
- ◊ there's a good chance that you can get a C compiler for whatever device you are using

- **Powerful:**

- ◊ the UNIX/Linux kernel is written in C
 - kernel: the brain of the operating system
 - the brain and knowledge of the operating system goes
 - it typically contains interfaces to the hardware
- ◊ many compilers and interpreters for other languages (FORTRAN, Perl, Python, Pascal, LISP, Logo and BASIC) have been written in C
- ◊ C programs have been used for solving physics and engineering problems and even for animating special effects for movies.

- **Flexibility:**

- ◊ C is flexible
 - used for developing just about everything you can imagine by way of a computer program

- accounting applications to word processing and from games to operating systems
- it is the basis for more advanced languages, such as C++
- ◊ however there is a tradeoff between flexibility and performance
 - flexibility sometimes causes more bugs
 - it also cause developers not understand certain concepts because of complexity.
- ◊ it is also used for developing mobile phone apps in the form of Objective C
- ◊ C is easy to learn because of its compactness
 - is an ideal first language to learn if you want to be a programmer
 - you will acquire sufficient knowledge for practical application development quickly and easily

- **Programmer Oriented:**

- ◊ C fulfills the needs of programmers
 - gives you access to hardware
 - enables you to manipulate individual bits in memory
- ◊ C contains a large selection of operators which allows you to express yourself succinctly
- ◊ C is less strict than most languages limiting what you can do
 - can be both an advantage and a danger:
 - advantage is that many tasks (such as converting forms of data) are easier in C
 - danger is that you can make mistakes (pointers) that are impossible in some languages
 - C gives you more freedom, but it also puts more responsibility on you.
- ◊ C implementations have a large library of useful C functions
 - deal with common needs of most programmers

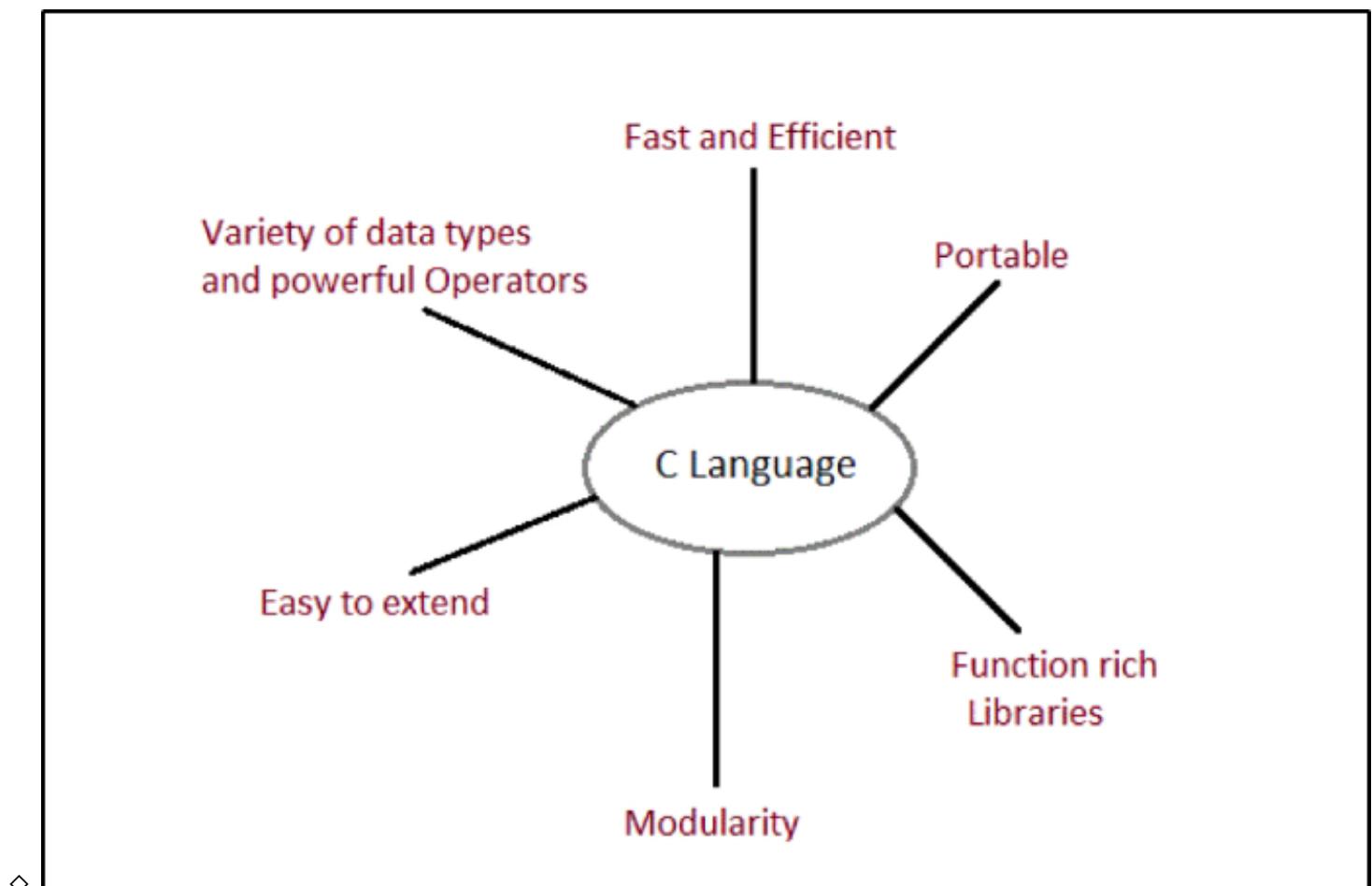
- Other features:

- ◊ provides low level features that are generally provided by the lower lever languages
- ◊ programs can be manipulated using bits
 - ability to manage memory representation at bit level
 - provides a wide variety of bit manipulation operators
- ◊ pointers play a big role in C
 - direct access to memory
 - supports efficient use of pointers

- Disadvantages:

- ◊ flexibility and freedom also requires added responsibility
 - use of pointers is problematic and abused
 - you can make programming errors that are difficult to trace
- ◊ sometimes because of its wealth of operators and its conciseness, it makes the language difficult to read and follow
 - there is an opportunity to write obscure code

- Features of C:



6. Creating a C Program

- There are four fundamental tasks in the creation of any C program
 - ◊ Editing
 - ◊ Compiling
 - ◊ Linking
 - ◊ Executing
 - These tasks will become second nature to you because you'll be doing it so often (everytime you write a C program)
 - The processes of editing, compiling, linking and executing are essentially the same for developing programs in any environment and with any compiled language
- **Editing:**
- ◊ is the process of creating and modifying your C source code
 - source code is inside a file and contains the program instructions you write
 - choose a wise name for your base file name (all source files end in the .c extension)
 - we will use an IDE (code blocks) for this class, but you can use any editor (notepad, etc.) to create your source code
- **Compilation:**
- ◊ compilation is a two stage process
 - the first stage is called preprocessing phase, during which your code may be modified or added to
 - the second stage is the actual compilation that generates the object code
 - ◊ the compiler examines each program statement and checks it to ensure that it conforms to the syntax and semantics of the language
 - can also recognize structural errors (dead code → code that's not being used)
 - does not find logic errors
 - typical errors reported might be due to an expression that has unbalanced parentheses (syntax error), or due to the use of a variable that is not "defined" (semantic error)
 - ◊ after all the errors are fixed, the compiler will then take each statement of the program and translate it into Assembly language
 - the compiler will then translate the Assembly language statements into actual machine instructions
 - the output from the compiler is known as object code and it is stored in files called object files (same name as source file with a .obj → Windows or .o → Linux or macOS extension)
 - ◊ The standard command to compile your C programs will be **cc** (or the GNU compiler, which is **.gcc**)

- **cc -c myprog.c**
- or
- **gcc -c myprog.c**
 - if you omit the **-c** flag, your program will automatically be linked as well

- **Linking:**

- ◊ after the program has been translated into object code, it is ready to be linked
 - the purpose of the linking phase is to get the program into a final form for execution on the computer
 - linking usually occurs automatically when compiling depending on what system you are on, but, can sometimes be a separate command
 - linking is getting all the dependancies (external libraries) in place and building an actual executable file
 - ◊ the linker combines the object modules generated by the compiler with additional libraries needed by the program to create the whole executable
 - also detects and reports errors
 - if part of the program is missing or a nonexistent library component is referenced.
 - ◊ program libraries support and extend the C language by providing routines to carry out operations that are not part of the language
 - input and output libraries, mathematical libraries, string manipulation libraries
 - ◊ a failure during the linking phase means that once again you have to go back and edit your source code
 - ◊ success will produce an executable file
 - in a Windows environment, the executable file will have an .exe extension
 - in UNIX/Linux, there will be no such extension (a.out by default)
 - many IDEs have a build option, which will compile and link your program in a single operation to produce the executable
 - ◊ a program of any significant size will consist of several source code files
 - each source code file needs the compiler to generate the object file that needs to be linked
 - ◊ the program is much easier to manage by breaking it up into a number of smaller source code files
 - it is cohesive and makes the development and maintenance of the program a lot easier
 - the set of source code files that make up the program will usually be integrated under a project name, which is used to refer to the whole program

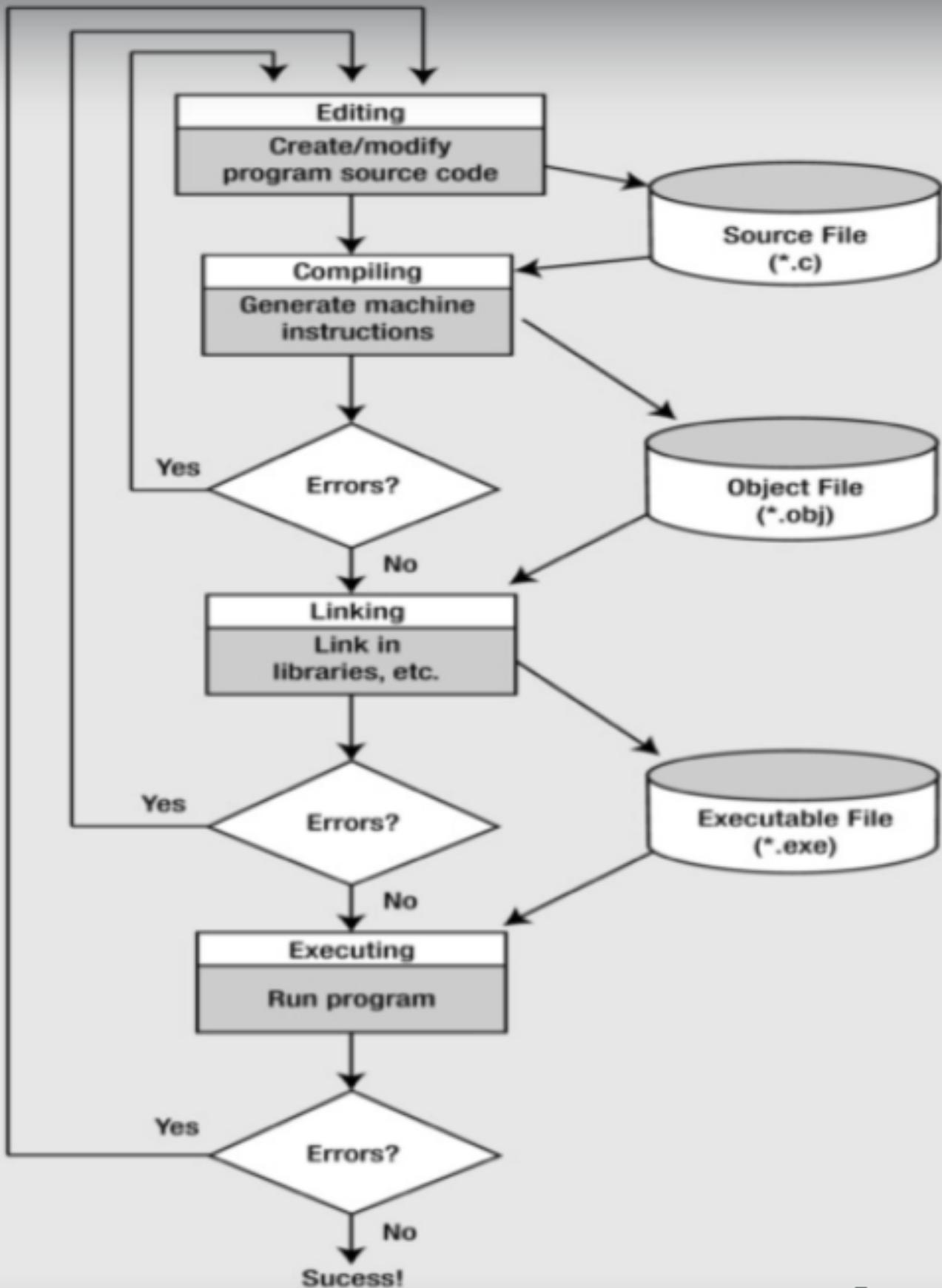
- **Executing:**

- ◊ in most IDEs, you'll find an appropriate menu command that allows you to run or execute your compiled program
 - otherwise double click the .exe file or type a.out on the console in Linux manually (you may have to add +x permissions to the file)
 - ◊ the execution stage is where you run your program

- each of the statements of the program is sequentially executed in turn
 - if the program requests any data from the user the program temporarily suspends its execution so that the input can be entered
- ◊ results that are displayed by the program (output) appear in a window called the console
- ◊ this stage can also generate a wide variety of error conditions
 - producing the wrong output
 - just sitting there and doing nothing
 - crashing your computer
- ◊ if the program does not perform the intended functionality then it will be necessary to go back and reanalyze the program's logic
 - this is known as the debugging phase, correct all the known problems or bugs from the program
 - will need to make changes to the original source code program
 - the entire process of compiling, linking and executing the program must be repeated until the desired results are obtained.

- Flow map of a C program:

- ◊



Section 2: Installing Required Software

7. Overview

- Windows required installations:
 - ◊ C compiler
 - Cygwin
 - this compiler allows you to write more advanced C applications, using sockets, threads and some more advanced code etc.
 - ◊ IDEs
 - Code::Blocks
 - VScode
 - the course is not dependant on any IDE, **you can use any IDE you want.**

8. Installing the C Compiler (Windows)

- Overview:
 - ◊ you need to install the GNU gcc compiler, make, and gdb debugger from cygwin.com
 - this simulates a UNIX environment
 - this allows us to write C code in the 89 standard
- Download either the 32 or 64 bit version of the setup file from:
 - ◊ <https://cygwin.com/install.html>
- Select everything by default when installing
- When you get to the packages section, we're only gonna need to install 3 packages:
 - ◊ gcc-core (compiler)
 - ◊ gdb
 - ◊ make (utility)
 - to do this change **view** to **full**
 - then search for:
 - gcc-core
 - ⇒ it should look something like this, make sure the version field doesn't say "skip"
 - ⇒ if the **bin?** checkbox shows up for you, make sure you check it
 - gdb
 - ⇒ it should look something like this, make sure the version field doesn't say "skip"
 - make
 - ⇒ it should look something like this, make sure the version field doesn't say "skip"
 - ◊ now hit "next" and the packages will be downloaded and installed
 - then hit finish
 - now your C installation is done for your compiler
 - Next step is check the PATH is set up correctly, this allows us to use the gcc binary anywhere in any directory
 - ◊ go to the Windows File Explorer
 - then go to **C:/cygwin64**
 - now look for the **bin** folder and copy the path to this directory
 - now on the file explorer click on "**This PC**", right click and select **properties**
 - ⇒ this will bring the System settings window, on the left click on **Advanced System Settings**

- now at the bottom on the new window click on **Environment Variables...**
 - ◊ on the bottom it says **System Variables**
 - select the one that says **Path** and click on the **edit** button
 - click on **new** on the right side and then paste the location of the bin folder
 - now click **ok** and exit
- Once the previous step is done we can verify its installation by bringing up the DOS prompt:
 - ◊ an enter the following command:
 - `cygcheck -c cygwin`
 - and a message that tells us that cygwin was installed fine should pop up
 - ◊ next we're gonna check that the C compiler was installed:
 - enter the following command:
 - `gcc --version`
 - ◊ after this we're done

10. Installing Code::Blocks (Windows)

- To set the path for the compiler that we're gonna use, we need to:
 - ◊ copy the file path where the Cygwin folder's at
 - in our case:
 - **C:/cygwin64**
 - ◊ also set the C++ compiler to where **gcc.exe** is at (in the bin subfolder on the **C:/cygwin64** directory)

Section 3: Starting to write code

18. Exploring The Visual Studio Code Environment

- **Keyboard Shortcuts** (these also work on **Atom**):

- ◊ **Ctrl + Shift + p** → view keyboard shortcuts
- ◊ **Ctrl + p** → will let you navigate to any file
- ◊ **Ctrl + shift + tab** → will let you navigate through the last set of files opened
- ◊ **Ctrl + shift + o** → will let you navigate to a specific symbol in a file (**different command in Atom**)
- ◊ **Ctrl + g** → will let you navigate to a specific line in a file

19. Creating a Workspace and Configuring the Compiler in Visual Studio Code

- How to create a workspace in Visual Studio Code
- We have to link the compiler and Visual Studio Code in order for both to work together so we can run a C program
- First start the Windows cmd prompt
 - ◊ we're gonna create a workspace
 - **a workspace is an area where you wanna work for different C projects**
 - input (this is in whatever directory you want):
 - **mkdir projects**
 - ⇒ then:
 - **cd projects**
 - ◊ then:
 - **mkdir helloWorld**
 - then:
 - **cd helloWorld**
 - ⇒ then we're gonna start Visual Studio Code, when it starts it's gonna automatically create the workspace
 - to start VSCode from the cmd prompt use:
 - ◊ **code .**
 - Next we're gonna configure the compiler path and intellisense settings, then we're gonna launch a task for the build instructions and finally set the debugger settings
 - To configure the compiler path:
 - ◊ on VSCode go to:
 - **view**
 - **command palette** option
 - then type in:
 - ⇒ **c/c++ edit configurations UI**
 - here we're gonna set up mainly the compiler
 - ◊ here VSCode will try to automatically find it otherwise we'll have to input the path ourselves
 - on **Compiler Path**
 - this would be the path to our Cygwin environment that we installed previously

- now scroll down to **Intellisense Mode**
- ⇒ and make sure you select the right one:
- **windows-gcc-x64**
- ◊ after this you can exit out these options

- Now we're gonna create a **C_Cpp** properties file in your vscode folder
 - ◊ the IDE created a **.vscode** workspace folder
 - and inside that folder there's a **c_cpp_properties.json**
 - file with some code (settings) in it
 - here you'll see the compiler path and this has to be set correctly
 - after you're done you can close the window
- Our next step is going to be to configure some tasks
 - ◊ we want to basically set some settings for the build task
 - this will tell VSCode how to build and compile the program, it'll invoke the GCC compiler based on some source code
 - go to:
 - **View**
 - ⇒ **Command Palette**
 - then type in:
 - ◊ **task**
 - here choose:
 - **Configure Default Build Task**
 - then select:
 - ⇒ **Create task.json file from template**
 - then select:
 - ◊ **Others**
 - and this is gonna create a default tasks file, here we're gonna specify some arguments for compilation and also set the command dump.
 - on label type in the name of your application in this case it'd be:
 - `"label": "build hello world"`
 - ⇒ the **type** can stay as default
 - the **command** is gonna change and it's gonna say:
 - ◊ `"command": "gcc"`
 - then we're gonna set some arguments for the command:

```

"args": [
    "-g", //COMPILATION COMMAND
    "-o", //TO CREATE AN OBJECT FILE
    "helloworld", //NAME OF THE EXECUTABLE
    "helloWorld.c" //NAME OF SOURCE CODE
],

```

→ **-g** → compilation command for global
→ **-o** → to create an object file
→ helloworld → name of the executable
- then we're gonna specify our source files, if you have more than one source file here you'd specify them

→ helloWorld.c
- this file is gonna be changed everytime you create different source files, you're gonna have to create a new **tasks** file for each project

→ then we're gonna paste/add some extra stuff:

```
"group"  {  
    "kind": "build",  
    "isDefault": true  
}
```

- and now we can save the file
 - ◊ here we're telling VSCode how to build the program.

- Now let's configure our debug settings:
 - ◊ to do this we're gonna have to create a **launch** task
 - go to:
 - **Run**
 - then:
 - ⇒ **Add Configuration**
 - and then choose:
 - ◊ **C++ (GDB/LLDB)**
 - a new file is created
 - we have to specify a different number of options here
 - the main thing we have to change is the:
 - ⇒ **"program"** option, it should look like this:
 - `"program" "${workspaceFolder}/helloWorld.exe"`
 - ◊ we also have to change the **"stopAtEntry"** option:
 - `"stopAtEntry" true`
 - we also have to specify the debugger path:
 - `"miDebuggerPath" "C:/cygwin64/bin/gdb.exe"`
 - ⇒ the rest of the options can be left as default
 - now save and close the file

- One last thing we wanna do is make sure our shell is set up correctly (this only applies to

Windows)

◊ we wanna make sure we use the **bash shell** provided by cygwin

▪ go to:

- **View**

→ **Command palette**

⇒ type in:

• **settings**

◊ then click on:

- **Open Settings (JSON)**

- and here we're gonna specify our shell if it's not set:

→ `"terminal.integrated.shell.windows" "C:\\cygwin64\\bin\\bash.exe"`

⇒ after this close the IDE and open it again

• now go to:

◊ **View**

- **Terminal**

- and here you'll see the bash terminal pop up

→ here you can also compile and run your code using **gcc**

commands

• Now we're gonna create a source file to see if everything worked

◊ go to:

- **File**

- and create a **New File**

→ here we're gonna create a **helloWorld.c** file

⇒ paste in the following code

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

◊ this file has to be saved inside the workspace, save it in the **helloWorld** project

folder

▪ save it as **helloWorld.c**

- the file needs to have the same name as the name you provided in the **tasks.json** file, it can't be a different name, otherwise it won't compile.

• Now we're gonna build the program, hit:

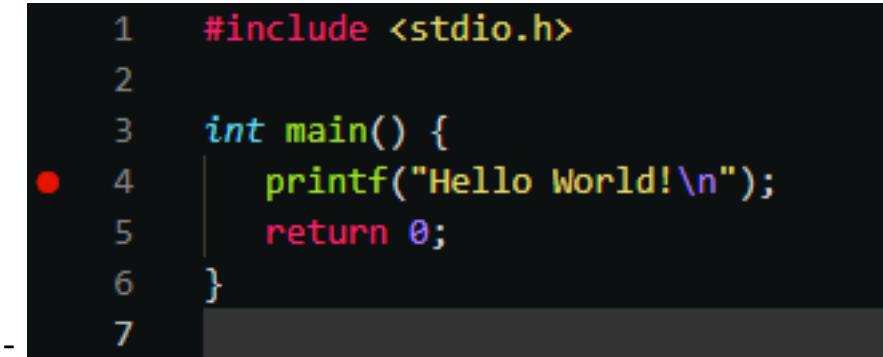
◊ **Ctrl + shift + b**

▪ that's gonna go ahead and compile our program

- we successfully compiled our program
 - it doesn't show us the program's output
 - ⇒ to show the output you're gonnna have to type in the name of the executable on the bash cmd line (along with the ./ characters):
 - **./helloWorld.exe**
 - ◊ and it should print the **Hello World** string

- To **debug**:

- ◊ in order to debug we first have to set a breakpoint because we want the program to stop
- we set a breakpoint on the ***printf()*** line (the red dot on the following image):



```

1 #include <stdio.h>
2
3 int main() {
4     printf("Hello World!\n");
5     return 0;
6 }
7

```

The screenshot shows a code editor with a dark theme. A C program is displayed. Line 4 contains the `printf` statement. A red circular breakpoint icon is positioned to the left of the line number 4. The code is numbered from 1 to 7.

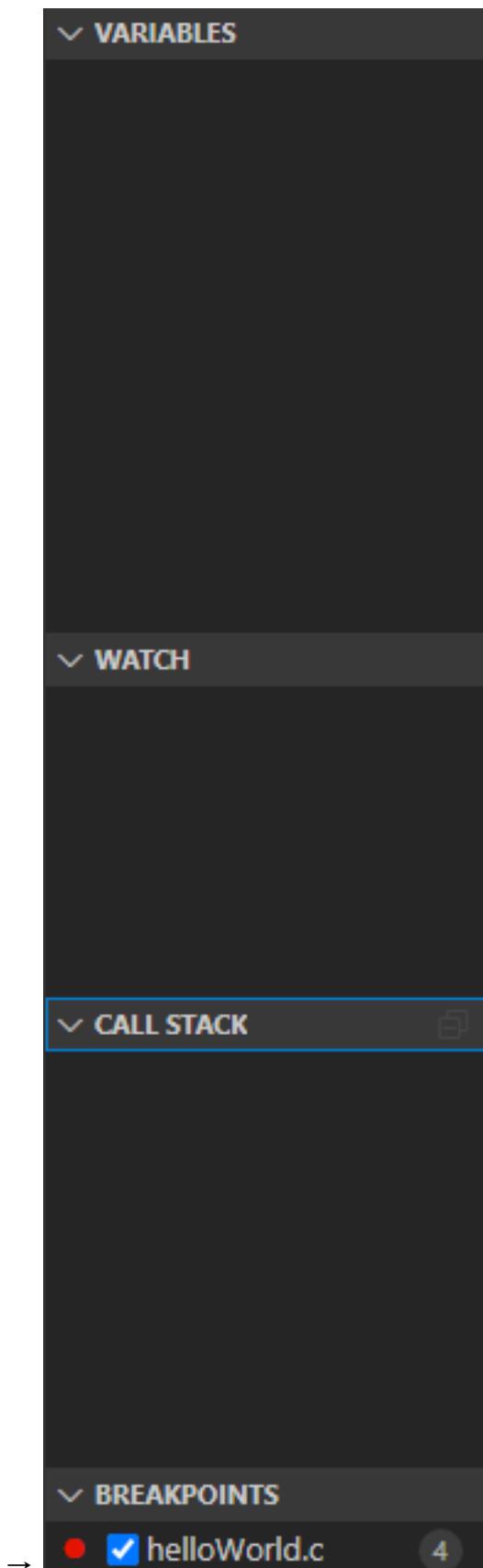
→ the breakpoint will stop the program on that line when it executes

⇒ we can execute **launch.json** (the debugging process) by hitting:

- **F5**

- ◊ now you can see that it's highlighted at the breakpoint that you set
 - and now you can run it one to the next line
 - and it comes on the debug view on the left hand size

→



- the debug view will have the **call stack** listed
 - along with the **breakpoints** that you set
 - ◊ also the **watch** window and the **variables**
 - (we'll talk about this later in the course)

- The **different debugger buttons**:

- ◊ **continue**:

- runs the program until the next breakpoint

- ◊  **step over:**

- will run to the next line

- ◊  **step into:**

- will go inside of a function

- ◊  **step out:**

- will jump out of the function

- Instead of doing all of this set up you can install a **VSCode** extension called:

- ◊ **coderunner**

- to run with **coderunner** hit:

- **Ctrl + alt + n**

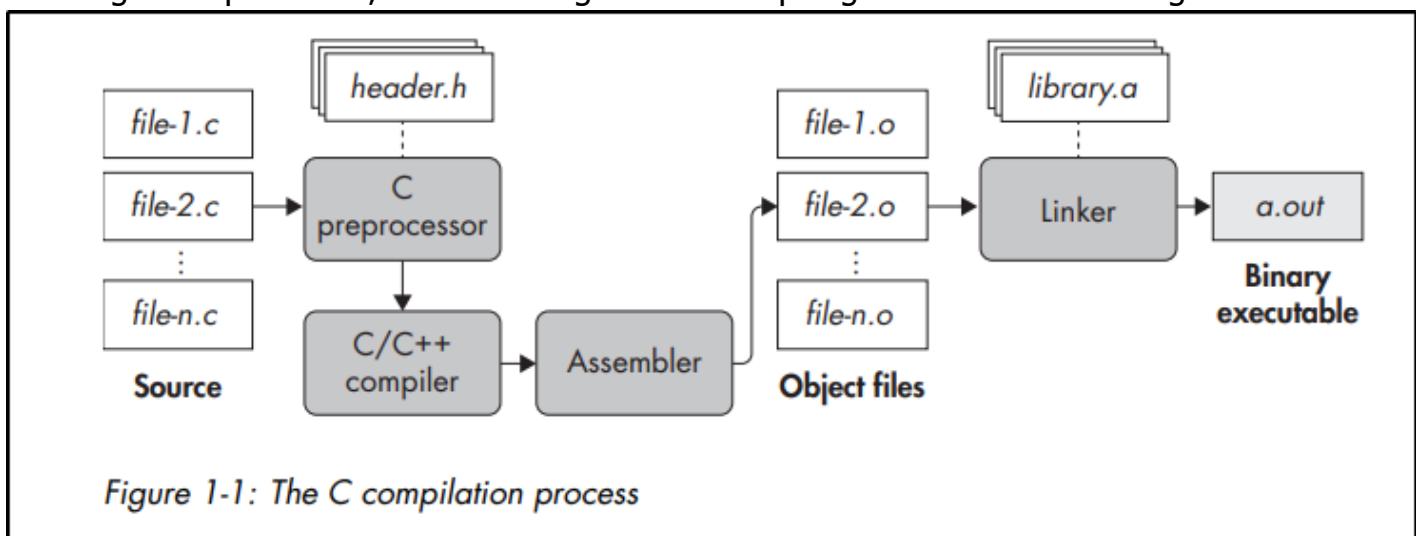
20. Creating and running your first C Program

- The purpose of this lecture is to introduce us to the build and compilation phases
- Our first program:

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

- When we **build** a program what happens is that it is:
 - ◊ **compiled**
 - ◊ **linked**
- After taking the input .c file, it is linked right after compiling the .o file that was generated.



- We can run and build the program in one single step.
- Build messages windows tell us if there were any errors during the build.

21. (Challenge) Writing a C program that displays your name

- This is gonna be our first challenge.
- We're not doing this one because it's for beginners.
- Code to solve the challenge:

```
#include <stdio.h>

int main() {
    printf("JFITECH\n");
    return 0;
}
```

Steps to solve the challenge

1. We got to edit the source code.
2. Compile and link the program.
3. Run the program!
4. Analyze the output to confirm it's correct.

23. Structure of a C Program

- C code structure:

```
#include <stdio.h>

int main() {
    printf("JFITECH\n");
    return 0;
}
```



- the two parentheses after **main()** on **line 3** signify that it's a function
 - the **main()** function is part of every C program
 - it's the **entry point to the program**
 - this is what gets involved when you run the program
 - everything that's inside the function will be executed
 - the **main()** is a special function that gets invoked when execute the program
 - whatever's inside the squiggly brackets **{ }** (lines 4 to 5) is referred to as a block of code
- C code programs are case sensitive
- all statements must end in a semicolon ;
 - lines 3 and 1 aren't statements
 - 3 is defining a function
 - 1 is including a library
- indentation helps improve quality of code
 - it also makes it more readable
 - it also helps us identify which lines are inside other blocks of code
- meaningful names also improve code readability
 - for example naming variables and functions

- The **int** before declaring the function name:

- means that it has to return a number
- when we talk about functions in further lectures we'll talk about how we can return data from a function
 - it basically means that you need to have a **return** statement in the function
 - this says that the **main()** is gonna return a number
 - we won't always see the **int** there
 - other times we'll see something like a **void** keyword there
 - this means that the function doesn't return any data
 - typically you'll see a compiler warning for this

◊ sometimes inside the parentheses on the function we'll see the keyword **void**:

- **int main(void)**

- this means that the **main** program doesn't accept any parameters or any data
 - void is another way to say **no data**
 - it's pretty much the same as with empty parentheses

- The **return 0** inside the function:

- ◊ it usually represents whether or not there was an error in the program
 - when you **return 0** it means that there was no error in the program
 - if you **return** an error code, let's say **return 44** that might represent an error in the program

- Because **main** is a **keyword** you can't declare any variables named **main**.

- ◊ this would give you an error

Section 4: Basic Concepts

24. Comments

- In this section we're gonna talk about some of the basic concepts in C

- **Comments:**

- ◊ they are used to document a program and enhance its readability
- ◊ they are ignored by the compiler
- ◊ a simple comment can save a significant amount of time otherwise wasted on having to re-understand the code

- **Syntax for comments in C:**

- ◊ there are two ways to add comments into a C program
 - this is very similar to many other languages
- ◊ **multi-line comments:**
 - open the comment → `/*`
 - close the comment → `*/`
 - all characters included between the opening `/*` and closing `*/` are treated as part of the comment.
- ◊ **single-line comments:**
 - `//`
 - any characters that follow these slashes up to the end of the line are ignored by the compiler.

- **Styling:**

- ◊

```
/*
 * Written by Jason Fedin
 * Copyright 2017
 */
```

You can also embellish comments to make them stand out:

```
*****
 * This is a very important comment      *
 * so please read this.                  *
*****/
```

You can add a comment at the end of a line of code

```
printf("Hello, nope!"); // This line displays a quotation
```

- Example:



```
/* This program adds two integer values  
and displays the results */  
  
#include <stdio.h>  
  
int main (void)  
{  
    // Declare variables  
    int value1, value2, sum;  
  
    // Assign values and calculate their sum  
    value1 = 50;  
    value2 = 25;  
    sum = value1 + value2;  
  
    // Display the result  
    printf ("The sum of %i and %i is %i\n", value1, value2, sum);  
  
    return 0;  
}
```



- **Readability is an advantage because it makes code easier to maintain**

- **Use of comments:**

- ◊ it is possible to insert so many comments into a program that the readability of the program is actually degraded instead of improved!
- ◊ you need to intelligently use comments
- ◊ it is a good idea to get into the habit of inserting comment statements into the program as the program is being written or typed in
 - this makes it easier to document the program while the particular program logic is still fresh in your mind
 - reap the benefits of the comments during the debug phase, when program logic errors are being isolated and debugged
 - comments can help you fix errors during the debugging phase.

- a comment can help you read through the program, but it can also help point the way to the source of the logic mistake
 - ◊ self documenting comments by using meaningful names
 - **by using meaningful names** in **variables** and **functions** we won't need to add comments on these.
 - very loosely coupled
 - highly cohesive
- Typically you can add comments at the top of the source file with standard information like:
 - ◊ author
 - ◊ purpose
 - ◊ description
 - ◊ date created

- We can use the following comment example at the start of the source file:

```
/*
 AUTHOR: JFITECH
 PURPOSE: THIS PROGRAM PRINTS OUT A HELLO WORLD TO THE SCREEN
 DATE: DECEMBER 15TH, 2021

 */
```

- an then we can provide comments throughout the code, either single-line comments or multi-line comments.

25. The preprocessor

- The preprocessor is being used in every single program
 - ◊ this is the **#include** directive
- **Overview:**
 - ◊ this is another unique feature of the C language that is not found in many other higher-level programming languages.
 - ◊ allows for programs to be easier to develop, read and modify; and easier to port to different computer systems.
 - ◊ part of the C compilation process that recognizes special statements
 - analyzes these statements before analysis of the C program itself takes place.
 - an instruction to your compiler to do something before compiling the source code.
 - could be anywhere in your code.
 - the word **pre-processor** implies that something goes on before you process the code.
 - it analyzes certain statements in your code before it analyzes the program itself.
 - because you do these things before you compile you can optimize things and you can make things a lot easier.
 - ◊ very often you'll have these pre-processor commands at the top of your source code, but they can be anywhere throughout your code.
 - ◊ one of the big ones we're gonna talk about is the include.
 - ◊ pre-processor statements are identified by the presence of a pound sign, **#**, which must be the first non-space character on the line.
 - **pre-processor commands** are also known as **pre-processor directives**.
 - ◊ our first challenge included a pre-processor directive, specifically the **#include** directive.
 - ◊ we will utilize the pre-processor to:
 - create our own **constants** and **macros** with the **#define** statement.
 - you can create **constants** in the **pre-processor** and this optimizes your code, makes it more efficient and runs faster.
 - a **macro** is another **pre-processor directive**.
 - the way that you can create **macros** is to use the **#define** statement.
 - build your own library files with the **#include** statement.
 - a library is just reusing existing code.
 - the C programming language comes with its own standard library that does many things for you, like:

- display output to the console.
- read and write files.
- handle strings
- similar to importing Java APIs
- make more powerful programs with the conditional **#ifdef**, **#endif**, **#else** and **#ifndef** (if not defined) statements.
- we will talk about the above in further lectures.

26. The #include statement

- Here we'll see what the **#include** statement does.
- We're gonna see this in every C program.

- **Overview:**

- ◊ the **#include** statement is a pre-processor directive.
- ◊ the example in our previous program was:
 - **#include <stdio.h>**
- ◊ it is not strictly part of the program, however, the program won't work without it.
- ◊ it's just telling the compiler before it runs: "hey we're doing something here with this **#include**".
- ◊ if you don't have it in there the compiler is going to throw errors.
- ◊ the symbol **#** indicates this is a pre-processing directive
 - an instruction to your compiler to do something before the compiling the source code.
 - many pre-processing directives.
 - they are usually at the beginning of the program source file, but they can be anywhere.
 - similar to the **import statement** in Java.
- ◊ in the above example, the compiler is instructed to "include" in your program the contents of the file with the name **stdio.h**
 - called a header file because it's usually included at the head of a program source file.
 - **.h** extension.
 - the include directive will have files that end in **.h**
 - the pre-processor directive basically says "I want some of the contents inside that file".

- **Header Files:**

- ◊ header files define information about some of the functions that are provided by that file
 - **stdio.h** is the standard C library header and provides functionality for displaying output, among many other things
 - we need to include this file in a program when using the **printf()** function from the standard library.
 - **stdio.h** contains the information that the compiler needs to understand what **printf()** means, as well as other functions that deal with input and output.

- **stdio** is short for standard input/output.

◊ header files specify information that the compiler uses to integrate any predefined functions within a program.

◊ you will be creating your own header files for use with your programs.

- header files allow for software reuse.

- basically it's a software reuse technique.

• Syntax:

◊ header files are case sensitive on some systems, so you should always write them in lowercase.

◊ there's two ways to **#include** header files in a program:

- using angle brackets → **#include <stdio.h>**

- tells the pre-processor to look for the file in one or more standard system directories.

- using double quotes → **#include "stdio.h"**

- tells the pre-processor to first look in the current directory.

- **this usually means that it's a user defined header file.**

◊ every C compiler that conforms to the **C11 standard** will have a set of standard header files supplied with it

- these are just header files that represent libraries.

◊ you should use **#ifndef** and **#define** to protect against multiple inclusions.

▪ this is a common practice

▪ this is so that you don't include the file more than once.

▪ this is also for efficiency.

▪ this is so the preprocessor doesn't have to do many things that it would do.

◊ example of a header file and its contents:

```
// some header

//typedefs
typedef struct names_st names;

//function prototypes
void get_names(names *);
void show_names(const names*);
char * s_gets(char * st, int n);
```

- it mainly has **function prototypes** in it, but it can have **constants**, it can also have **typedefs**.

- it contains many different things.

- ◊ header files include many different things:
 - **#define** directives.
 - **constants**
 - **structure declarations.**
 - **typedef statements.**
 - **function prototypes.**
 - these are the important ones.
 - they're telling you how functions are defined so the compiler can generate the code it needs to.
- ◊ executable code usually goes into a source code file, not a header file.
 - there are some exceptions though.

27. Displaying Output

- In this lecture we're gonna learn how to display output to the screen.
- **printf()** is a standard library function:
 - ◊ it outputs information to the command line (the standard output stream, which is the command line by default).
 - ◊ the information displayed is based on what appears between the parentheses that immediately follow the function name (**printf**)
 - ◊ also notice that this line does end with a semicolon.
 - the semicolon tells the compiler "we're done, we're printing this information."
 - ◊ this function can get more complex using format specifiers (we're gonna discuss this later).
- **printf()** function:
 - ◊ probably the most common function used in C.
 - because you often need to display output.
 - it'll be function that we use the most in this class.
 - ◊ provides an easy and convenient means to display program results.
 - ◊ not only can simple phrases be displayed, but the values of variables and the results of computations can also be displayed.
 - used for debugging (we're gonna use it as a debugging technique so we can see what's going on in our program as it's running).
 - you can see where you are in your code as you're executing.
 - ◊ to use it include everything that you want to print out inside double quotes " ".

```
printf("What we want to print out goes in here\n");
```

28. Reading input from the terminal

- Reading input from the keyboard.

- **Overview:**

- ◊ The C library contains several input functions, and **scanf()** is the most general of them
 - can read a variety of formats.
- ◊ **scanf()**
 - reads the input from the standard input stream **stdin** and scans that input according to the **format** provided.
 - format can be a simple constant string, but you can specify **%s**, **%d**, **%c**, **%f**, etc., to read strings, integer, character or floats.
 - it can also be used for reading from files.
 - here the input stream is a file.
 - using **scanf()** is gonna read from **standardin (stdin)**, then after how you wanna store the data (after you enter the input), it's gonna store the text that you enter and convert it to the format that you provided.
 - basically it's gonna use format specifiers.
 - ◊ if the **stdin** is input from the keyboard then text is read in because the keys generate text characters: letters, digits and punctuation:
 - when you enter the integer 2014, you type the characters **2 0 1** and **4**
 - if you want to store that as a numerical value rather than as a string, your program has to convert the string character-by-character to a numerical value and this is the job of the **scanf()** function.

- **scanf()**

- ◊ like **printf()**, **scanf()** uses a control string followed by a list of arguments:
 - control string indicates the destination data types for the input stream of characters (**format specifiers** → **%d**, **%s**, **%f** etc.)
 - it's different from **printf()** in certain arguments that it takes.
 - ◊ the **printf()** uses variable names, constants and expressions (something that returns a value, the most common type are arithmetic expressions) as its argument list.
 - inside the double quotes you use format specifiers.
 - ◊ the **scanf()** function uses pointers to variables:
 - you do not have to know anything about pointers to use the function.
 - it's different because instead of displaying variables, it's gonna use variables that are pointed to.

- ◊ Remember these 3 rules about the **scanf()** function:

1. it returns the number of items that it successfully reads.
 - if it reads one variable, two variables, three variables it'll return that.
2. if you use **scanf()** to read a value for one of the basic variable types that we're gonna discuss (such as **int**, **float**, **double**), precede the variable name with an **&** (**ampersand is address**, because **scanf() takes a pointer, you have to pass in the address**).
3. if you use **scanf()** to read a string into a character array, don't use **&**.

- ◊ the **scanf()** function uses whitespace (newlines, tabs and spaces) to decide how to divide the input into separate fields.
 - if it has multiple things inside the double quotes it'll read multiple inputs and the inputs are gonna be divided by new lines.
- ◊ **scanf()** is the inverse of **printf()**, which converts integers, floating-point numbers, characters and C strings to text that is to be displayed on screen.

- **scanf()** example code:

```
#include <stdio.h>

int main() {
    char str[100]; //VARIABLE
    int i; //VARIABLE

    printf("Enter a value: "); //DISPLAY OUTPUT TO THE SCREEN
    scanf("%s %d", str, &i); //FORMAT SPECIFIERS TO READ IN THE INPUT

    printf("\nYou entered: %s %d", str, i); //

    return 0;
}
```

- ◊ at the top we're creating two variables:
 - **char str[100];** → character string variable
 - **int i;** → int variable
- then we display output to the screen:
 - **printf("Enter a value: ");**
 - **scanf("%s %d", str, &i)** has format specifiers to read in the input, so you have to specify the data that you have to read in:
 - **%f %d** → means that we're **reading in a string**, then we're reading and **int**.

- notice that the **str** doesn't have the **ampersand &**.
- before the **i** we do have an **ampersand &** because we're **reading in an integer**.
- **printf("\nYou entered: %s %d", str, i);**
- displays output.
- it shows the **string** and the **integer** that the user entered.

• **scanf()**:

- ◊ when a program uses **scanf()** to gather input data from the keyboard, it waits for you to input some text:
 - when you enter some **text** and press the **enter** key, the program proceeds and reads the input.
- ◊ remember, **scanf()** expects input in the same format as you provided **%s** and **%d**:
 - you have to provide valid inputs like:
 - a **string** followed by a **space** followed by an **integer**.
→ basically a word followed by a number.

• **scanf()** example code in IDE:

```
/*
 AUTHOR: JFITECH
 DATE: DECEMBER 16TH, 2021
 PURPOSE: LEARN HOW THE printf() AND scanf()
 FUNCTIONS WORK TOGETHER

*/
#include <stdio.h>

int main() {
    char str[100]; //STRING ARRAY OF 100
    int i;

    printf("Enter a value: "); //WE ASK THE USER FOR
INPUT

    scanf("%d", &i); /* WE READ THE USER INPUT.
    %d IS AN INT.
    THEN WE STORE THAT DATA IN A
VARIABLE, BECAUSE WE WANNA LOOK AT IT, THIS IS GONNA
BE i.
    WE HAVE TO SPECIFY THE AMPERSAND &
BEFORE THE i BECAUSE IT'S THE ADDRESS,
    WE'RE BASICALLY STORING THAT VALUE
IN A MEMORY ADDRESS. */

    printf("\nYou entered: %d\n", i); // %d DISPLAYS
THE VALUE OF THE VARIABLE.

    return 0;
}
```

- ◊ then we add some extra stuff:

```

/*
AUTHOR: JFITECH
DATE: DECEMBER 16TH, 2021
PURPOSE: LEARN HOW THE printf() AND scanf()
FUNCTIONS WORK TOGETHER

*/
#include <stdio.h>

int main() {
    char str[100]; //STRING ARRAY OF 100
    int i;

    printf("Enter a value: "); //WE ASK THE USER FOR
INPUT

    scanf("%d %s", &i, str); /* WE READ THE USER
INPUT.
        %d IS AN INT, %s IS A STRING
        THEN WE STORE THAT DATA IN A
VARIABLE, BECAUSE WE WANNA LOOK AT IT, THIS IS GONNA
BE i.
        WE HAVE TO SPECIFY THE AMPERSAND &
BEFORE THE i BECAUSE IT'S THE ADDRESS,
        WE'RE BASICALLY STORING THAT VALUE
IN A MEMORY ADDRESS.
        str IS USED TO STORE */

    printf("\nYou entered: %d and %s\n", i, str); //%
%d DISPLAYS THE VALUE OF THE VARIABLE.

    return 0;
}

```

- here on the **scanf()** function we're gonna read **two inputs separated by a space.**

- **scanf() how to read in a double** example code in IDE:

```

int main() {
    double x;

    scanf("%lf", &x); //%lf IS THE FORMAT SPECIFIER
USED FOR DOUBLE VARIABLE TYPE.

    return 0;
}

```

- **scanf() alternatives:**

- ◊ Sometimes you'll notice that **scanf()** doesn't work that well:
 - you can instead use **fgets()** for other things.
- ◊ the main issue with **scanf()** is that you only read in until you see a space:
 - so if you try to use **scanf()** and try to call it multiple times in a function with other combinations, it may not work very well because what's happening is:

- when you hit the **return** key on the keyboard, it's not gonna read the return key that you just pressed, because it only reads up into a word.

- so it's not gonna get the line feed, so the next time you use **scanf()**, it's gonna skip that line and it's gonna read in the line feed.

- you'll see this issue if you write multiple **scanf()** one after another in a single program:

- you'll notice that the other **scanf()** aren't executing because they're reading in the line feed.

- ◊ another workaround is to use:

- **getchar()**

- after using a **scanf()**

- then after the **getchar()**:

- you can use another **scanf()**

Section 5: Variables and Data Types

29. Overview

- **Overview:**

- ◊ Remember that a program needs to store the instructions of its program and the data that it acts upon while your computer is executing that program.
 - this information is stored in memory (RAM).
 - RAM's contents are lost when the computer is turned off or when the program exits
 - Hard drive store persistent data.
- ◊ You can think of RAM as an ordered sequence of boxes:
 - the box is full when it represents 1 or the box is empty when it represents 0.
 - each box represent one binary digit, either 0 or 1 (true and false).
 - each box is called a bit.
- ◊ **Bits in memory are grouped into sets of eight (bytes):**
 - **each byte has been labeled with a number (address).**
 - **the address of a byte uniquely references that byte in your computer's memory.**
 - if you have data stored in hardware you gotta know how to access it.
 - you gotta get **access** to that **address through** some **kind of label.**
- ◊ **Basically memory consists of a large number of bits that are in groups of eight (called bytes) and each byte has a unique address:**
 - your program will access that data usually through an identifier
 - but it can also use hex addresses
- ◊ the efficiency of your program will improve if you have more free memory and if your memory is faster, if the buses are faster then you can access memory quicker

- **Variables:**

- ◊ The true power of programs you create is their manipulation of data:
 - so, we need to understand the different data types you can use, as well as how to create and name variables
- ◊ Constants are types of data that do not change and retain their values throughout the life of the program:
 - as long as the program is running that data is always the same.
- ◊ Variables are data types that may change or be assigned values as the program runs.
- ◊ Variables and constants are stored in memory with an address.

- however it'd be very hard to access that memory and memorize all the hexadecimal addresses
 - so it's a lot easier to create variables that have meaningful names.
 - you can give variables names that represent the memory addresses.
 - ⇒ this way you can access and modify this data much easier
- ◊ For example assume you want to store two values, **10** and **20** in your program and at a later stage, you want to use these two values:
 - create variables with appropriate names.
 - store your values in those two variables.
 - retrieve and use the stored values from the variables.

• Naming Variables:

- ◊ The rule for naming variables in C is that:
 - all names must begin with a letter or underscore (_) can be followed by any combination of letters (upper or lower case), underscores or the digits 0 to 9
- **valid** variable names examples:
 - **Jason**
 - **myFlag**
 - **i**
 - **J5x7**
 - **my_data**
 - **_anotherVariable**
- **invalid** variable names examples:
 - **temp\$value** → \$ is not a valid character
 - **my flag** → embedded spaces are not permitted
 - **3JFi** → variable names cannot start with a number
 - **int** → int is a reserved word
- ◊ Use meaningful names when selecting variable names:
 - can dramatically increase the readability of a program and pay off in the debug and documentation phases.

• Data Types:

- ◊ Some types of data in programs are **numbers**, **letters** or **words**:
 - the data that's associated with the variable is referred to as a data type.
 - the computer needs a way to identify and use these different kinds.

- so it can do certain things
- this affects the size of the data in memory, how it's stored.

- ◊ A data type represents a type of the data which you can process using your program:
 - examples include:
 - **ints**
 - **floats**
 - **doubles**
 - etc.
 - also correspond to byte sizes on the platform of your program.
 - data types correspond to byte sizes depending on:
 - the operating system
 - the architecture of your program
 - ⇒ it all has to do with size and how stuff is stored in memory
 - the program needs to know:
 - ◊ do I need to allocate 8 bytes, 6 bytes, 4 bytes?
 - all of this depends on the data that is being stored
 - and the way that the program knows how to do that is you specify a data type

- ◊ **Primitive data types are types that are not objects and store all sorts of data:**
 - **in C all we have is primitive data types because we have no objects:**
 - **everything is a primitive data type in C.**

• Declaring Variables:

- ◊ Declaring a variable is when you specify the type of the variable followed by the variable name:
 - specifies to the compiler how a particular variable will be used by the program.
- ◊ for example, the keyword **int** is used to declare the basic integer variable
 - first comes **int**, then the **chosen name of the variable** and **ends with a semicolon**:
 - ***data-type variable-name;***
 - to declare more than one variable, you can declare each variable separately, or you can follow the **int** with a list of names in which each name is separated from the next by comma.
 - C requires that all program variables be declared before they are used in a program.
- ◊ example:


```
int x;
int x, y, z; //MULTIPLE VARIABLES ON THE SAME LINE
```

- the above example creates variables but does not provide values for them.

→ we can assign a variable a value by using the `=` operator:

⇒ `x = 112;`

• Initializing variables:

◊ To initialize a variable means to assign it a starting or initial value.

▪ it's important to initialize your variables

▪ because if they don't have any data, any values associated with them, it could cause problems in your program.

◊ This can be done as part of the declaration:

▪ follow the variable name with the assignments operator (`=`) and the value you want the variable to have.

◊ for example:

```
int x = 21;

int y = 32, x = 14;

int x, z = 94; //VALID, BUT POOR
FORM
```

▪ - in the last line only **z** is initialized:

→ it is best to avoid putting initialized and noninitialized variables in the same declaration statement.

• Example on IDE:

```
/*
    AUTHOR: JFITECH
    DATE: DECEMBER 16TH, 2021
    PURPOSE: VARIABLES DECLARATIONS

*/
#include <stdio.h>

int main() {
    int julz = 5;

    return 0;
}
```

◊ another example is:



- you could also assign values to your variables later in the program

30. Basic Data Types

- Data type are important because you have to provide this when you declare a variable:
 - ◊ they tell the actual compiler how much memory you wanna create:
 - it's specific to size.
 - the compiler needs this information.
- Let's talk about the ones supported by the C language.
- **Overview:**
 - ◊ We understand that C supports many different types of variables and each type of variable is used for storing kind of data:
 - types that store integers.
 - types that store nonintegral numerical values:
 - things like scientific notation
 - booleans
 - types that store characters:
 - single characters
 - many characters
 - ◊ Some examples of basic data types in C are (remember these data type are special keywords):
 - **int**
 - when you say **int** in a C program, the compiler is gonna know that this is a data type.
 - **float**
 - **double**
 - **char**
 - **_Bool** → boolean
 - these previous ones are the main ones you're gonna use
 - there's also different variations off of these depending on the type of data that you want to store and how big the data is
 - ◊ The difference between the types is in the amount of memory they occupy and the range of values they can hold:
 - the amount of storage that is allocated to store a particular type of data.
 - the amount of storage that is allocated to store a particular type of data is important to the compiler.
 - the memory and the amount is gonna determine the range of the values that can be stored.
 - depends on the computer that you are running (machine-dependant)
 - computer architecture
 - an **integer** might take up **32 bits** on your computer, or perhaps it might be stored in **64**.

- **int:**

- ◊ A variable of type **int** can be used to contain **integral values only** (values that do not contain decimal places).
- ◊ A **minus sign** preceding the data type and variable indicates that the value is negative
- ◊ The type **int** is a **signed integer**:
 - it must be an integer and it **can be positive, negative or zero**.
 - this is the reason why you can put a minus in front of it
- ◊ If an integer is **preceded** by a **zero** and the letter **x** (either lowercase or uppercase), the value is taken as being expressed in **hexadecimal (base 16)** notation:
 - `int rgbColor = 0xFFEFO;`
 - the previous hex value can be referred to as a **constant**.
 - an **int** can store different representations of numbers.
- ◊ The values **158**, **-10** and **0** are all valid examples of integer constants:
 - no embedded spaces are permitted between the digits.
 - values larger than **999** cannot be expressed using commas (**12,000** must be written as **12000**)

- **float:**

- ◊ A variable to be of type **float** can be used for storing **floating-point numbers (values containing decimal places)**.
- ◊ The values **3.**, **125.8** and **-0.0001** are all valid examples of floating-point constants that can be assigned to a variable.
- ◊ Floating point constants can also be expressed in scientific notation:
 - **1.7e4** is a floating point value expressed in this notation and represents the value → **1.7 * 10⁴**

- **double:**

- ◊ Very similar to the **float**.
- ◊ The **double** type is the same as type **float**, only with roughly twice the precision:

- used whenever the range provided by a **float** variable is not sufficient.
 - can store twice as many significant digits.
 - most computers represent **double** values using **64 bits (8 bytes)**.
 - it collects more data in memory.
- ◊ You wanna use a **double** whenever the **float** is not sufficient, it's too small.
- ◊ Say a **float** stores (depending on the architecture) **8 bytes**
- this means it can store values from:
→ **-32000 to 32000**
 - if you try to represent that same data in memory using a **double**, you can then store even higher numbers:
 - **millions and millions and trillions** much bigger than a float.
- ◊ All floating **point constants** are taken as **double** values by the C compiler.
- this is by default.
- ◊ To explicitly express a **float** constant, append either by an **f** or an **F** to the end of the number:
- **12.5f**

- **_Bool:**

- ◊ The **_Bool** data type can be used to store just the values **0** or **1**:
 - used for indicating an **on/off, yes/no** or **true/false** situation (**binary choices**).
- ◊ **_Bool** variables are used in programs that need to indicate a **Boolean** condition:
- a variable of this type might be used to indicate whether all data has been read from a file.
- ◊ **0 is used to indicate a false value.**
- ◊ **1 indicates a true.**
- ◊ The **_Bool** type also usually has been alias to other names, so you don't have to always use **_Bool:**
- this depends on the compiler you're using.
- Defined on the **C 89** standard:
- **_Bool**
- Defined on the **C 99** standard:

- it allows you to use the **true** or **false** values in the **bool** data type:

→ in order to use this we have to do the following:

```
#include <stdbool.h>

int main(){
    bool myBoolean = true;

}
```

- the **#include <stdbool.h>** is a macro that allows you to use different data types, the traditional **bool** data type.

- now the **bool** keyword is gonna be recognized as a **data type**.
-

- **C code example:**

```
/*
AUTHOR: JFITECH
DATE: DECEMBER 21ST, 2021
PURPOSE: DATA TYPES

*/

#include <stdio.h>
#include <stdbool.h>

int main(void) {
    int integerVar = 100;
    float floatingVar = 331.79;
    double doubleVar = 8.44e+11; //THIS LARGE NUMBER WOULDN'T BE ABLE TO BE
STORED IN AN float VARIABLE

    _Bool boolVar = 0;
    _Bool boolVar2 = false; //ANOTHER WAY TO DECLARE A _Bool VARIABLE

    return 0;
}
```

- **Other Data Types:**

- ◊ the **int** type will probably meet most of your integer needs when beginning in C.
- ◊ C is a flexible language.
- ◊ However, C offers **many other integer types**:
 - gives the programmer the option of **matching a type to a particular use case**.
 - **integer** types vary in the **range of values offered** and in **whether negative numbers**

can be used.

◇ C offers **three adjective keywords** to **modify** the basic **integer** type (**can also be used by itself**):

- **short**
- **long**
- **unsigned**

▪ can be used to have more specified versions of **int**, **float** and **double** data types.

◇ This is to be more precise and making the program more efficient:

▪ for the most part you'll just use:

- **int**
- **float**
- **double**

▪ but as you get more advanced in C you're gonna start using **short**, **long** and **unsigned**.

◇ The types **short int** or **short** may use less storage than **int**, thus saving space when only small numbers are needed:

▪ can be used when the program needs a lot of memory and the amount of available memory is limited.

◇ The type **long int** or **long**, may use more storage than **int**, thus enabling you to express larger values.

▪ uses more memory.
▪ larger data.

◇ The type **long long int** or **long long** may use more storage than **long**:

▪ a **constant** value of type **long** is formed by optionally appending the letter **L (upper or lowercase)** onto the end of an integer constant:

▪ `long int numberOfPoints = 131071100L;`

→ you can also apply suffixes at the end **short** data types.

◇ Type specifiers can also be applied to **double** data types:

▪ `long double US_deficit_2017;`

◇ A **long double** constant is written as a floating constant with the letter **I** or **L** immediately following:

▪ **1.234e+7L**

◇ The type **unsigned int** or **unsigned** is used for variables that **have only non-negative values** (positive):

▪ `unsigned int counter;`

- the accuracy of the integer variable is extended (only allows positive values).
 - this is how it works in theory, but some compilers don't enforce it.
- ◊ The keyword **signed** can be used with any of the signed types to make your intent explicit:
 - **signed** means that it can be **negative** and **positive**:
 - can be applied to **int**, **float** and **double** data types.
 - **short**, **short int**, **signed short** and **signed short int** are all names for the same type.

31. Enums and Chars

- We're gonna talk about a couple more of data types.

- **Enums:**

- ◊ A data type that allows a programmer to define a variable and specify the valid values that could be stored into that variable:

- can create a variable named "**myColor**" and it can only contain one of the primary colors: **yellow**, **blue** or **red**, and no other values.

- ◊ You first have to define the **enum type** and give it a name:

- initiated by the keyword **enum**.
 - then the **name** of the **enumerated data**.

- then list of identifiers (enclosed in a set of curly braces) that define the permissible values that can be assigned to the type.

- ◊ code example:

- `enum primaryColor {yellow, blue, red};`

- ◊ Variables declared to be of this data type can be assigned the values **yellow**, **red** and **blue** inside the program and no other values.

- ◊ To declare a variable to be of type **enum primaryColor**:

- use the keyword **enum**.
 - followed by the enumerated type **name**.
 - followed by the **variable list**.

- ◊ code example:

- `enum primaryColor myColor, gregsColor;`

- here we have two names of type **primaryColor**.

- **myColor**

- **gregsColor**

- this is where we're using the enum, we've already defined what the **enum** can be, just **yellow**, **blue** or **red**.

- the previous variables can only contain the values: **yellow**, **blue** or **red**, it can't contain anything else.

- ◊ The previous code example defines two variables **myColor** and **gregsColor**, to be of type **primaryColor**:

- the only permissible values that can be assigned to these variables are the names **yellow**, **blue** and **red**.

- `myColor = red;`

- ◊ This is a very powerful data type because you're essentially defining your own data types that can contain only certain values.

- ◊ Another example:

```
enum month {January, February, March, April, May,
June, July, August, September, October, November,
December};
```

- **Enum as ints** (enums under the hood):

- ◊ the compiler actually treats enumeration identifiers as integer constants:

- first name in list 0.

- what this means is that, if we have:

```
enum month thisMonth;
```

→ `thisMonth = February;`

⇒ the **thisMonth** variable is actually gonna be equal to an integer.

⇒ but we can still compare it and use it to compare it against **February**.

⇒ if you were to print out **thisMonth** to the console, it would actually be an integer value corresponding to when it was defined in the list.

- ◊ The value **1** is assigned to **thisMonth** (and not the name **February**) because it is the second identifier listed inside the the enumeration list.

- this is important because you can then compare it against **ints** as well as comparing against actual values like **February**.

- ◊ If you want to have a specific integer value associated with an enumeration identifier, the integer can be assigned to the identifier when the data type is defined.

- code example:

- `enum direction {up, down, left = 10, right};`

→ an **enumerated** data type **direction** is defined with the values **up**, **down**, **left** and **right**.

⇒ **up = 0** because it appears first in the list.

⇒ **down = 1** because it appears next.

⇒ **left = 10** because it was explicitly assigned this value.

⇒ **right = 11** because it appears immediately after left in the list.

- **Char:**

- ◊ Char represents a single character such as the letter 'a', the digit character '6' or a semicolon (;).
- ◊ Character literals use single quotes such as 'A' or 'Z'.
 - anything inside single quotes in a program is referred to as a **character** data type.
- ◊ You can also declare **char** variables to be unsigned:
 - can be used to explicitly tell the compiler that a particular variable is a signed quantity.
 - what it's actually telling the compiler is that the **character** can only be a positive number.
 - this is because character can be represented as numbers in the **ascii** table.
- ◊ We will talk about a **character string** in another lecture much different than a **single** character.
 - a **sequence of characters**.

- **Declaring a Char:**

- ◊ code example:
 - ```
char broiled; /* DECLARE A CHAR VARIABLE */
```

    - we're declaring a character and use it you can simply specify the data type **char** and then the identifier (name).
    - ```
broiled = 'T';
```

 - we're declaring a variable named **broiled** of type **char**.
 - when we assign to that variable we put the value inside single quotes.

- ◊ Not valid **char declarations**:

1-

```
broiled = T; /* NOT A VALID STATEMENT  
IT'LL THINK T IS A  
VARIABLE */
```

2-

```
broiled = "T"; /* NOT A VALID STATEMENT,  
IT'LL THINK "T" IS A  
STRING */
```

- double quotes in C means it's a string.
- if you omit the quotes, the compiler thinks that T is the name of variable and it's not gonna work.
 - if you use double quotes, it thinks you are using a **string**.
 - you can also use the **numerical code** to assign value.

- this will use the **ASCII table**.

- code example:

```
char grade = 65; /* OK FOR ASCII,  
⇒ BUT POOR STYLE */  
⇒
```

- **Escape Characters:**

- ◊ C contains special characters that represent actions:

- **backspacing**
 - **going to the next line**
 - **making the terminal bell ring (or speaker beep)**

- ◊ We can represent these actions by using special symbol sequences:

- called special sequences.

- ◊ Escape sequence **must be enclosed** in **single quotes** when assigned to a character variable.

- code example:

```
- char x = '\n';
```

- a lot of the escape characters start with a backslash.
 - the **\n** represents a new line in a program.
 - and then print the variable **x** to advance the printer or screen **one line**.

- ◊ Escape characters table:

▪

| Sequence | Meaning |
|----------|--|
| \a | Alert (ANSI C). |
| \b | Backspace. |
| \f | Form feed. |
| \n | Newline. |
| \r | Carriage return. |
| \t | Horizontal tab. |
| \v | Vertical tab. |
| \\\ | Backslash (\). |
| \' | Single quote ('). |
| \" | Double quote ("). |
| \? | Question mark (?). |
| \ooo | Octal value. (o represents an octal digit.) |
| \xhh | Hexadecimal value. (h represents a hexadecimal digit.) |

- Code example on IDE:

- ◊ **Enums:**

```
/*
    AUTHOR: JFITECH
    DATE: DECEMBER 22ND, 2021
    PURPOSE: ENUM AND CHAR DATA TYPES

*/

#include <stdio.h>

int main() {
    enum gender {male, female};

    enum gender myGender; //HERE WE'RE USING THE PREVIOUSLY CREATED VARIABLE
    //WE HAVEN'T ASSIGNED IT ANY VALUES

    myGender = male;

    return 0;
}
```

- there's two steps when you create an **enum**:
 1. You have to define the type.
 2. You have to create the variable of that type.

⇒ you use the **enum** keyword for both.

- to see what this example represents in terms of **integers**:

```
/*
AUTHOR: JFITECH
DATE: DECEMBER 22ND, 2021
PURPOSE: ENUM AND CHAR DATA TYPES

*/

#include <stdio.h>
#include <stdbool.h>

int main() {
    enum gender {male, female};

    enum gender myGender; //HERE WE'RE USING THE PREVIOUSLY CREATED VARIABLE
    WE HAVEN'T ASSIGNED IT ANY VALUES

    myGender = male;

    enum gender anotherGender = female;

    bool isMale = (myGender == anotherGender); //THIS WOULD RETURN FALSE
    VALUE) BECAUSE myGender IS NOT EQUAL TO anotherGender

    return 0;
}
```



- we can see the values of **enums** by printing them out:

- we'll notice that if we were to print out the value of the **enumGender**:
 - we would notice that **gender myGender** is gonna be equal to **0** because that's the first in the list.

⇒ **anotherGender** would be equal to **1**.

◇ **Chars:**

```
/*
AUTHOR: JFITECH
DATE: DECEMBER 22ND, 2021
PURPOSE: ENUM AND CHAR DATA TYPES

*/

#include <stdio.h>
#include <stdbool.h>

int main() {
    char myCharacter = 'J';
    char myCharacter2 = '\n'; //WE CAN ALSO ASSIGN THEM ESCAPE CHARACTERS WHICH REPRESENT CERTAIN ACTION

    printf("%c", myCharacter2);
}
```

```
    return 0;  
}
```

- the output of this will be a new line:
 - here we're printing out a specific action, we're not printing any output.

32. Format Specifiers

- This is in a section for variables and data types:
 - ◊ the reason for this is that it's very important to be able to display the value of variables as output and **format specifiers** help you do that.
- We're gonna use format specifiers to specify the type of data that we wanna display.

• Overview:

- ◊ Format specifiers are used when displaying variables as output:
 - they specify the type of data of the variable to be displayed.
 - code example:

```
int sum = 89;  
→ printf("The sum is %d\n", sum);
```

⇒ the way that we can identify the format specifiers is through the **% (percent)** symbol.

- ◊ Format specifiers specify the format of data (how it should be displayed).
- ◊ In the previous code example:
 - the **printf()** function can display as output the values of variables:
 - it has two items or arguments enclosed within the parentheses:
 - inside the parentheses are what is called function arguments:
 - ⇒ data that you're passing to the function.
 - the **first argument** is what's inside the double quotes " ".
 - ⇒ this represents what you want to be displayed as output:
 - the **%d** means that it doesn't actually print out the **percent symbol**, print out what's the next argument.
 - ◊ **%d** is gonna be mapped to the variable **sum**.
 - **%d** is telling you that it's an integer.
 - depending on the type of data what's after the percent symbol is gonna change.
 - the **second argument** would be the variable **sum**.
 - arguments are separated by a comma.
 - first argument to the **printf()** routine is always the character string to be displayed.
 - along with the display of the character string, you might also frequently want to have the value of certain program variables displayed.
 - ◊ The **%** character is what's important in terms of **format specifier**:
 - it's a special character recognized by the **printf()** function:

- what immediately follows the **%** symbol is gonna identify the type of data that we wanna print:

- this is gonna be a single character.
- ⇒ the data that we're gonna print is after the comma
 - you could put many percent symbols to print many different types of data.

◊ Code example walk-through:

```
/*
AUTHOR: JFITECH
DATE: DECEMBER 23RD, 2021
PURPOSE: FORMAT SPECIFIERS

*/

#include <stdio.h>
#include <stdbool.h>

int main(void) {
    // HERE WE'RE DECLARING AND INITIALIZING
VARIABLES
    int integerVar = 100;
    float floatingVar = 331.79;
    double doubleVar = 8.44e+11;
    char charVar = 'W';

    bool boolVar = 0;

    printf("IntegerVar = %i\n", integerVar); /* THE
FIRST ARGUMENT INSIDE THE DOUBLE QUOTES IS WHAT'S
GONNA BE PRINTED OUT TO THE SCREEN,
THE
SECOND ARGUMENT IS GONNA BE MAPPED TO THE FORMAT
SPECIFIER
THE \n
IS JUST PRINTING NEW LINES BECAUSE IT'S AN ESCAPE
SEQUENCE */
    printf("floatingVar = %f\n", floatingVar);
    printf("doubleVar = %e\n", doubleVar);
    printf("doublerVar = %g\n", doubleVar);
    printf("charVar = %c\n", charVar);

    printf("boolVar = %i\n", boolVar);

    printf("integerVar = %i test %f\n", integerVar,
floatingVar); //TO PRINT MORE THAN ONE VARIABLE IN A
SINGLE LINE

    return 0;
}
```

- on the **printf()** arguments we have different letters:

- these letters identify different data types:

- ⇒ **%i** and **%d** → **integer**
- ⇒ **%f** → **float**
- ⇒ **%e** → **double**

- ⇒ **%g** → **double** (it's just a different format)
- ⇒ **%c** → **character**
- ⇒ **%i** → **boolean** (this is because boolean values are either **true** or **false**, **0** or **1**, so we can use a **%i** there as well)

→ on the previous code example:

- ⇒ **%i** gets mapped to whatever the values of **integerVar** are, in this case it's 100, the same for the rest of the variables.

⇒ we can notice that the format is a little different depending on the specifier you provide (for **double** variables).

- ⇒ the **bool** variable is a **0** because an **integer** specifier was provided.

- ⇒ the **float variable** gives more precision because we used the **%f** specifier:

- this also depends on the machine that you're running, because it's machine independent it may be displayed differently.

- ⇒ if we wanted to print more than one variable in one line we could do the following:

- ```
printf("integerVar = %i test %f\n", integerVar, floatingVar);
```

### ◊ The **Width Specifier**:

- this is where you have a number after the percent **%** symbol that specifies the minimum field width to be printed if the characters are less than the size of the width that remains in space:

- this just really has to do with precision, **floating point numbers** and **whole numbers**.

→ code example:

```
/*
AUTHOR: JFITECH
DATE: DECEMBER 23RD, 2021
PURPOSE: WIDTH FORMAT SPECIFIERS

*/

#include <stdio.h>
#include <stdbool.h>

int main(void) {
 float x = 3.92324244; //WE CAN PRINT THE WIDTH OF THIS BY SPECIFYING
 //PRECISE THE NUMBERS HAVE TO BE

 printf("%.2f", x); // THE %.2f WILL ONLY PRINT THE FIRST TWO DECIMAL
 //THAT WERE ASSIGNED TO THE VARIABLE

 return 0;
}
```

- we can print the width of this by specifying how precise the numbers have to be:

◊ to do this we use:

```
• printf("% .2f", x);
```

- this will only **print** the **first two decimal numbers** that were assigned to the variable

## • Summary:

| Type                   | Constant Examples                 | printf chars     |
|------------------------|-----------------------------------|------------------|
| char                   | 'a', '\n'                         | %c               |
| _Bool                  | 0, 1                              | %i, %u           |
| short int              | —                                 | %hi, %hx, %ho    |
| unsigned short int     | —                                 | %hu, %hx, %ho    |
| int                    | 12, -97, 0xFFE0, 0177             | %i, %x, %o       |
| unsigned int           | 12u, 100U, 0XFFFu                 | %u, %x, %o       |
| long int               | 12L, -2001, 0xffffL               | %li, %lx, %lo    |
| unsigned long int      | 12UL, 100ul, 0xffeeUL             | %lu, %lx, %lo    |
| long long int          | 0xe5e5e5e5LL, 5001l               | %lli, %llx, %llu |
| unsigned long long int | 12ull, 0xffeeULL                  | %llu, %llx, %llu |
| float                  | 12.34f, 3.1e-5f, 0x1.5p10, 0x1P-1 | %f, %e, %g, %a   |
| double                 | 12.34, 3.1e-5, 0x.1p3             | %f, %e, %g, %a   |
| long double            | 12.341, 3.1e-51                   | %Lf, %Le, %Lg    |



## **33. Command line arguments**

- This is a way to pass command line arguments into your program.

- **Overview:**

- ◊ There are times when a program is developed that requires the user to enter a small amount of information at the terminal.
  - command line arguments are basically program arguments.
    - this is how you can pass data to a program.
- ◊ This information might consist of a number indicating the triangular number that you want to have calculated or a word that you want to have looked up in the dictionary.
- ◊ There are two ways of handling data:
  - requesting the data from the user.
  - supply the information to the program at the time the program is executed (command line arguments).
- ◊ We know that the **main()** function is a special function in C:
  - entry point of the program:
- ◊ When **main()** is called by the runtime system, two arguments are actually passed to the function:
  - the first argument (**argc** for argument count) is an integer value that specifies the number of arguments typed on the command line.
  - the second argument (**argv** for argument vector) is an **array of character pointers (strings)**.
    - arguments are how you pass data to a function.
  - you can pass the **main()** function some data when you run the program:
    - that data can be specified when you run the program at the command line.
- ◊ The first entry in this array is a pointer to the name of the program that is executing.
- ◊ code example:

```
/*
 AUTHOR: JFITECH
 DATE: DECEMBER 23RD, 2021
 PURPOSE: COMMAND LINE ARGUMENTS

*/
#include <stdio.h>
```

```

#include <stdbool.h>

int main(int argc, char *argv[]) { /* THIS MAIN
FUNCTION HAS TWO PARAMETERS.
WHAT'S AFTER THE
COMMA IS SOME DATA THAT YOU'RE PASSING TO THE
PROGRAM
THE argc WOULD
BE THE NUMBER OF THE DATA */

 return 0;
}

```

◊ code example in IDE:

```

/*
AUTHOR: JFITECH
DATE: DECEMBER 23RD, 2021
PURPOSE: COMMAND LINE ARGUMENTS

*/

#include <stdio.h>
#include <stdbool.h>

int main(int argc, char *argv[]) { /* THIS MAIN
FUNCTION HAS TWO PARAMETERS.
WHAT'S AFTER THE
COMMA IS SOME DATA THAT YOU'RE PASSING TO THE
PROGRAM
THE argc WOULD
BE THE NUMBER OF THE DATA */

 int numberOfArguments = argc;
 char *argument1 = argv[0]; // THIS SETS argument1
VARIABLE EQUAL TO THE PROGRAM NAME
 char *argument2 = argv[1];

 printf("Number of arguments: %d",
numberOfArguments);
 printf("Argument 1 is the program name: %s",
argument1); // HERE WE USE %s BECAUSE THIS IS A
STRING
 printf("Argument 2 is the command line argument:
%s", argument2);

 return 0;
}

```

- **argc** can be any name, but it represents the argument count
- we're gonna create some variables that demonstrate some data being passed
- because the data is being passed it needs to be stored somewhere:
  - ⇒ we're gonna create some variables:
    - **int** **numberOfArguments**
    - **char \*****argument1**
    - **char \*****argument2**

- ◊ When we execute a program we can pass data to it:
  - since we're not executing the command, we're doing the menu:
    - we have to pass the data via the **IDE**:
      - to pass data as a command line argument to a program (in **Code::Blocks**):
      - ⇒ go to your **project**, then highlight it:
        - go up to the menu option that says **Project**:
        - ◊ and click on **Set Programs' Arguments**:
        - here a window will pop up:
        - here you can basically say your arguments, you can put in as many as you want in here:
        - this data is passed to the program when it is run:
        - ⇒ the program will print out on **argument2** whatever data we passed to it.

◊ This is a way to pass data to the program without asking from the user specifically.

  - you just type it in the IDE and pass it to the program.
    - you can pass many many arguments:
      - either separated by **new lines**
      - ⇒ or
      - S
    - this is useful to pass data to a program without asking the user to type something in the keyboard:
      - **if you don't do it this way, the program will pause and ask the user for input:**
      - then it'll save that input in a variable and you'll have to continue executing.

## **34. (Challenge) Print the Area of a Rectangle**

- In this challenge, you are going to create a C program that displays the perimeter and area of a rectangle.

- **Requirements:**

- The program should create **4 variables of type double**:
    - one variable to store the **width** of a rectangle.
    - one variable to store the **height** of a rectangle.
    - one variable to store the **perimeter** of the rectangle.
    - one variable to store the **area** of the rectangle.
  - The program should perform the calculation for the perimeter of a rectangle:
    - use the **+** operator for addition and the **\*** operator for multiplication.
    - Perimeter**: is calculated by adding the height and width and then multiplying by two.  
**- perimeter = (h + w) \* 2**
    - Area**: is calculated by multiplying the width and the height variables.  
**- area = w \* h**
  - The program should display the **height**, **width** and **perimeter variables** in the **correct format** in **one print statement**.
  - The program should display the **height**, **width** and **area variables** in the **correct format** in **one print statement**.

- **Hints:**

- Create a new project:
    - give it a meaningful name.
  - Edit the **main.c** file that is auto-generated for you as part of creating the project.
  - Declare and initialize** the **height** and **width** variables to any value (need to be of type **double**).
    - We're not getting input from the user for this program:
      - you can if you want to, if you feel comfortable enough to do it.**
        - or you could just harcode the variables.
  - Declare the perimeter and area variables values to be **0.0**
  - Assign to the perimeter and area values the correct data based on the calculations:

- perimeter = 2.0 \* (height + width);**
- area = width \* height;**

- **Next steps:**

- Use the **printf** function and the correct format specifier to display the required output.
  - format specifier for doubles:
    - %f**
    - you could use something like:
      - %f2**
  - to see how the decimal precision changes.
- Compile and link the source code.
- Run the program.
- Analyze the output to confirm it's correct.

# My Code

```
/*
 AUTHOR: JFITECH
 PURPOSE: THIS PROGRAM PRINTS OUT THE AREA AND PERIMETER OF A RECTANGLE TO
THE SCREEN
 DATE: DECEMBER 29TH, 2021
*/

#include <stdio.h>

int main() {
 // DECLARE VARIABLES
 double width = 0;
 double height = 0;
 double perimeter = 0;
 double area = 0;

 //REQUEST USER INPUT
 printf("Enter the height and width of the triangle: ");
 scanf("%lf %lf", &height, &width);

 //PERIMETER AND AREA LOGIC
 perimeter = (height + width) * 2.0;
 area = height * width;

 //FINAL PRINT STATEMENTS
 printf("\nThe height is: %.2lf and the width is: %.2lf \nThe perimeter is: %.2lf\n",
height, width, perimeter);
 printf("\nThe height is: %.2lf and the width is: %.2lf \nThe area is %lf",
height, width, area);

 return 0;
}
```

## **35. (Demonstration) Print the Area of a Rectangle**

- We could also include the **standard library** at the beginning of the program:

◇ `#include <stdlib.h>`

- It's always a good habit to initialize all your variables.
- You could also use program arguments but that's more advanced.

## **36. (Challenge) Create and use an enum type**

- In this challenge, you are to create a C program that defines an **enum** type and uses that type to display the values of some variables.

- **Requirements:**

- The program should create an enum type named **Company**:
    - valid values for this type are **GOOGLE, FACEBOOK, XEROX, YAHOO, EBAY, MICROSOFT**
    - all caps!
    - the order should be the same as they represent integers
  - The program should create three variables of the above enum type that are assigned values:
    - XEROX, GOOGLE and EBAY**
  - The program should display as output, the value of the three variables with each variable separated by a newline:
    - correct output would be if **XEROX, GOOGLE and EBAY** variables are printed in that order:
      - **2**
      - **0**
      - **4**

- **Hints:**

- ◊ Define the enum type and its values:
    - use an **enum** keyword.
    - use the curly brackets **{ }**
  - ◊ **Declare** and **initialize** three variables with the values specified on the previous slide
  - ◊ Use **printf()** to display the value of the **enum** variables:
    - use the **\n** escape sequence to display a new line.
    - since **enums** are **ints** we can use the following **format specifier**:
      - **%d**

# **My Code**

```
/*
AUTHOR: JFITECH
PURPOSE: THIS PROGRAM PRINTS OUT ENUM DATA TYPES
DATE: DECEMBER 29TH, 2021
*/

#include <stdio.h>
#include <stdlib.h>

int main() {
 //DECLARE THE ENUMS
 enum Company {GOOGLE, FACEBOOK, XEROX, YAHOO, EBAY, MICROSOFT};
 enum Company myCompany1 = XEROX;
 enum Company myCompany2 = GOOGLE;
 enum Company myCompany3 = EBAY;

 //PRINT THE ENUMS
 printf("The values are:\n %d\n %d\n %d\n", myCompany1, myCompany2,
myCompany3);

 return 0;
}
```

## **37. (Demonstration) Create and use an enum type**

- Notice that the **enum** values that are assigned don't go in double quotes " ":
  - ◊ this is because they're actual values.
- **Enums** provide some kind of validation.

## ***Section 6: Operators***

# 38. Overview

- **Operators** are functions that use a symbolic name:
  - ◊ perform mathematical (arithmetic) or logical functions.
- **Operators** are predefined in C, just like they are in most other languages, and most operators tend to be combined with the infix style:
  - ◊ **5 + 8**
    - the **+** symbol in between the 5 and 8 is referred to as **infix** style.
    - **+ 5 8 :**
      - this previous example is considered as **prefix** style.
      - ⇒ **5 8 + :**
        - the previous example is known as **postfix** style.
  - **Operator:**
    - ◊ is the **symbol**.
  - **Operand:**
    - ◊ are the **arguments** to the symbol.
  - A **logical operator** (sometimes called a “Boolean operator”) is an operator that returns a Boolean result that's based on the Boolean result of one or two other expressions.
  - An arithmetic operator is the mathematical function that takes two operands and performs a calculation on them.
  - Other operators include **assignment**, **relational** (**<**, **>**, **!=**), **bitwise** (**<<**, **>>**, **~**).
    - ◊ **Assignment:**
      - you're assigning data.
      - the equals operator → **=**
    - ◊ **Relational:**
      - comparison operators:
        - **<**
        - **>**
        - **!=**
    - ◊ **Bitwise:**
      - allows us to perform operations on bits:
        - **<<**
        - **>>**
        - **~**

- **Expressions** and **statements**:

- ◊ **Statements** form the basic program steps of C, and **most statements** are **constructed from expressions**:

- you may have seen an expression inside of a statement

- ◊ An **expression consists** of a **combination of operators** and **operands**:

- **operands** are what an **operator operates** on.
- operands **can be constants, variables, or combinations of the two.**
- **every expression** has a **value**.

- ◊ The following are examples of expressions:

```
-6 //
THE MINUS SYMBOL IS AN ACTUAL OPERATOR IN
C, IT'S A UNARY OPERATOR, A SINGLE OPERATOR
WITH A SINGLE ARGUMENT
4 + 21 // THIS IS
AN EXPRESSION
a * (b + c / d) /20 // THIS IS A MORE
COMPLICATED EXPRESSION
q = 5 * 2 // ON THE
RIGHT YOU HAVE AN EXPRESSION, ON THE LEFT
YOU HAVE A VARIABLE, TOGETHER THESE TWO
MAKE UP A STATEMENT
x = ++q % 3 //
q > 3
```

- ◊ **Statements** are the building blocks of a program (declaration):

- a program is a series of statements with special syntax ending with a semicolon (simple statements).
- a complete instruction to the computer.

- ◊ **Declaration** statement:

- `int Jfitech;`

- - **data type** followed by **variable name**.

- ◊ **Assignment** statement:

- `Jfitech = 5;`

- ◊ **Function call** statement:

- `printf("Jfitech");`

- ◊ **Structure** statement:

- `while (Jfitech < 20) Jfitech = Jfitech + 1;`

◊ **Return statement:**

- `return 0;`

- this is used when we return data from a function.

◊ C considers any expression to be a statement if you append a semicolon (expression statements):

- so the following example is perfectly valid in C:

- **8;**
- **3 - 4;**

◊ **Compound Statements:**

- two or more statements grouped together by enclosing them in braces {} (this is what is known as a block of code).

- the following is an example of this:

```
int index = 0;
while (index < 10) {
 printf("Hello");
 index += 1;
→ }
```

# 39. Basic Operators

- In this lecture we're going to discuss some of the basic **operators** supported by the C programming language.

- **Overview:**

- ◊ Lets discuss, **arithmetic**, **logical**, **assignment** and **relational** operators.
- ◊ An **arithmetic operator** is a mathematical function that takes two operands and performs a calculation on them.
- ◊ A **logical operator** (sometimes called a "**Boolean Operator**") is an operator that returns a **Boolean** result that's based on the **Boolean** result of one or two other expressions.
- ◊ **Assignment operators** set variables equal to values:
  - assigns the value of the expression at its right to the variable at its left.
- ◊ A **relational operator** will compare variable against each other.

- **Arithmetic Operators:**

| Operator | Description                                                  | Example       |
|----------|--------------------------------------------------------------|---------------|
| +        | Adds two operands.                                           | $A + B = 30$  |
| -        | Subtracts second operand from the first.                     | $A - B = -10$ |
| *        | Multiplies both operands.                                    | $A * B = 200$ |
| /        | Divides numerator by de-numerator.                           | $B / A = 2$   |
| %        | Modulus Operator and remainder of after an integer division. | $B \% A = 0$  |
| ++       | Increment operator increases the integer value by one.       | $A++ = 11$    |
| --       | Decrement operator decreases the integer value by one.       | $A-- = 9$     |

- ◊
  - the **increment** and **decrement** operators on this example are known as **post fix** (the assignation comes after the statement).
    - if it's before it'll be known as **pre fix**.

◊ arithmetics code example:

```
/*
 AUTHOR: JFITECH
 PURPOSE: THIS PROGRAM SHOWS US DIFFERENT
 ARITHMETIC OPERATORS
 DATE: FEBRUARY 13TH, 2022
*/

#include <stdio.h>
#include <stdlib.h>

int main() {
 //DECLARE THE VARIABLES
 int a = 33;
 int b = 15;
 int result;

 result = a + b; // YOU CAN CHANGE THE OPERATOR
 FOR - / * %

 printf("The result is: %d\n", result);

 return 0;
}
```

◊ Post fix notation code example:

```
/*
 AUTHOR: JFITECH
 PURPOSE: THIS PROGRAM SHOWS US HOW THE POST FIX
 NOTATION WORKS
 DATE: FEBRUARY 13TH, 2022
*/

#include <stdio.h>
#include <stdlib.h>

int main() {
 //DECLARE THE VARIABLES
 int a = 33;
 int b = 15;
 int result;

 printf("a is: %d\n", a++);
 printf("a now is: %d \n", a);

 return 0;
}
```

◊ Pre fix notation code example:



- the previous are also **unary operators** because they only have **one operand**.

## • Logical Operators:

| Operator | Description                                                                                                                                                | Example            |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------|
| &&       | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.                                                           | (A && B) is false. |
|          | Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.                                                       | (A    B) is true.  |
| !        | Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false. | !(A && B) is true. |

- ◊ - **non-zero** means **1**

### ◊ logical operator && code example:

```
/*
AUTHOR: JFITECH
PURPOSE: THIS PROGRAM SHOWS US LOGICAL OPERATORS
DATE: FEBRUARY 13TH, 2022
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main() {
 //DECLARE THE VARIABLES
 bool a = true;
 bool b = true;
 bool result;

 result = a && b;

 printf("%d\n", result);

 return 0;
}
```

### ◊ logical operator || code example:

```

/*
 AUTHOR: JFITECH
 PURPOSE: THIS PROGRAM SHOWS US LOGICAL OPERATORS
 DATE: FEBRUARY 13TH, 2022
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main() {
 //DECLARE THE VARIABLES
 bool a = true;
 bool b = false;
 bool result;

 result = a || b;

 // ONLY ONE OF THE VARIABLES HAS TO BE true IN ORDER FOR THE RESULT TO BE
true
 printf("%d\n", result);

 return 0;
}

```

## ◊ logical operator ! code example:

```

/*
 AUTHOR: JFITECH
 PURPOSE: THIS PROGRAM SHOWS US LOGICAL OPERATORS
 DATE: FEBRUARY 13TH, 2022
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main() {
 //DECLARE THE VARIABLES
 bool a = true;
 bool b = false;
 bool result;

 result = !a;

 // ONLY ONE OF THE VARIABLES HAS TO BE true IN ORDER FOR THE RESULT TO BE
true
 printf("%d\n", result);

 return 0;
}

```

## • Assignment Operators:

| Operator | Description                                                                                                                         | Example                                             |
|----------|-------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------|
| =        | Simple assignment operator                                                                                                          | $C = A + B$ will assign the value of $A + B$ to $C$ |
| +=       | Add AND assignment operator. It adds the right operand to the left operand and assigns the result to the left operand.              | $C += A$ is equivalent to $C = C + A$               |
| -=       | Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.  | $C -= A$ is equivalent to $C = C - A$               |
| *=       | Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand. | $C *= A$ is equivalent to $C = C * A$               |

- the last three operators are called **compound operators**

|    |                                                                                                                                |                                        |
|----|--------------------------------------------------------------------------------------------------------------------------------|----------------------------------------|
| /= | Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand. | $C /= A$ is equivalent to $C = C / A$  |
| %= | Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.               | $C %= A$ is equivalent to $C = C \% A$ |
| <= | Left shift AND assignment operator.                                                                                            | $C <= 2$ is same as $C = C << 2$       |
| >= | Right shift AND assignment operator.                                                                                           | $C >= 2$ is same as $C = C >> 2$       |
| &= | Bitwise AND assignment operator.                                                                                               | $C &= 2$ is same as $C = C \& 2$       |
| ^= | Bitwise exclusive OR and assignment operator.                                                                                  | $C ^= 2$ is same as $C = C ^ 2$        |
| =  | Bitwise inclusive OR and assignment operator.                                                                                  | $C  = 2$ is same as $C = C   2$        |

- the last five are **bitwise operators**, we'll review these in a future lesson

## • Relational Operators:



|                        |                                                                                                                                |                                         |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------|
| <code>/=</code>        | Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand. | $C /= A$ is equivalent to $C = C / A$   |
| <code>%=</code>        | Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.               | $C \%= A$ is equivalent to $C = C \% A$ |
| <code>&lt;&lt;=</code> | Left shift AND assignment operator.                                                                                            | $C <<= 2$ is same as $C = C << 2$       |
| <code>&gt;&gt;=</code> | Right shift AND assignment operator.                                                                                           | $C >>= 2$ is same as $C = C >> 2$       |
| <code>&amp;=</code>    | Bitwise AND assignment operator.                                                                                               | $C \&= 2$ is same as $C = C \& 2$       |
| <code>^=</code>        | Bitwise exclusive OR and assignment operator.                                                                                  | $C ^= 2$ is same as $C = C ^ 2$         |
| <code> =</code>        | Bitwise inclusive OR and assignment operator.                                                                                  | $C  = 2$ is same as $C = C   2$         |

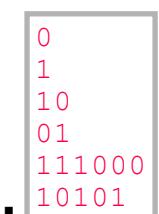


# 40. Bitwise Operators

- With these operators you can perform an operation (like addition) and then also assign.
- Overview:**
  - C offers bitwise logical operators and shift operators:
    - they look something like the logical operators we saw earlier but are quite different.
    - they operate on the bits in integer values.
  - Not used in the common program.
    - it's more of an advanced technique
      - we're not really gonna focus much on them
  - One major use of the bitwise **AND**, **&**, and the bitwise **OR**, **|**, is in operations to test and set individual bits in an integer variable:
    - can use individual bits to store data that **involve one of two choices**.
    - the difference from the logical operators (**&&**, **||**) is that these have only one character (**&**, **|**).
  - You can turn on and off each bit in (depending on the operating system) a byte
    - and then you can use those bits to tell you something.
    - you can store a lot of data by having bits represent some kind of data
  - You could use a single integer variable to store several characteristics of a person:
    - store whether the person is male or female with one bit.
    - use three other bits to specify whether the person can speak German, French or Italian.
    - another bit to record whether the person's salary is \$50,000 or more.
    - in just five bits you have a substantial set of data recorded.

## • Binary Numbers:

- A binary number is a number that includes only **one** and **zeroes**.
- The number could be of any length.
- The following are all examples of binary numbers:



0  
1  
10  
01  
111000  
10101

|             |
|-------------|
| 0101010     |
| 10111110101 |
| 0110101110  |
| 000111      |

- **byte**: 8 bits
  - **integer**: 4 bytes (on most platforms, depends on the architecture, but mostly **32 bits**)
    - so you have 32 values that you can store 1's and 0s to indicate on or off.

- ◊ Every binary number has a corresponding Decimal value (and vice versa).

- so you need to know how to convert binary to decimal:

| Binary Number | Decimal Equivalent |
|---------------|--------------------|
| 1             | 1                  |
| 10            | 2                  |
| 11            | 3                  |
| ...           | ...                |
| 1010111       | 87                 |

- each position for a binary number has a value.
  - for each digit, multiply the digit by its position value.
  - and add up all the results to get the final result.
  - in general, the "position values" in a binary number are the powers of two.

## Example

- The value of binary 01101001 is decimal 105. This is worked out below:

# Bitwise Operators (tutorials point)

| Operator | Description                                                                                                                | Example                                                    |
|----------|----------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|
| &        | Binary AND Operator copies a bit to the result if it exists in both operands.                                              | $(A \& B) = 12$ , i.e., 0000 1100                          |
|          | Binary OR Operator copies a bit if it exists in either operand.                                                            | $(A   B) = 61$ , i.e., 0011 1101                           |
| ^        | Binary XOR Operator copies the bit if it is set in one operand but not both.                                               | $(A ^ B) = 49$ , i.e., 0011 0001                           |
| ~        | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.                                            | $(\sim A) = -61$ , i.e., 1100 0011 in 2's complement form. |
| <<       | Binary Left Shift Operator. The left operand's value is moved left by the number of bits specified by the right operand.   | $A << 2 = 240$ i.e., 1111 0000                             |
| >>       | Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand. | $A >> 2 = 15$ i.e., 0000 1111                              |

LearnProgramming.com

- ◊ **AND:** if both operands are **true** then it's gonna copy a bit.
- **OR:** if either operand is equal to one, it'll copy a bit.
- **XOR:** if you have a true, but only one true, then it's gonna copy a bit and it's gonna say it's **true**.
  - **~:** it's gonna flip (change them) the bits.
  - **<<:** it's going to shift bits to the left.
  - **>>:** you're just shifting them to the right as opposed to the left

- ◊ The other important thing to understand with the bitwise operators is how do you evaluate this **A** and **B**:

# Truth Table

| p | q | p & q | p   q | p ^ q |
|---|---|-------|-------|-------|
| 0 | 0 | 0     | 0     | 0     |
| 0 | 1 | 0     | 1     | 1     |
| 1 | 1 | 1     | 1     | 0     |
| 1 | 0 | 0     | 1     | 1     |

- Now let's take a look at some examples in our IDE:

- ◇ & operator:

```
/*
 AUTHOR: JFITECH
 PURPOSE: THIS PROGRAM SHOWS US BITWISE OPERATORS
 DATE: MARCH 3RD, 2022
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main(){
 //WE'RE GONNA CREATE A COUPLE UNSIGNED VARIABLE
 unsigned int a = 60; // GONNA BE REPRESENTED IN BINARY AS 0011 1100
 unsigned int b = 13; // GONNA BE REPRESENTED IN BINARY AS 0000 1101
 int result = 0;

 result = a & b; // IF a IS EQUAL TO 1 AND b IS EQUAL TO 1, THEN THAT BIT'S
 GONNA BE TURNED ON --> 0000 1100

 printf("\nThe result is: %d", result);

 return 0;
}
```

- prints out:

- 12

- ◇ || operator:

```

/*
 AUTHOR: JFITECH
 PURPOSE: THIS PROGRAM SHOWS US BITWISE OPERATORS
 DATE: MARCH 3RD, 2022
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main() {
 //WE'RE GONNA CREATE A COUPLE UNSIGNED VARIABLE
 unsigned int a = 60; // GONNA BE REPRESENTED IN BINARY AS 0011 1100
 (INVERSE ORDER, LEFT TO RIGHT)
 unsigned int b = 13; // GONNA BE REPRESENTED IN BINARY AS 0000 1101
 (INVERSE ORDER, LEFT TO RIGHT)
 int result = 0;

 result = a | b; // WE'RE ONLY GONNA TURN ON BITS IF EITHER ONE IS TRUE -->
0011 1101

 printf("\nThe result is: %d", result);

 return 0;
}

```

- prints out:

**- 61**

#### ◊ ~ operator:

```

/*
 AUTHOR: JFITECH
 PURPOSE: THIS PROGRAM SHOWS US BITWISE OPERATORS
 DATE: MARCH 3RD, 2022
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main() {
 //WE'RE GONNA CREATE A COUPLE UNSIGNED VARIABLE
 unsigned int a = 60; // GONNA BE REPRESENTED IN BINARY AS 0011 1100
 (INVERSE ORDER, LEFT TO RIGHT)
 unsigned int b = 13; // GONNA BE REPRESENTED IN BINARY AS 0000 1101
 (INVERSE ORDER, LEFT TO RIGHT)
 int result = 0;

 result = ~a; //

 printf("\nThe result is: %d", result);

 return 0;
}

```

- prints out:

**- -61**

#### ◊ << (shift to the left) operator:

▪

```

/*
 AUTHOR: JFITECH
 PURPOSE: THIS PROGRAM SHOWS US BITWISE OPERATORS
 DATE: MARCH 3RD, 2022
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main() {
 //WE'RE GONNA CREATE A COUPLE UNSIGNED VARIABLE
 unsigned int a = 60; // GONNA BE REPRESENTED IN BINARY AS 0000 0000 0000
0000 0000 0000 0011 1100 (INVERSE ORDER, LEFT TO RIGHT)
 unsigned int b = 13; // GONNA BE REPRESENTED IN BINARY AS 0000 1101
(INVERSE ORDER, LEFT TO RIGHT)
 int result = 0;

 result = a << 2; // --> 1111 0000

 printf("\nThe result is: %d", result);

 return 0;
}

```

- prints out:
  - **240**

◊ >> (shift to the right) operator:

```

/*
 AUTHOR: JFITECH
 PURPOSE: THIS PROGRAM SHOWS US BITWISE OPERATORS
 DATE: MARCH 3RD, 2022
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main() {
 //WE'RE GONNA CREATE A COUPLE UNSIGNED VARIABLE
 unsigned int a = 60; // GONNA BE REPRESENTED IN BINARY AS 0000 0000 0000
0000 0000 0000 0011 1100 (INVERSE ORDER, LEFT TO RIGHT)
 unsigned int b = 13; // GONNA BE REPRESENTED IN BINARY AS 0000 1101
(INVERSE ORDER, LEFT TO RIGHT)
 int result = 0;

 result = a >> 4; // --> 0000 0011

 printf("\nThe result is: %d", result);

 return 0;
}

```

- prints out:
  - **3**

◊ When shifting to the **right** and to the **left** make sure you use **32 bit** bits.

# 41. The Cast and sizeof Operators

- These are strange operators in the sense that you wouldn't think exactly they'd be operators.

- **Type Conversions:**

- ◊ Conversion of data between different types can happen automatically (implicit conversion) by the language or explicit by the program:

- to effectively develop C programs, you must understand the rules used for the implicit conversion of floating-point and integer values in C.

- sometimes when developing programs we're gonna have to mix different type of numbers (int, float, etc.)

- we should avoid this as much as we can by using the **Cast** operator

- ⇒ but when you don't use the **Cast** operator understand what's going on.

- ⇒ and what's going on here when you assign data (to different types: **int**, **float**, **doubles**, etc.) the data is either gonna be truncated or it's gonna be promoted:

- **truncated**: means that it becomes less precise.

- **promoted**: means it becomes more precise.

- ◊ Normally, you shouldn't mix types, but there are occasions when it is useful:

- remember, **C is flexible**, gives you the freedom, but don't abuse it (the idea of mixing types).

- try to use the same types when you can:

- integers with integers

- floats with floats

- etc...

- but, when you can't, either understand the implicit conversions or do the explicit conversions using the **Cast** operator.

- Whenever a **floating-point** value is assigned to an **integer** variable in C, the decimal portion of the number gets truncated

- code example:

```
int x = 0;
float f = 12.125
→ x = f; // VALUE STORED IN x IS THE NUMBER 12, ONLY THE int PORTION IS STO
```

- ◊ Assigning an integer variable to a floating variable does not cause any change in the value of the number:

- value is converted by the system and stored in the floating variable.

- ◊ When performing integer arithmetic:

- if two operands in an expression are integers then any decimal portion resulting from a

division operation is discarded, even if the result is assigned to a floating variable.

- if one operand is an int and the other is a float then the operation is performed as a floating point operation.

- **The Cast Operator:**

- ◊ As mentioned, you should steer clear of automatic type conversions, especially of demotions:
  - better to do an explicit conversion
- ◊ It is possible for you to demand the precise type conversion that you want:
  - called a cast and consists of preceding the quantity with the name of the desired type in parentheses.
  - parentheses and type name together constitute a cast operator, i.e. (**type**).
  - the actual type desired, such as **long**, is substituted for the word **type**.
- ◊ The type cast operator has a higher precedence than all the arithmetic operators, except the **unary minus** and **unary plus**:

- precedence is basically the order of the operations.
  - code example:

```
(int) 21.51 + (int) 26.99
/* THE Cast Operator ON THIS PREVIOUS EXAMPLE IS THE (int)
→ /* THAT'S BEFORE THE NUMBERS.
```

⇒ this previous code is evaluated in C as:

- 21 + 26

⇒ usually you don't cast to a more precise type because you're not gonna get more precise, you cast from a more precise type to a less precise:

- like **casting** from a **float** to an **int**.

- **sizeof Operator:**

◊ This operator is kind of strange, not similar to the other ones, this actually looks just like a function.

◊ You can find out how many bytes are occupied in memory by a given type by using the **sizeof** operator:

- **sizeof** is a special keyword in C.
  - it looks like a function because it takes an argument.
- depending on the system that you're running (hardware architecture) different data types are gonna be represented in memory with different sizes:
  - for example:
    - an **int** might be **4 bytes** on one system and **8 bytes** on another one.

- ◊ **sizeof** is actually an operator and not a function:
  - **evaluated at compile time and not at runtime**, unless a variable-length array is used in its argument.
- ◊ The argument to the **sizeof** operator can be a **variable**, an **array name**, the **name** of a **basic data type**, the **name of derived data type**, or an **expression**.
- ◊ **sizeof(int)** will result in the number of bytes occupied by a variable of type **int**.
- ◊ You can also apply the **sizeof** operator to an expression:
  - result is the size of the value that results from evaluating the expression.
- ◊ We'll see the **sizeof** operator being used when we talk about pointers:
  - because when you're calculating how much memory to allocate, which is what pointers are:
    - you use **sizeof** because you wanna be precise.  
→ but you can also use for other things as well.
- ◊ Use the **sizeof** operator wherever possible to avoid having to calculate and hard-code sizes into your program:
  - for example:
    - if you were to hard code, 4 on one system for an int and the other system might be an 8:  
→ so use **sizeof** as much as you can.

## • Other Operators:

- ◊ The **asterisk \*** operator is an operator that represents a pointer to a variable:
  - this asterisk **takes two operands**:
    - **one on the left and one on the right**:
      - if you apply the asterisk operator to a variable and you just say:  
⇒ **\*variable**
      - with only one argument to the right it represents something else:
        - ◊ it represents a pointer, it's **dereferencing** a pointer
- ◊ **? :** (question mark and colon)
  - is an operator used for comparisons (decision making):
    - if **Condition** is true ? then value X: otherwise then value Y  
→ if the thing to the left of the **true** is true, then the value to the right after the colon is evaluated and so forth:
      - it's a short hand notation for an **if - else** statement.
      - ◊ it's called a **ternary operator**.

- ◇ We'll discuss both of these operators when we talk about pointers and decision statements.

## 42. Operator Precedence

- How do you evaluate the order of these operators.
- Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated:
  - ◊ Dictates the order of evaluation when two operators share an operand.
  - ◊ Certain operators have higher precedence than others.
  - ◊ for example:
    - the multiplication operator has a higher precedence than the addition operator.

### • Overview:

- ◊ The order of executing the various operations can make a difference, so C needs unambiguous rules for choosing what to do first.
- ◊ Code example:
  - `x = 7 + 3 * 2`
- ◊ If you don't put a statement inside parentheses the program is gonna do the operator precedence decision for you.
- ◊ In C (following the previous code example), **x** is assigned **13**, not **20**, because the **\*** operator has a higher precedence than **+**:
  - first it does the **3 \* 2** operation, then it adds that result to **7**.
- ◊ Each operator is assigned a precedence level:
  - **multiplication** and **division** have a higher precedence than **addition** and **subtraction**, so they are performed first.
- ◊ Whatever is enclosed in parentheses is executed first, should just always use **()** to group expressions.
  - just always use parentheses due to the amount of operators, you'd have to memorize each operator's precedence level:
    - .

### • Associativity:

- ◊ What if two operators have the same precedence?:
  - then associativity rules are applied:

- ◊ If they share an operand, they are executed according to the order in which they occur in the statement:

- for most operators the order is from left to right:

- code example:

→ `1 == 2 != 3`

⇒ operators `==` have the same precedence:

- associativity of both `==` and `!=` is left to right.
- the expression on the left is executed first and moves towards the right.

- ◊ The previous code example is equal to:

- `((1 == 2) != 3)`

- ◊ **(1 == 2)** executes first resulting into **0 (false)**, then, **(0 != 3)** executes resulting into **1 (true)**.

- ◊ Table (highest to lowest):

| Category       | Operator                                        | Associativity |
|----------------|-------------------------------------------------|---------------|
| Postfix        | <code>() [] -&gt; . ++ --</code>                | Left to right |
| Unary          | <code>+ - ! ~ ++ -- (type)* &amp; sizeof</code> | Right to left |
| Multiplicative | <code>* / %</code>                              | Left to right |
| Additive       | <code>+ -</code>                                | Left to right |
| Shift          | <code>&lt;&lt; &gt;&gt;</code>                  | Left to right |
| Relational     | <code>&lt; &lt;= &gt; &gt;=</code>              | Left to right |
| Equality       | <code>== !=</code>                              | Left to right |

| Category    | Operator                         | Associativity |
|-------------|----------------------------------|---------------|
| Bitwise AND | &                                | Left to right |
| Bitwise XOR | ^                                | Left to right |
| Bitwise OR  |                                  | Left to right |
| Logical AND | &&                               | Left to right |
| Logical OR  |                                  | Left to right |
| Conditional | ?:                               | Right to left |
| Assignment  | = += -= *= /= %=>>= <<= &= ^=  = | Right to left |
| Comma       | ,                                | Left to right |

## **43. (Challenge) Convert minutes to years and days**

- In this challenge you are to create a C program that **converts** the **number** of **minutes** into **days** and **years**.

- **Requirements:**

- The program should ask the user to enter the number of minutes via the terminal.
  - use the **scanf()** function.
- The program should display as output the **minutes** and then its **equivalent** in **days** and **years**.
  - all of these are gonna be **doubles**.
- The program should create variables to store (should all be of type **double**):
  - minutes (can also be of type **int**)
  - minutes in year
  - years
  - days
- Need to perform a calculation and use arithmetic operators:
  - You have to figure out the conversions.

# My Code

```
/*
 AUTHOR: JFITECH
 PURPOSE: THIS PROGRAM PRINTS OUT THE EQUIVALENT OF GIVEN MINUTES TO DAYS
 AND YEARS TO THE SCREEN.
 DATE: MARCH 5TH, 2022
 */

#include <stdio.h>
#include <stdlib.h>

int main() {
 // DECLARE VARIABLES
 double minutes, days, years, minutesInYears, minutesInDays;

 // REQUEST USER INPUT
 printf("\nEnter the minutes that you wish to convert: ");
 scanf("%g", &minutes);

 minutesInDays = minutes / 1440;

 printf("\nThe amount of given minutes in days is: %g\n", minutesInDays);
 //printf("\nThe amount of given minutes in years is: %f", minutesInYears);

 return 0;
}
```

## **44. (Demonstration) Convert minutes to years and days**

- If you want the program to be more precise consider changing the **ints** to **doubles** or even to **longs**.

# Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>

int main() {
 // Declare variables
 int minutesEntered = 0;
 double years = 0.0;
 double days = 0.0;
 double minutesInYear = 0.0;

 printf("Please enter the number of minutes: ");
 // Get user input
 scanf("%d", &minutesEntered); // WE USE %d FOR INTEGER.

 minutesInYear = (60 * 24 * 365);

 years = minutesEntered / minutesInYear;
 days = (minutesEntered / 60.0) / 24.0;

 printf("%d minutes is approximately %f years and %f days\n",
minutesEntered, years, days);

 return 0;
}
```

## **45. (Challenge) Print the byte size of the basic data types**

- In this challenge you are to create a C program that displays the byte size of basic data types supported in C:

◊ the output varies depending on the system you are running the program.

- **Requirements:**

Display the byte size of the following types:

- int**
- char**
- long**
- long long**
- double**
- long double**

You can use the **%zd** format specifier to format each size.

Use the **sizeof** operator.

Test on more than one computer to see the differences.

# My Code

```
/*
 AUTHOR: JFITECH
 PURPOSE: THIS PROGRAM PRINTS OUT THE BYTE SIZE OF THE BASIC DATA TYPES TO
THE SCREEN.
 DATE: MARCH 5TH, 2022
 */

#include <stdio.h>
#include <stdlib.h>

int main() {
 // INT
 printf("\nThe byte size of an int is: %zd", sizeof(int));

 // CHAR
 printf("\nThe byte size of a char is: %zd"), sizeof(char) ;

 // LONG
 printf("\nThe byte size of a long is: %zd", sizeof(long));

 // LONG LONG
 printf("\nThe byte size of a long long is: %zd", sizeof(long long));

 // DOUBLE
 printf("\nThe byte size of a double is: %zd", sizeof(double));

 // LONG DOUBLE
 printf("\nThe byte size of a long double is: %zd\n", sizeof(long double));

 return 0;
}
```

## **46. (*Demonstration*) Print the byte size of the basic data types**

- Try using different format specifiers.

## ***Section 7: Control Flow***

# 47. Overview

- Control flow deals with conditional statements, if statements.
  - ◊ making decisions in your program.
- As well as repeating code, which is loops.
- This is an important section, these are fundamental programming techniques used in most languages.
- We're gonna talk about how they're specific to C.

- **Overview:**

- ◊ The statements inside your source files are generally executed from **top to bottom**, in the order that they appear:
  - **top to bottom** means **sequential**.
- ◊ Control flow statements, however, break up the flow of execution by employing decision making, looping and branching, enabling your program to conditionally execute particular blocks of code:
  - Decision-making statements
    - **if-then**
    - **if-then-else**
    - **switch**
    - **goto**
  - Looping statements:
    - **for**
    - **while**
    - **do-while**
  - Branching statements:
    - **break**
    - **continue**
    - **return**

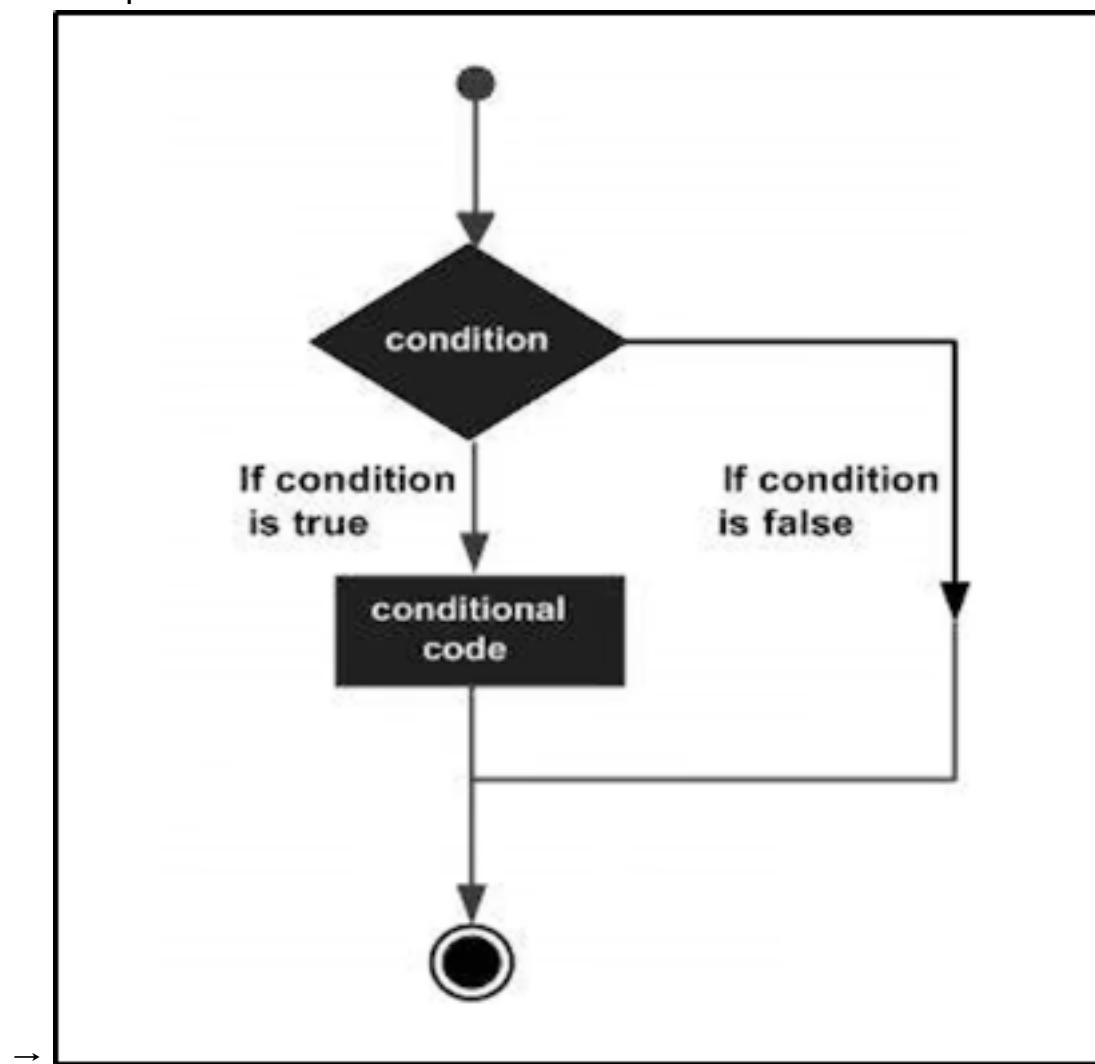
→ these are used inside a decision or a loop.

- **Decision Making:**

- ◊ Structures require that the programmer specify one or more conditions to be evaluated or

tested by the program:

- if a condition is **true** then a statement or statements are executed.
- if a condition is **false** then other statements are executed.
- flow map:

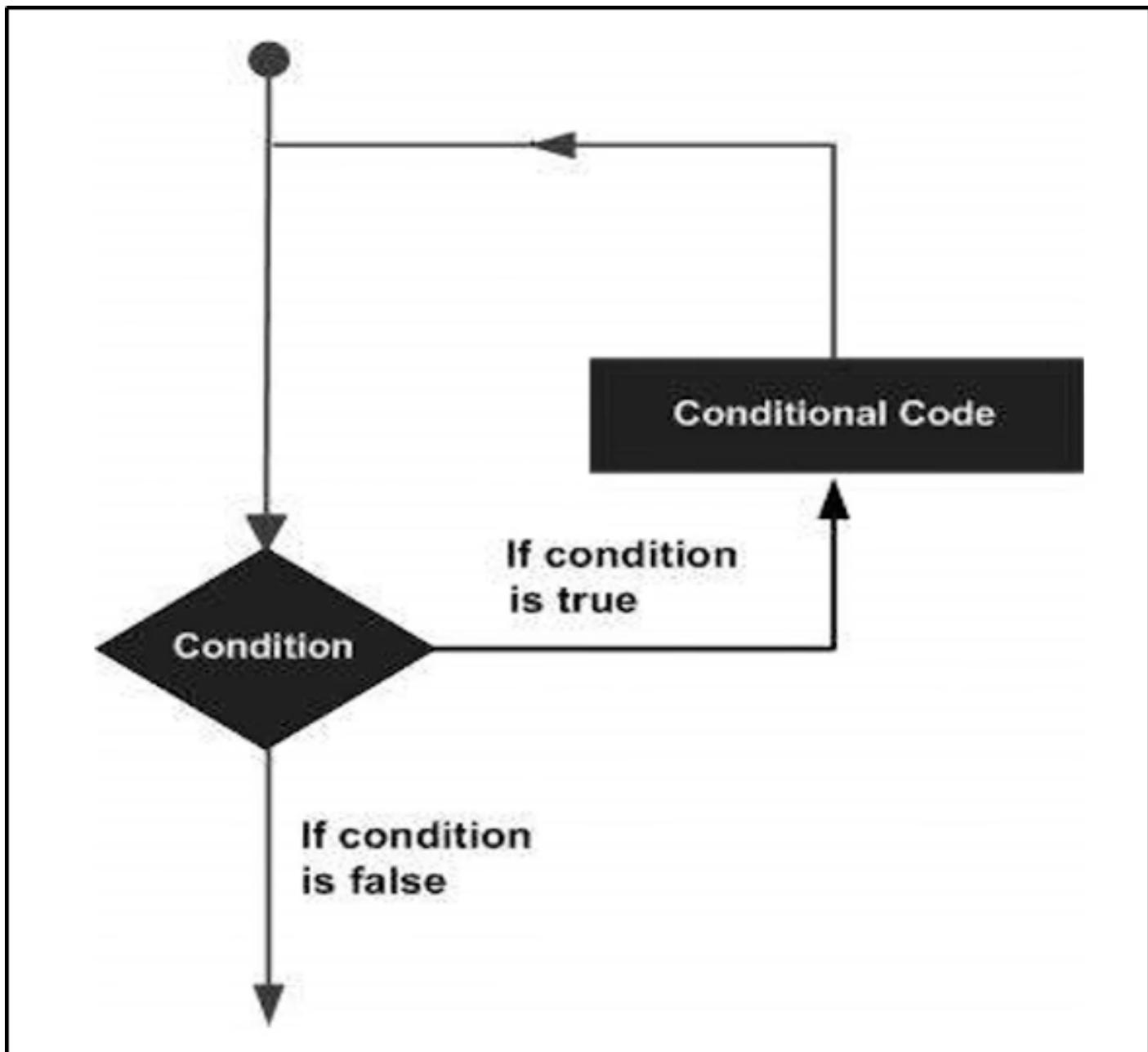


#### ◊ If statements:

| Statement            | Description                                                                                                         |
|----------------------|---------------------------------------------------------------------------------------------------------------------|
| if statement         | An if statement consists of a boolean expression followed by one or more statements.                                |
| if...else statement  | An if statement can be followed by an optional else statement, which executes when the boolean expression is false. |
| nested if statements | You can use one if or else if statement inside another if or else if statement(s)                                   |

- **Repeating Code:**

- ◊ There may be a situation, when you need to execute a block of code several number of times:
  - the statements are executed sequentially:
    - the first statement in a function is executed first, followed by the second and so on.
- ◊ A **loop statement** allows us to execute a statement or a group of statements multiple times.
- ◊ Loop control statements change execution from its normal sequence:
  - when execution leaves a scope, all automatic objects that were created in that scope are destroyed (break and continue).
    - this is talking about private variables that are inside the loop, you're not gonna be able to access those variables from outside the loop.
- ◊ A loop becomes an infinite loop if a condition never becomes false:
  - the **for loop** is traditionally used for this purpose.
    - sometimes we'll need to do this, for example:
      - when you're monitoring something.
      - when you're listening for something.
- ◊ Loop data flow map:
  -



◊ Loops:

| Loop Type              | Description                                                                                                                             |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <b>while loop</b>      | It repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body. |
| <b>for loop</b>        | It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.                            |
| <b>do...while loop</b> | It is similar to a while statement, except that it tests the condition at the end of the loop body                                      |
| <b>nested loops</b>    | You can use one or more loop inside any another while, for or do..while loop.                                                           |



# 48. If Statements

- **Overview:**

- ◊ The C programming language provides a general decision-making capability in the form of an **if statement**.

- ◊ What it looks like in pseudo code:

```
if (expression)
 program statement
```

- the expression has to evaluate to **true** or **false**.

- ◊ Translating a statement such as:

- "if it's not raining, then I will go swimming" into the C language is easy:

```
if (it is not raining)
 I will go swimming
```

- The if statement is used to stipulate execution of a program statement/s based upon specified conditions:

- I will go swimming if it's not raining.

- ◊ The curly brackets **{ }** are required for compound statements inside the if block.

- **If statement code example:**

```
int score = 95;
int big = 90;

// simple if statement, no brackets.
if (score > big)
 printf("Jackpot!\n");

// compound if statement, with brackets
if (score > big) {
 score++;
 printf("You win!\n");
}
```

- **If with an else:**

- ◊ You can extend the if statement with a small addition that gives you a lot more flexibility:

- example:

```
If the rain today is worse than the rain yesterday,
I will take my umbrella.
Else
I will take my jacket.
Then I will go to work.
```

◊ This is exactly the kind of decision making the if-else statement provides.

◊ Pseudo code example:

```
if (expression)
 Statement1;
else
 Statement2;
```

◊ If with an else code example:

```
// Program to determine if a number is even or odd

#include <stdio.h>

int main() {
 int number_to_test, remainder;

 printf("Enter your number to be tested: ");
 scanf("%i", &number_to_test);

 remainder = number_to_test % 2;

 if (remainder == 0) {
 printf("The number is even.\n");
 } else {
 printf("The number is odd.\n");
 }
 return 0;
}
```

• Else if:

◊ You can handle additional complex decision making by adding an if statement to your else clause:

▪ example:

```
if (expression 1)
 program statement 1
else
 if (expression 2)
 program statement 2
 else
 program statement 3
```

◊ The above extends the if statement from a two-valued logic decision to a three-valued logic decision:

▪ formatted using the else if construct.

◊ We can re-write the previous example to use an else-if statement:

▪ example:

```

if (expression 1)
 program statement 1
else if (expression 2)
 program statement 2
else
 program statement 3

```

### ◊ Else if code example:

```

// Program to implement the sign function

#include <stdio.h>

int main(void) {
 int number, sign;

 printf("Please type in a number: ");
 scanf("%i", &number);

 if(number < 0) {
 sign -= 1;
 } else if (number == 0) {
 sign = 0;
 } else { // must be positive
 sign = 1;
 }

 printf("\nSign = %i\n", sign);

 return 0;
}

```

### • Nested if-else statement:

- ◊ A nested if-else statement means you can use one if or else if statement inside another if or else if statement(s).

### ◊ example:

```

if(boolean_expression1) {
 // executes when the boolean_expression1 is true
 if(boolean_expression2) {
 // executes when the boolean_expression2 is true
 }
}

```

### ◊ code example:

```

if (gameIsOver == 0)
 if (playerToMove == YOU)
 printf("Your move\n");
 else
 printf("My move\n");
else
 printf("The game is over\n");

```

- **The conditional operator (ternary statement):**

- ◊ There's also a third type of way to ask questions in your program, its called:
  - the **conditional operator**
- ◊ The **conditional operator** is a unique operator:
  - unlike all other operators in C.
  - most operators are either unary or binary operators.
  - is a ternary operator (**takes three operands**).
- ◊ The two symbols that are used to denote this operator are the question mark (?) and the colon (:).
  - ◊ The first operand is placed before the ?
    - the second between the ? and the :
    - and the third after the :
      - example:
        - ⇒ condition ? expression1 : expression2
          - **condition** → the condition, the boolean expression in the earlier examples.
          - **expression1** → the body of the if statement, if the if statement were true.
          - **expression2** → if the condition evaluates to false, you'd be executing that body of code.
    - ◊ This operator is equivalent to an **if-else** statement.
    - ◊ This is short-hand notation for an **if-else** statement.
    - ◊ The conditional operator evaluates to one of two expressions, depending on whether a logical expression evaluates to true or false.
    - ◊ Notice how the operator is arranged in relation to the operands:
      - the ? character follows the logical expression, condition.
      - on the right of the ? are two operands, expression1 and expression2, that represent choices.
        - the value that results from the operation will be the value of expression1 if condition evaluates to true, or the value of expression2 if condition evaluates to false.
    - ◊ example:
      - `x = y > 7 ? 25 : 50`

- results in x being set to 25 if y is greater than 7, or to 50 otherwise.

- ◊ the previous code example is the same as:

```
if (y > 7)
 x = 25;
else
 x = 50;
```

- ◊ An expression for the maximum or minimum of two variables can be written very simple using the conditional operator.

# 49. Switch Statement

- An alternative to the if statement, but usually used only specific situations.

- **Overview:**

- ◊ The conditional operator and the if else statements make it easy to write programs that choose between two alternatives.
- ◊ However many times a program needs to choose one of several alternatives:
  - you can do this by using if else if ... else.
  - tedious, prone to errors.
- ◊ Everything you can do in a switch you can do in an if, but it's cleaner and a lot more efficient.
- ◊ When the value of a variable is successively compared against different values use the switch statement:
  - more convenient and efficient.
- ◊ If the boolean expressions are a bit more complicated you can't use a switch statement:
  - you can only use a switch statement when you're comparing against a value, like a constant, the expression cannot be that complicated.

- **Switch statement syntax:**

```
switch (expression) {
 case value1:
 program statement
 ...
 break;
 case valueN:
 program statement
 program statement
 ...
 break
 default:
 program statement
 ...
 break;
}
```

- **Switch statement details:**

- ◊ The expression enclosed within parentheses is successively compared against the values: value1, value2, ..., valueN:

- cases must be simple constants or constant expressions.
- ◊ If a case is found whose value is equal to the value of expression then the statements that follow the case are executed:
  - when more than one statement is included, they do not have to be enclosed within braces.
- ◊ The break statement signals the end of a particular case and causes execution of the switch statement to be terminated:
  - include the break statement at the end of every case.
  - forgetting to do so for a particular case causes program execution to continue into the next case.
- ◊ The special optional case called **default** is executed if the value of expressions does not match any of the case values:
  - same as a "fall through" else.

- **Switch case code example:**

```
#include <stdio.h>
#include <stdlib.h>

int main() {
 enum Weekday {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
Sunday};
 enum Weekday today = Monday;

 switch(today) {
 case Sunday:
 printf("Today is Sunday");
 break;
 case Monday:
 printf("Today is Monday");
 break;
 case Tuesday:
 printf("Today is Tuesday");
 break;
 default:
 printf("Default");
 break;
 }

 return 0;
}
```

- **Another switch statement code example:**

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
```

```

float value1, value2;
char operator;

printf("Type in your expression.\n");
scanf("%f %c %f", &value1, &operator, &value2);

switch (operator) {
 case '+':
 printf("%.2f\n", value1 + value2);
 break;
 case '-':
 printf("%.2f\n", value1 - value2);
 break;
 case '*':
 printf("%.2f\n", value1 * (value2));
 break;
 case '/':
 if (value2 == 0){
 printf("Division by zero\n");
 } else {
 printf("%.2f\n", value1 / value2);
 }
 break;
 default:
 printf("Unknown operator\n");
 break;
}
return 0;
}

```

## • goto Statement:

- ◊ The **goto** statement is available in C:
- ◊ It's used to jump to a direct line of code.
- ◊ It has two parts:
  1. the **goto** keyword.
  2. and a **label name**.
  - label is named following the same convention used in naming a variable.
- ◊ Naming example:
  - **goto part2;**
  - for the above there must be another statement bearing the part2 label.
    - to jump to a line of code you have to label that line of code.
- ◊ You should never need to use the **goto** statement:
  - if you have a background in older versions of **FORTRAN OR BASIC**, you might have developed programming habits that depend on using **goto**.

- ◊ Problems associated with it:
  - maintenance
  - it's very problematic, it's hard to follow code when you're jumping all over the place.

- ◊ **goto** example:

- syntax:

```
goto label;
.
.
.
label: statement
```

- code example:

```
top: ch = getchar(); // top is the
label
.
.
.
if (ch != 'y')
 goto top;
```

## **50. (Challenge) Determine Amount of Weekly Pay**

- In this challenge you are to create a C program that calculates your weekly pay.
- Requirements:
  - The program should ask the user to enter the **number of hours worked in a week via the keyboard**.
  - The program should display as output the gross pay, the taxes and the net pay.
    - net pay = gross pay - taxes
  - The following assumptions should be made:
    - Basic pay rate = \$12.00/hr
    - Overtime (in excess of 40 hours) = time and a half = \$18.00/hr
      - an **if/else** statement
      - keep in mind that this only applies for the hours after the first 40
  - Tax rate:
    - 15% of the first \$300
    - 20% of the next \$150 (\$301 (>\$300) to \$450 (< \$451))
    - 25% of the rest (more than \$450)
      - if statements
- ◊ you're gonna have to write a lot of decisions.
- You will need to utilize if/else statements.

# My Code

```
/*
AUTHOR: JFITECH
PURPOSE: THIS PROGRAM CALCULATES AND PRINTS OUT THE AMOUNT OF WEEKLY PAY
DATE: MARCH 6TH, 2022
*/

#include <stdio.h>
#include <stdlib.h>

int main() {
 // DECLARE VARIABLES
 float hoursWorked, basicPay = 12.00, grossPay, grossPayOvertime,
overtimePay = 18.00, fifteenTaxRate;
 float twentyTaxRate, netPay, twentyFiveTaxRate;

 // REQUEST THE AMOUNT OF HOURS WORKED FROM THE USER
 printf("\nPlease enter the amount of hours worked this week: ");
 scanf("\n%f", &hoursWorked);

 // CALCULATE GROSSPAY
 if (hoursWorked < 40.1f) {
 grossPay = basicPay * 40;
 } else if (hoursWorked > 40.0f) {
 grossPay = basicPay * 40;
 grossPayOvertime = overtimePay * (hoursWorked - 40.0f);
 grossPay += grossPayOvertime;
 }

 // CALCULATE TAXES AND NET PAY
 if (grossPay < 301) {
 fifteenTaxRate = grossPay * 0.15f;
 netPay = grossPay - fifteenTaxRate;
 } else if (grossPay > 300 && grossPay < 451) {
 fifteenTaxRate = grossPay * 0.15f;
 twentyTaxRate = (grossPay - 300) * 0.20f;
 netPay = (grossPay - fifteenTaxRate) - twentyTaxRate;
 } else if (grossPay > 450) {
 fifteenTaxRate = grossPay * 0.15f;
 twentyTaxRate = (grossPay - 300) * 0.20f;
 twentyFiveTaxRate = (grossPay - 450) * 0.25f;
 netPay = ((grossPay - fifteenTaxRate) - twentyTaxRate) -
twentyFiveTaxRate;
 }

 printf("\nYour pay is: %.2f", grossPay);
 printf("\nYour net pay is: %.2f", netPay);

 return 0;
}
```

# **51. (Demonstration) Determine the Amount of Weekly Pay**

- Here we're gonna use a lot of constants.
- We're gonna create 5 constants for all of our data in this program.

- **To Define Pre-processor Constants:**

- ◊ We're gonna use something that we haven't seen yet called a **define**:
  - It's a pre-processor directive to create constants.
- ◊ The way that we know it's a pre-processor directive is because it's got the following symbol:
  - **#**
- ◊ The way that the **#define** works is the following:
  - **#define** keyword
  - then you'll provide a name in all caps:
    - **#define PAYRATE**
    - ⇒ and then a **space** and a **value**:
      - **#define PAYRATE 12.00**
      - ◊ the data that you assign can be:
        - **integers**
        - **floating points**
        - **strings**
  - ◊ We can declare the **#define** constants at the top before the **main**.
  - It's never a good idea to have magic numbers inside variables.
    - ◊ for example:
      - let's say the **OVERTIME** changed to 45, we'd only have to change in one spot.
        - if we were to use the magic numbers, we'd have to change it everywhere we're using 40.
  - We should declare the local variables at the top of the main function.

# Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>

// We're defining constants
#define PAYRATE 12.00
#define TAXRATE_300 0.15
#define TAXRATE_150 0.20
#define TAXRATE_REST 0.25
#define OVERTIME 40

int main() {
 // declare variables
 int hours = 0;
 double grossPay = 0.0;
 double taxes = 0.0;
 double netPay = 0.0;

 printf("Please enter the number of hours worked this week: ");
 // get the user input
 scanf("%d", &hours);

 // calculate the gross pay
 if (hours <= 40)
 grossPay = hours * PAYRATE;
 else {
 grossPay = 40 * PAYRATE;
 double overTimePay = (hours - 40) * (PAYRATE * 1.5);
 grossPay += overTimePay;
 }

 // calculate the taxes
 if (grossPay <= 300) {
 taxes = grossPay * TAXRATE_300;
 } else if (grossPay > 300 && grossPay <= 450) {
 taxes = 300 * TAXRATE_300;
 taxes += (grossPay - 300) * TAXRATE_150
 }
 else (grossPay > 450) {
 taxes = 300 * TAXRATE_300;
 taxes += 150 * TAXRATE_150;
 taxes += (grossPay - 450) * TAXRATE_REST
 }

 // calculate net pay
 netPay = grossPay - taxes;

 // display output
 printf("Your gross pay this week is: %.f\n", grossPay);
 printf("Your taxes this week are: %.f\n", taxes);
 printf("Your net pay this week is: %.f\n", netPay);

 return 0;
}
```

# 52. For Loop

## • Overview:

- ◊ Let's discuss repeating code:
  - the C programming language has a few constructs specifically designed to handle these situations when you need to use the same code repeatedly.
  - you can repeat a block of statements until some condition is met or a specific number of times.
  - repeating code without a condition is forever/infinite loop.
- ◊ The number of times that a loop is repeated can be controlled simply by a count:
  - repeating the statement block a given number of time (counter controlled loop).
- ◊ The number of times that a loop is repeated can depend on when a condition is met:
  - for example:
    - the user entering "**quit**".

## • for loop:

- ◊ You typically use the for loop to execute a block of statements a given number of times.
- ◊ It has the **for** keyword.
- ◊ It's typically used for counter controlled loop:
  - but it doesn't need to be.
    - it can also be used for sentinel controlled loop.
- ◊ If you want to display the numbers from 1 to 10:
  - instead of writing ten statements that call **printf()**, you would use a for loop.
    - code example:

```
for (int count = 1; count <= 10; ++count) {
 printf("%d", count);
}
```
- ◊ The **for loop** operation is controlled by what appears between the parentheses that follow the keyword **for**:
  - the three control expressions that are separated by semicolons control the operation of the loop.
- ◊ The action that you want to repeat each time the loop repeats is the block containing the statement that calls **printf()** (body of the loop):
  - for single statements you can omit the braces.

- ◊ The general pattern for the **for** loop is:

```
for (starting_condition; continuation_condition; action_per_iteration)
 loop_statement;
```

- ◊ The statement to be repeated is represented by **loop\_statement**:
  - could equally well be a block of several statements enclosed between braces.
- ◊ The **starting\_condition** usually (but not always) sets an initial value to a loop control variable:
  - the loop control variable is typically a counter of some kind that tracks how often the loop has been repeated.
  - can also declare and initialize several variables of the same type here with the declarations separated by commas.
    - depending on the c compiler that you use, you may not be able to declare those variables inside the for statement.
      - if you use anything below the **C99** compiler, you have to declare the variables outside of the **for** statement and then initialize them where the starting condition is at.
    - variables will be local to the loop and will not exist once the loop ends.
- ◊ The **continuation\_condition** is a logical expression evaluating to true or false:
  - determines whether the loop should continue to be executed.
  - as long as this condition has the value true, the loop continues.
  - typically checks the value of the loop control variable.
  - you can put any logical or arithmetic expression here as long as you know what you are doing.
- ◊ The **continuation\_condition** is tested at the beginning of the loop rather than at the end:
  - means that the **loop\_statement** will not be executed at all if the **continuation\_condition** starts out as false.
  - this is referred to as a pre-test loop.
- ◊ The **action\_per\_iteration** is executed at the end of each loop iteration:
  - usually an increment or decrement of one or more loop control variables.
  - can modify several variables here, just need to use commas to separate.
- ◊ Chart:
  -

This expression executes once, when the loop starts.  
It declares *count* and initializes it to 1.

```
for(int count = 1 ; count <= 10 ; ++count)
{
 printf(" %d", count);
}
```

This expression is executed at the end of every loop cycle.  
It increments *count*.

This expression is evaluated at the beginning of each loop cycle. If it is *true*, the loop continues, and if it is *false*, the loop ends.

#### ◊ Brief code example:

```
for (int i = 1, j = 2, i <= 5; ++i, j = j + 2)
 printf(" %5d", i*j);
```

- the output produced by this fragment will be the values 2, 8, 18, 32 and 50 on a single line.

#### ◊ Code example (flexibility):

```
/*
 * AUTHOR: JFITECH
 * PURPOSE: THIS PROGRAM DEMONSTRATES A FOR LOOP IN C
 * DATE: MARCH 7TH, 2022
 */

#include <stdio.h>
#include <stdlib.h>

int main(){
 // DECLARE VARIABLES
 unsigned long long sum = 0; // STORES THE SUM OF INTEGERS
 unsigned int count = 0; // THE NUMBER OF INTEGERS TO BE SUMMED

 // READ THE NUMBER OF INTEGERS TO BE SUMMED
 printf("\nEnter the number of integers you want to sum: ");
 scanf(" %u", &count);

 // SUM INTEGERS FROM 1 TO COUNT
 // for(unsigned int i = 1; i <= count; ++i);
 // sum += i;
```

```
// OR
for(unsigned int i = 1; i <= count; sum += i++);
printf("\nTotal of the first %u numbers is %u\n", count, sum);

return 0;
}
```

- **infinite loop:**

- ◊ You have no obligation to put any parameters in the for loop statement.

- code example:

```
- for(; ;) {
 /* statements */
}
```

- ◊ The condition for continuing the loop is absent, the loop will continue indefinitely:

- sometimes useful for monitoring data or listening for connections.

- ◊ Avoid infinite loops as much as possible.

## 53. While and Do-While

- **While loop:**

- It's a mechanism for repeating a set of statements, allows execution to continue for as long as a specified logical expression evaluates to **true**:

- While this condition is **true**  
keep on doing this

- or

- While you are hungry  
Eat sandwiches

- ◊ Everything you can do in a for loop you can do in a while loop.

- it's just different syntax

- ◊ The general syntax for the while loop is as follows (one statement in body):

- **while**(expression);  
statement1;

- or

- **while**(expression) {  
statement1;  
statement2;  
}

- ◊ The condition for continuation of the while loop is tested at the start (top of the loop):

- pre-test loop.

- ◊ If expression starts out false, none of the loop statements will be executed:

- if you answer the first "No, I'm not hungry", then you don't get to eat any sandwiches at all and you move straight to the coffee.

- ◊ If the loop condition starts out as true, the loop body must contain a mechanism for changing this if the loop is to end.

- otherwise you'll have an infinite loop.

- ◊ **Counter controlled while loop example:**

```
#include <stdio.h>

int main(void){
 int count = 1;

 while (count <= 5) {
 printf("%i\n", count);
 ++count;
 }
}
```

```
 return 0;
}
```

- this is always gonna execute the same number of times, which is 5.
  - if you look at the structure it has the same components as a for loop:

```
⇒ int count = 1;
 • starting_condition
⇒ while (count <= 5)
 • continuation_condition
⇒ ++count;
 • action_per_iteration
```

## ◊ Logic (sentinel) controlled while loop example:

```
int num = 0;
scanf("%d", &num);

while (num != -1) {
 /* loop actions */
 scanf("%d", &num);
}
```

- this loop may execute 100 times or 0 times.
  - this is dependent on what the user enters:
    - ⇒ **num** is what the user enters:
      - if the user never enters -1 then the loop is never gonna exit.

## • do-while loop:

- ◊ In the **while** loop, the body is executed while the condition is true.
- ◊ The **do-while** loop is a loop where the body is executed for the first time unconditionally:
  - always guaranteed to execute at least once.
  - condition is at the bottom (**post-test** loop).
  - also called an **exit controlled** loop.
- ◊ You can do the same thing in a **for loop**.
- ◊ After initial execution, the body is only executed while the condition is true.
- ◊ What it looks like:

```
do
 statement
while (expression);
```

- example:

```

do {
 prompt for password
 read user input
} while(input no equal to password);
→

```

◊ **do-while** loop code example:

```

do {
 scanf("%d", number);
} while (number != 20);

```

- this one may execute 1 time or many times:

→ but never 0

⇒ it all depends on what the user enters, it's always gonna execute at least once.

◊ counter controlled **do-while** loop code example:

```

int number = 4;
do {
 printf("\nNumber = %d", number);
 number++;
} while (number < 4);

```

• **Which loop to use? :**

- ◊ First, decide whether you need a **pre** or **post test** loop:
  - usually it will be a **pre test** loop (**for** or **while**), a bit better option in most cases.
  - it is better to **look before you leap** (or **loop**) than after.
  - **easier to read** if the loop test is **found at the beginning of the loop**.
  - in many cases, it is important that the loop be skipped entirely if the test is not entirely met.

- ◊ If you want it to execute at least once, then use a:

▪ **do while**

- if that's not a condition, then you can use:

→ **for**

⇒ or

→ **while**

- ◊ So should you use a **for** or a **while**:

- a matter of taste, because what you can do with one, you can do with the other.
- to make a **for** loop like a **while**, you can omit the first and third expressions.

```

for (;test;)

```

- is the same as:

- `while (test)`

◊ To make a **while** like a **for**:

- **preface** it with an **initialization** and **include update statements**.

- Code example:

```
initialize;
→ while (test) {
 body;
 update;
}
```

⇒ is the same as:

```
→ for (initialize; test; update)
 body;
```

◊ A **for** loop is appropriate when the loop involves initializing and updating a variable.

◊ A **while** loop is better when the conditions are otherwise.

◊ The instructor usually uses the **while loop** for **logic controlled** loops and the **for loop** for **counter controlled** loops:

- `while (scanf("%l", &num) == 1)`

- `for (count = 1; count <= 100; count++)`

# 54. Nested Loops and Loop Control - Break and Continue

## • Overview:

- ◊ Nested loops:
  - loops inside loops
- ◊ Loop control:
  - it has to do with the **break** and **continue** statements

## • Nested loops:

- ◊ Sometimes you may want to place one loop inside another.
- ◊ You might want to count the number of occupants in each house on a street:
  - step from house to house and for each house you count the number of occupants.
    - going through all the house could be an outer loop, and for each iteration of the outer loop you would have an inner loop that counts the occupants.

### ◊ Example:

```
for(int i = 1; i <= count;i) {
 sum = 0; // initialize sum for the outer loop

 // Calculates sum of integers from 1 to i
 for(int j = 1;j <= i; ++j)
 sum += j;

 printf("\n%d\t%d", i, sum); // Output sum of 1 to i
}
```

- ◊ It's not always a **for loop** inside a **for loop**:
  - you can have a **while loop** inside of a **for loop**:
    - example:

```
for(int i = 1; i <= count; ++i) {
 sum = 1; // Initialize sum for the inner loop
 j = 1; // Initialize integer to be added
 printf("\n1");

 // Calculate sum of integers from 1 to i
 while(j < i){
 sum += ++j;
 printf(" + %d", j); // Output +j - on the same line
 }
 printf(" = %d", sum); //Output = sum
}
```

⇒ if we didn't modify **++j** we would have an infinite loop for the **while loop**.

- it's doing the same output as the example before, it's just using a while loop as a nested loop instead of the for loop.

- **Continue Statements:**

- ◊ Sometimes a situation arises where you do not want to end a loop, but you want to skip the current iteration.

- up until this point when we have talked about loops:

- we've always executed the loop a number of times, based on the exit condition, the boolean expression that evaluates to false.

- however there are times when you want to specifically end a loop early:

- ⇒ or you want to continue on the next iteration of the loop without doing anything

- ◊ To continue without doing anything (skipping) you can use the **continue statement** to do this:

- ◊ The continue statement in the body of a loop does this:

- all you need to do is use the keyword "**continue;**" in the body of the loop.

- ◊ An advantage of using the **continue statement** is that it can sometimes eliminate nesting or additional blocks of code:

- it can enhance readability when the statements are long or are deeply nested already.

- it can eliminate a lot of **if** statements inside your blocks of loops.

- ◊ Anything below the continue will not be executed if it's inside the body of the loop.

- so **continue** a lot of times will wanna be at the top of the loop, based on some condition.

- it's usually wrapped in an **if** statement:

- "if this happens (any special situation), I don't wanna execute any of the code in the body of the loop"

- ⇒ just skip this iteration and check the next time.

- ◊ So if you find yourself doing a lot of nesting, you may wanna use a **continue statement** instead:

- or if you find that you have a lot of if statements, they're doing certain things inside the body of the loop, maybe you wanna do a **continue**.

- ◊ **Don't use a continue if it complicates rather than simplifies code.**

- ◊ Example:

```

enum Day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

for(enum Day day = Monday; day <= Sunday; ++day) {
 if(day == Wednesday)
 continue

 printf("It's not Wednesday!\n");
 /* Do something useful with day*/
}

```

- remember:

→ an enum under the hood is an integer, so everyday is equal to an int value:

⇒ Monday = 0

⇒ Tuesday = 1

⇒ etc.

- on this loop if it's Wednesday, we wanna skip, we don't wanna do anything:

→ it's gonna skip the **printf()** line.

→ it's gonna go back up to the top of the loop.

→ and it's gonna execute the next iteration.

- this is a common use case of using a **continue statement**.

## • Break Statement:

◊ Normally, after the body of a loop has been entered, a program executes all the statements in the body before doing the next loop test:

- we learned how the **continue statement** works.
- another statement named **break** alters this behavior.

◊ If you want to exit early in the loop.

◊ You can also use the **break** keyword inside of a loop:

- what this will do is that:
  - it'll just jump out of the loop.

◊ The **break statement** cause the program to immediately exit from the loop it is executing:

- statements in the loop are skipped and execution of the loop is terminated.
- if the **break statement is inside nested loops, it affects only the innermost loop containing it.**
- use the keyword "**break;**".

◊ Basically what you're doing is jumping out of the loop.

◊ **Break** is often used to leave a loop when there are two separate reasons to leave.

- when there are **two conditions** to exit the loop we use a **break statement**.

- ◊ **Break** is also used in switch statements.

- ◊ Code example:

```
while (p > 0) {
 printf("%d\n", p);
 scanf("%d", &q)
 while(q > 0) {
 printf("%d\n", p*q);
 if(q > 100)
 break; // break from inner loop
 scanf("%d", &q);
 }
 if(q > 100)
 break; // break from outer loop
 scanf("%d", &p);
}
```

- if you didn't add the second **brake statement** you'd never jump out of the second outer loop.

- Provide good comments when you use **break** or **continue** statements.

## **55. (Challenge) Guess the Number**

- In this challenge you are to create a "Guess the Number" C program.

- **Requirements:**

Your program will generate a random number from **0** to **20**

You will then ask the user to guess it:

the user should only be able to enter number from **0-20**

if the user tries to enter something display an **error**

do while loop

The program will indicate the user if their guess is too high or too low.

**high**

**too high**

**too low**

If they're correct, you'll display that they're **correct** and they'll win the game

The player wins the game if they can guess the number within five tries.

after five time they lose the game.

If they enter something and if they have less than 5 tries, you're gonna tell them:

**too high**

- or

**too low**

You're gonna have to use **if statements** and **while loops**

The best way to approach this problem is by writing down on paper what you're gonna do on the program:

where the if statements go.

where the loops go.

write it in pseudo code.

An outer loop that asks the user to enter numbers until either:

they have successfully guessed the correct number.

- or

they have succeeded their number of tries.

An inner loop where they ask the user to enter a valid number between 0 and 20, if it's not valid:

it keeps asking the user to enter a valid number.

- If statements to check whether the number is:
  - too low**
  - 
  - too high**
  - or
  - equal**
  
- Because you have multiple exit conditions for your outer loop:
  - whether you guessed correctly or not:
    - this might be a good case to use a **break statement**
  - we use a **break** statement when we have multiple exits.
  
- You could have something inside your outer loop that says:
  - if they guess correctly jump out.
  - and then the condition in the while outer (while/for) loop itself to exit would be:
    - whether the number of tries is greater than 5

### • Sample Output:

```
This is a guessing game.
I have chosen a number between 0 and 20 which you must guess.

You have 5 tries left.
Enter a guess: 12
Sorry, 12 is wrong. My number is less than that.

You have 4 tries left.
Enter a guess: 8
Sorry, 8 is wrong. My number is less than that.

You have 3 tries left.
Enter a guess: 4
Sorry, 4 is wrong. My number is less than that.

You have 2 tries left.
Enter a guess: 2

◇ Congratulations. You guessed it!
```

### • To generate a random number:

- ◊ from 0 to 20:
  - include the correct system libraries:

```
#include <stdlib.h> // will allow you to use the srand function, which
 // allows you to generate random numbers
#include <time.h> // allows you to provide the seed for the random
 // number, using the current time
```

- ◊ So when we include libraries outside of our program, it just means we wanna do additional functionality

- ◊ Then we're gonna create a time variable inside of our main function called:

- `time_t t;`

- ◊ Initialize the random number generator:

- `srand((unsigned) time(&t));`

- ◊ Get the random number (0-20) and store in an int variable:

- ```
int randomNumber = rand() % 21; // because we
want it to be between 0 and 21, we hardcode the %
21
                                // the rand() is
a function.
```

- this is the number that we're gonna compare against to exit the loop, whether or not they guessed correctly.

- it's also the number that you're gonna compare against to determine whether you print **too low** or **too high**.

My Code

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 6 SOLUTION
DATE: MARCH 10TH, 2022
*/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>

// DECLARE CONSTANTS

int main() {
    // DECLARE VARIABLES
    int guessNumber;

    // GENERATE RANDOM NUMBER
    time_t t;
    srand((unsigned) time(&t));
    int randomNumber = rand() % 21;

    // INITIAL MESSAGE
    printf("\n\nThis is a guessing game.\nI have chosen a random number between
0 and 20 which you must guess.\n");

    // REQUEST USER INPUT
    printf("\nEnter a guess: ");
    scanf("%d", &guessNumber);

    for(int remainingGuesses = 5; remainingGuesses > 0; --remainingGuesses) {
        if (guessNumber < 0 && guessNumber < 20) {
            printf("Error: please enter a number between 0 and 20.\n");
            continue;
        } else {
            do {
                printf("\n\nYou have %d tries left\n", remainingGuesses);
                printf("\nEnter a guess: ");
                scanf("%d", &guessNumber);

                if (guessNumber == randomNumber) {
                    printf("Congratulations. You guessed it!");
                    break;
                } else if (guessNumber > randomNumber) {
                    printf("Sorry, %d is wrong. My number is less than that.", guessNumber);
                } else if (guessNumber < randomNumber) {
                    printf("Sorry, %d is wrong. My number is more than that.", guessNumber);
                } else {
                    printf("No more guesses remaining. You lose.\nThe correct number was %d", randomNumber);
                    break;
                }
            } while(guessNumber > -1 && guessNumber < 21);
        }
    }
    return 0;
}
```

My Code 2

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 6 SOLUTION
DATE: MARCH 10TH, 2022
*/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>

// DECLARE CONSTANTS

int main() {
    // DECLARE VARIABLES
    int guessNumber;

    // GENERATE RANDOM NUMBER
    time_t t;
    srand((unsigned) time(&t));
    int randomNumber = rand() % 21;

    // INITIAL MESSAGE
    printf("\n\nThis is a guessing game.\nI have chosen a random number between
0 and 20 which you must guess.\n");

    // REQUEST USER INPUT
    for (int remainingGuesses = 5; remainingGuesses < 6 && remainingGuesses > 0;
remainingGuesses--) {
        do {
            printf("\nEnter a guess: ");
            scanf("%d", &guessNumber);

            if(guessNumber < 0 && guessNumber > 20) {
                printf("Error: your number must be between 0 and 20.");
                continue;
            }
            else if(guessNumber == randomNumber) {
                printf("\nCorrect, you guessed it! The number is %d\n\n",
randomNumber);
                break;
            } else {
                printf("\nSorry, %d is wrong. Remaining guesses: %d\n\n",
guessNumber, remainingGuesses);
                printf("=====\\n");
            }
            //continue;
        } while(guessNumber > -1 && guessNumber < 21);
        break;
    }

    return 0;
}
```

56. (Demonstration) Guess the Number

- We can use the **return** statement instead of the **break** statement:
 - ◊ this won't print out the last message on the instructor's code.

Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // for random number generator seed

int main()
{
    int randomNumber = 0;
    int guess = 0;
    int numberOfGuesses;
    time_t t;

    // Initialization of random number generator
    srand((unsigned) time(&t));

    // get the random number
    randomNumber = rand() % 21;

    printf("\nThis is a guessing game.");
    printf("\nI have chosen a number between 0 and 20, which you must guess.\n");
    printf("You have %d tries left.", numberOfGuesses, numberOfGuesses == 1 ? "y" : "ies");
    printf("\nEnter a guess: ");
    scanf("%d", &guess);

    if(guess == randomNumber)
    {
        printf("\nCongratulations. You guessed it!\n");
        return;
    }
    else if(guess < 0 || guess > 20) // checking for an invalid guess
        printf("I said the number is between 0 and 20.\n");
    else if(randomNumber > guess)
        printf("Sorry, %d is wrong. My number is greater than that.\n", guess);
    else if(randomNumber < guess)
        printf("Sorry, %d is wrong. My number is less than that.\n", guess);
    printf("\nYou have had five tries and failed. The number was %d\n", randomNumber);

    return 0;
}
```

Section 8: Arrays

57. Creating and using Arrays

- An array allows you to store many different values in a single variable.
- They help you simplify and make your program a lot more efficient.

- **Arrays:**

- ◊ It is very common to in a program to store many data values of a specified type.
 - in a sports program, you might want to store the scores for all games or the scores for each player:
 - you could write a program that does this using a different variable for each score:
 - if there are a lot of games to store then it is very tedious.
 - ⇒ using an array will solve this problem.
- ◊ Arrays allow you to group values together under a single name:
 - you do not need separate variables for each item of data.
- ◊ An array is a fixed number of data items that are all of the same type.

- ◊ **Array Restrictions:**

- the only one issue with an array is that you can't change its size.
- you can't store multiple types of data.

- **Declaring an Array:**

- ◊ The data items in array are referred to as elements.
- ◊ The elements in an array have to be the same type (**int**, **long**, **double**, etc.):
 - you cannot "mix" data types, no such thing as a single array of **ints** and **doubles**.
- ◊ Declaring to use an array in a program is similar to a normal variable that contains a single value:
 - difference is that you need a **number between** square brackets [] following the name.
- ◊ Array declaration example:
 - `long number[10];`
 - the number inside the brackets are specifying the size.

- ◊ The **number between square brackets** defines **how many elements the array contains**:

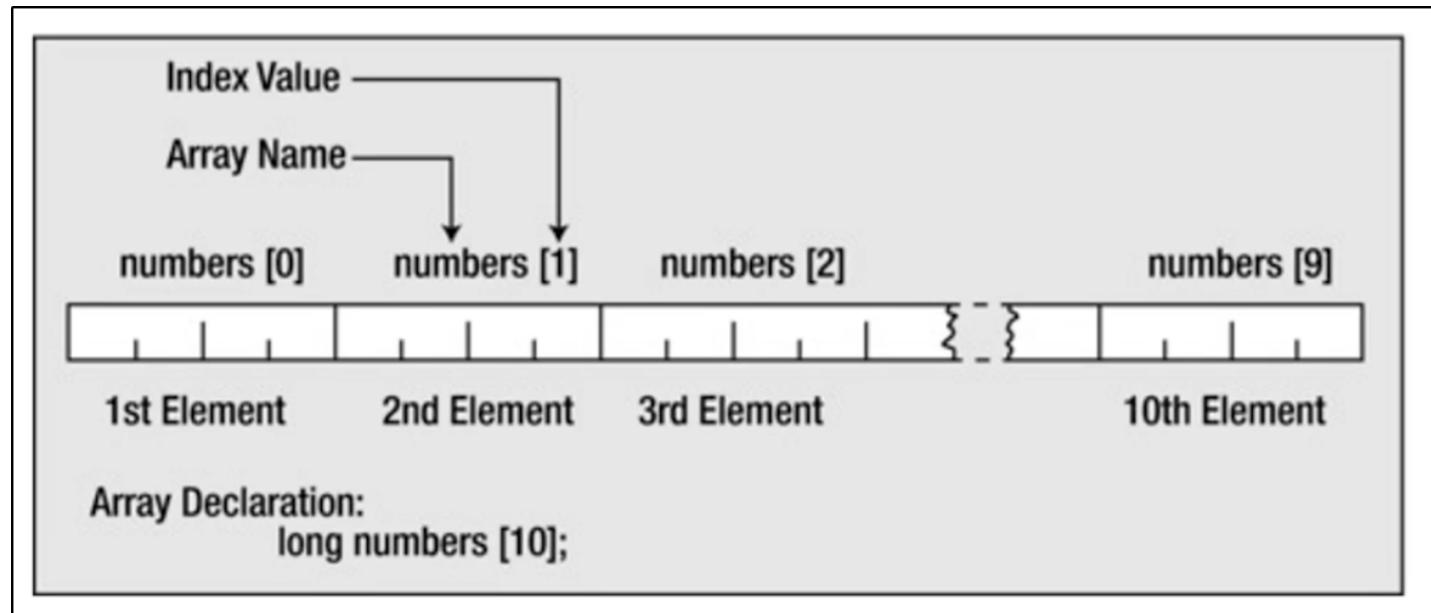
- called the **size of the array** or the **dimension**.

- **Accessing an Array's Elements:**

- ◊ Each of the data items stored in a array is accessed by the same name:
 - once you've declared an array:
 - you're then wanna start using it.
 - you're gonna want to start accessing the data inside of it , or writing data to it.
 - in order to do the previous, you have to understand how the array is stored in memory and how it's actually gonna be accessed.
- ◊ You select a particular element by using an **index (subscript)** value (number) **between square bracket following the array name**.
 - ◊ Index values are sequential integers that start from zero:
 - index values for elements in array of **size 10** would be from **0 - 9**.
 - arrays are zero based:
 - 0 is the first index value for the first element in an array.
 - for an array of 10 elements, index value 9 refers to the last element.
 - ◊ It is a very common mistake to assume that arrays start from one:
 - referred to as the **off-by-one** error.
 - you can use a simple integer to explicitly reference the element that you want to access.
 - to access the fourth value in an array, you use the following expression:
 - `arrayName[3];`
 - ◊ You can also specify an index for an array element by an expression in the square brackets following the array name:
 - the expression must result in an integer value that corresponds to one of the possible index values.
 - ◊ It is very common to use a loop to access each element in an array:
 - code example:

```
for(i = 0; i < 10; ++i){  
    printf("Number is %d", numbers[i]);  
}
```

→ the **i** variable in this case represents each element in the array.



- **Array out of Bounds:**

- ◊ If you use an expression or a variable for an index value that is outside the range for the array, your program may crash or the array can contain garbage data.
 - referred to as an **out of bounds error**.
 - outside of the range array means that it's either:
 - a **negative number**.
 - it's a number that's greater than the size.
 - ⇒ greater than the size - 1
- ◊ The problem with this is that it can cause your program to crash:
 - or even worse it can cause your program to look like it's acting normal, but then it doesn't act normal.
 - this makes it to not produce consistent results, it can cause garbage data.
- ◊ The **compiler cannot check for out of bounds errors** so your program will still compile.
 - the compiler doesn't know if you're assigning something greater or if you're accessing something greater than the size.
- ◊ The **array out of bounds** is a run time error:
 - you're not gonna find this error until you run the program.
- ◊ It'll create a very unpredictable error.

• Assigning Values to an Array:

◊ A value can be stored in an element of an array simply by specifying the array element on the left side of an equal sign:

- code example:

- `grades[100] = 95;`

→ the value 95 is stored in element number 100 of the grades array.

◊ You can also use variables to assign values to an array.

• Code example of using an array:

```
int main(void) {  
  
    int grades[10]; // ARRAY STORING 10 VALUES  
    int count[10]; // NUMBER OF VALUES TO BE READ  
    long sum = 0; // SUM OF THE NUMBERS  
    float average = 0.0f // AVERAGE OF THE NUMBERS  
  
    printf("\nEnter the 10 grades: \n"); // PROMPT FOR THE INPUT  
  
    // READ THE TEN NUMBERS TO BE AVERAGED  
    for(int i = 0; i < count; ++i) {  
        printf("%2u>", i + 1);  
        scanf("%d", &grades[i]); // READ A GRADE, & -> ADDRESS OF OPERATOR  
        sum += grades[i]; // ADD IT TO SUM  
    }  
  
    average = (float)sum / count; // AVERAGE. WE'RE CASTING THE float TYPE  
    VARIABLE TO sum  
    printf("\nAverage of the ten grades entered is: %.2f\n", average);  
  
    return 0;  
}
```



- to avoid an **out of bounds error** you gotta make sure that:

- **i < count**
- when you access the information you don't go out of bounds as well.

58. Initialization

- Assigning specific initial values to an array.

- **Initializing an Array:**

- ◊ You will want to assign initial values for the elements of your array most of the time:
 - defining initial values for array elements (or even variables) makes it easier to detect when things go wrong.

- ◊ There's many many elements in array and there's an specific way that you can initialize those elements:
 - There's different syntax that you can use.

- ◊ Just as you can assign initial values to a variable when they are declared, you can also assign initial values to an array's elements.

- ◊ To initialize an array's values, simply provide the values in a list:
 - values in the list are separated by commas and the entire list is enclosed in a pair of curly braces.

- ◊ Code example:

- `int counters[5] = {0, 0, 0, 0, 0};`

- this declares an array called **counters** to contain five integer values and initializes each of these elements to zero.

- ◊ Another code example:

- `int integers[5] = {0, 1, 2, 3, 4};`

- this declares an array named integers and sets the values of **integers[0]** to 0, **integers[1]** to 1, **integers[2]** to 2, and so on.

- ◊ It is not necessary to completely initialize an entire array.

- ◊ If fewer initial values are specified, only an equal number of elements are initialized:
 - remaining values in the array are set to zero.

- ◊ Code example:

- `float sample_data[500] = { 100.0, 300.0, 500.5 };`

- this previous code example initializes the first three values of **sample_data** to **100.0**, **300.0** and **500.5**, and sets the **remaining 497 elements** to zero.

- **Designated Initializers:**

- ◊ **C99** added a new feature called **designated initializers** and the in **C11** they made it optional:

- it allows you to pick and choose which elements are initialized.
 - depending on what compiler you have, you may or may not be able to do this.

- ◊ You can do this by enclosing an element number in a pair of brackets, specific array elements can be initialized in any order.

- Code example:

- ```
float sample_data[500] = { [2] = 500.5, [1] = 300.0, [0] = 100.0 };
```

→ what this is doing is initializing the **sample\_data** array to **100.0**, **300.0**, **500.5** for the first three values.

- ◊ Another code example:

- ```
int arr[6] = {[5] = 212}; // INITIALIZES arr[5] to 212
```

- **Example of traditional initialization:**

```
#define MONTHS 12

int main(void) {
    int days[MONTHS] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    int index;

    for (index = 0; index < MONTHS; ++index) {
        printf("Month %d has %2d days.\n", index + 1, days[index]);
    }
    return 0
}
```

- **Example of using designated initializers:**

```
#include <stdio.h>
#define MONTHS 12
```



```

int main(void) {
    int days[MONTHS] = {31, 28, [4] = 31, 30, 31, [1] = 29};
    int i;

    for(i = 0; i < MONTHS; ++i){
        printf("%2d %d\n", i + 1, days[i]);
    }

    return 0;
}

```

- **Repeating an initial value:**

- ◊ C does not provide any shortcut mechanisms for initializing array elements.
 - **(at least for C 89 and 99).**
- ◊ No way to specify a repeat count.
- ◊ There's no way to say for example:
 - all of the elements of this array should be the value **55**.
- ◊ If it were desired to initially set all 500 values of **sample_data** to 1, **all 500 would have to be explicitly assigned**.
- ◊ To solve this problem, you will want to initialize the array inside the program using a loop.

- **Initializing all elements to the same value:**

```

int main(void) {
    int array_values[10] = {0, 1, 4, 9, 16};
    int i;

    for(i = 5; i < 10; ++i){
        array_values[i] = i * i; // INITIALIZES ELEMENTS FROM 5 TO 9 TO BE i*i
    }

    for(i = 0; i < 10; ++i){
        printf("array_values[%i] = %i\n", i, array_values[i]); // GOES THROUGH
THE ENTIRE ARRAY AND DISPLAYS THE DATA
    }

    return 0;
}

```

59. Multidimensional Arrays

- **Overview:**

- ◊ The types of arrays that you have been exposed to so far are all **linear arrays**:
 - a single dimension.
- ◊ You can have many dimensions of an array.
- ◊ The C language allows arrays of any dimension to be defined:
 - two dimensional arrays are the most common.
 - this kind of a nested array, an array of an array.
- ◊ There's not many use cases for **3, 4 and 5 dimensional arrays**.
- ◊ **You can visualize a two-dimensional array as a rectangular arrangement like rows and columns in a spreadsheet:**
 - or a matrix.
 - or a table.
- ◊ One of the most natural applications for a two-dimensional array arises in the case of a matrix.
 - there are programs that have this concept of a matrix, specifically in mathematical programs:
- ◊ **Two dimensional array example:**
 - `int matrix[4][5];`
 - this declares an array named **matrix** to be a two dimensional array:
 - consisting of **4 rows** and **5 columns**, for a total of **20 elements**:
 - ⇒ note how **each dimension is between its own pair of square brackets**.
 - this is how you can easily spot how many dimensions an array has:
 - ◊ if you had **3 pair of brackets** it'd **have 3 dimensions**, if it had **4 pairs** it'd **have 4 dimensions** and so on.

- **Initializing a two dimensional array:**

- ◊ Two dimensional arrays can be initialized in the same manner of a one-dimensional array.
- ◊ When listing elements for initialization, the values are listed by row:
 - the difference is that you put the initial values for each row between braces, `{ }`, and then

enclose all the rows between braces.

◊ Code example:

```
int numbers[3][4] = {  
    {10, 20, 30, 40}, // VALUES FOR FIRST ROW  
    {15, 25, 35, 45}, // VALUES FOR SECOND ROW  
    {47, 48, 49, 50} // VALUES FOR THIRD ROW  
};
```

- commas are required after each brace that closes off a row, except in the case of the final row.

→ to access a multidimensional array, you use the **row** and the **column**.

◊ The use of the inner pair of braces is optional, but, should be used for readability.

◊ As with one-dimensional array, it is not required that the entire array be initialized:

- you can just initialize certain areas.

◊ Code example:

```
int matrix[4][5] = {  
    {10, 5, -3},  
    {9, 0, 0},  
    {32, 20, 1},  
    {0, 0, 8},  
};
```

- only initializes the first three elements of each row of the matrix to the indicated values:
 - remaining values are set to 0.
 - in this case the inner pairs of braces are required to force the correct initialization.

• **Designated initializers:**

◊ Subscripts can also be used in the initialization list, in a like manner to single-dimensional arrays.

▪ code example:

```
int matrix[4][3] = {[0][0] = 1, [1][1] = 5, [2][2] = 9};
```

◊ Initializes the three indicated elements of a matrix to the specified values:

- unspecified elements are set to zero by default.
- each set of values that initializes the elements in a row is between braces.
- the entire initialization goes between another pair of braces.
- the values for a row are separated by commas.
- each set of row values is separated from the next set by a comma.

- **Other dimensions:**

- ◊ Everything mentioned so far about two-dimensional arrays can be generalized to three-dimensional arrays and further.
- ◊ You can declare a three-dimensional array this way:
 - `int box[10][20][30];`
 - you might wanna use a three dimensional array if you wanna store X, Y, Z values of some sort.
- ◊ You can visualize a one-dimensional array as a row of data.
- ◊ You can visualize a two-dimensional array as a table of data, matrix or spreadsheet.
- ◊ You can visualize a three-dimensional array as a stack of data tables.
- ◊ Typically you would use three nested loops to process a three-dimensional array, four nested loops to process a four-dimensional array and so on.

- **Initializing an array of more than 2 dimensions:**

- ◊ For arrays of three or more dimensions, the process of initialization is extended.
- ◊ A three-dimensional array will have three levels of nested braces, with the inner level containing sets of initializing values for a row.

- ◊ Code example:

```
int numbers[2][3][4] = {  
    {  
        {10, 20, 30, 40},  
        {15, 25, 35, 45},  
        {47, 48, 49, 50}  
    },  
    {  
        {10, 20, 30, 40},  
        {15, 25, 35, 45},  
        {47, 48, 49, 50}  
    }  
};
```

- here we only initialized the first 2 blocks.

- **Processing elements in a n dimensional array:**

- ◊ You need a nested loop to process all the elements in a multidimensional array:
 - the level of nesting will be the number of array dimensions.
 - each loop iterates over one array dimension.

- ◊ Code example:

```
int sum = 0;

for(int i = 0; i < 2; ++i) {
    for(int i = 0; i < 3; ++j) {
        for(int i = 0; i < 2; ++k) {
            sum += number[i][j][k]
        }
    }
}
```

60. Variable Length Arrays

- So far, all the sizes of an array have been specified using a number.
- This feature was added in **C99**
- Variable length arrays is a bit misleading:
 - ◊ it doesn't mean that you can change the size of the array dynamically:
 - it's still gonna be the same size but now you can use variables.
- The term **variable** in **variable length arrays** does not mean that you can modify the length of the array after you create it:
 - ◊ a **VLA** keeps the same after creation.
- Variable length arrays allow you to specify the size of an array with a variable when creating an array.
- **C99** introduced **variable length arrays** primarily to allow C to become a better language for numerical computing:
 - ◊ **VLAs** make it easier to convert existing libraries of **FORTRAN** numerical calculation routines to C.
- You cannot initialize a **VLA** in its declaration.

- **Valid and invalid declarations of an array:**

```
int n = 5;
int m = 8;
float a1[5]; // YES
float a2[5*2+1]; // YES
float a3[sizeof(int) + 1]; // YES
float a4[-4]; // NO, SIZE MUST BE > 0
float a5[0]; // NO, SIZE MUST BE > 0
float a6[2.5]; // NO, SIZE MUST BE AN INTEGER
float a7[(int)2.5] // YES, TYPECAST FLOAT TO INT CONSTANT
float a8[n]; // NOT ALLOWED BEFORE C99, VLA
◊ float a9[m]; // NOT ALLOWED BEFORE C99, VLA
```

- the last two are not changing their size at run time.

61. (Challenge) Generate Prime Numbers

- In this challenge, you are going to create a program that will find all the prime numbers from 3 to 100.

- We're gonna do it very efficiently so we're gonna use an algorithm that makes you an array.

- **Requirements:**

- There will be no input to the program, just output.

- The output will be each prime number separated by a space on a single line.

- You will need to create an array that will store each prime number as it is generated.

- You can hard code the first two prime numbers (2 and 3) in the primes array.

- You should utilize loops to only find prime numbers up to 100 and a loop to print out the primes array.

- Make the size of the array 100.

- You're gonna need a loop to print them out at the end.

- you're gonna need another loop to try to find the prime numbers.

- **Hints:**

- The criteria that can be used to identify a prime number is that a number is considered prime if it is not evenly divisible by any other previous prime numbers.

- Can use the following as an exit condition in the innermost loop:

- `p / primes[i] >= primes[i]`

- a test to ensure that the value of **p** does not exceed the square root of **primes[i]**.

- A way that you can make it more optimal is that you don't wanna look at even numbers:
 - so if you're going through each number from 3 to 100, you can ignore the even numbers because even numbers are never gonna be prime.

- A for loop that goes from 5 to 100:

- and increment it by 2 (so that you only look for odd numbers).

- And inner for loop inside the previous one, you're gonna want to go from:

- 1 until `p / primes[i] >= primes[i]`

you wanna jump out of the inner loop if you found a prime number.

A way to check for prime number:

if (currentNumber / byAnyPreviousPrime {-> primes array} && previousStatement %
== 0

then you don't have a prime number

My Code

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 7 - PRIME NUMBERS - MY SOLUTION
DATE: MARCH 19TH, 2022
*/

#include <stdio.h>
#include <stdlib.h>

int main() {
    // DECLARE VARIABLES
    int numbers[100];
    int primes[100];

    // FILL OUT THE numbers[] ARRAY WITH THE NUMBERS FROM 0 TO A 100
    for(int i = 1; i < 100; ++i) {
        numbers[i] = i;
        //printf("\nThe numbers are: %d", numbers[i]);
        for(int j = 0; j < 100; ++j) {
            if(numbers[i] % numbers[i - 1] == 0) {
                primes[j] = numbers[i];
                printf("\n\nThe prime numbers are: %d", primes[j]);
            } else
                continue;
        }
    }

    return 0;
}
```

My Code 2

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 7 - PRIME NUMBERS - MY SOLUTION
DATE: MARCH 21ST, 2022
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main() {
    // DECLARE VARIABLES
    int numbers[100];
    int primes[100];
    bool isPrime;

    // FILL OUT THE numbers[] ARRAY WITH THE NUMBERS FROM 0 TO A 100
    for(int i = 1; i < 100; ++i) {
        numbers[i] = i;

        for(int j = 0; j < 100; ++j) {
            if(numbers[i] % numbers[i - 1] == 0) {
                isPrime = false;
            }
            if(isPrime == true) {
                primes[j] = numbers[i];
                printf("\n\nThe prime numbers are: %d", primes[j]);
            }
        }
    }

    return 0;
}
```

62. (*Demonstration*) Generate Prime Numbers

- The way to know that a number is prime:
 - ◊ you divide it by the number and if the result is not zero:
 - it's not a prime

Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main()
{
    int p;
    int i;

    int primes[50] = {0};
    int primeIndex = 2;

    bool isPrime;

    // hardcode prime numbers
    primes[0] = 2;
    primes[1] = 3;

    for(p = 5; p <= 100; p = p + 2)
    {
        isPrime = true;

        for (i = 1; isPrime && p / primes[i] >= primes[i]; ++i)
            if (p % primes[i] == 0)
                isPrime = false;

        if (isPrime == true)
        {
            primes[primeIndex] = p;
            ++primeIndex;
        }
    }

    for (i = 0; i < primeIndex; ++i)
        printf ("%i ", primes[i]);

    printf ("\n");
    return 0;
}
```

63. (Challenge) Create a simple Weather program

- This assignment is gonna deal with multidimensional arrays.
 - ◊ mainly a 2D array
- In this challenge, you are to create a C program that uses a two-dimensional array in a weather program.

- **Requirements:**

- This program will find:
 - total rainfall for each year.**
 - average yearly rainfall.**
 - average rainfall for each month.**

- Input** will be a **2D array** with **hard-coded values for rainfall amounts for the past 5 years:**

- the array should have **5 rows** and **12 columns**.
- rainfall amounts** can be **floating point numbers**.

- The **rows** are gonna represent the **past 5 years**:

- and the **columns** are gonna represent **the months**.

- Example output (can be formatted however you want):
 -

Example output

| YEAR | RAINFALL (inches) |
|------|-------------------|
| 2010 | 32.4 |
| 2011 | 37.9 |
| 2012 | 49.8 |
| 2013 | 44.0 |
| 2014 | 32.9 |

The yearly average is 39.4 inches.

MONTHLY AVERAGES:

| Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 7.3 | 7.3 | 4.9 | 3.0 | 2.3 | 0.6 | 1.2 | 0.3 | 0.5 | 1.7 | 3.6 | 6.7 |

- ✓ A **for loop** and an **inner for loop** to calculate the total rain for **each year**.
 - ✓ and then have a loop to print it out.
- ✓ In a **second loop** with an **inner loop** that **goes through all the months for each year**.
 - ✓ and calculates the monthly averages
- ✓ The **yearly average**:
 - ✓ take the **yearly total** and **divide it by the number of years**.

- **Hints:**

- ✓ Initialize your 2D arrays with hard coded rainfall amounts.
 - ✓ for a 2D array you can initialize rows.
- ✓ Remember, to iterate through a 2D array you will need a nested loop.
- ✓ The key to this solution will be to visualize a 2D array and understand how to iterate through one, via a nested loop.
- ✓ As you are iterating, you can keep a running total (**outer loop iterate by year [row]**,

inner loop iterate by month [column]) to get the total **rainfall for all years**.

To get the **average monthly rainfalls**, iterate through the 2D array by having the **outer loop go through each month** and the **inner loop go through each year**.

My Code

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 8 - SIMPLE WEATHER PROGRAM - MY SOLUTION
DATE: MARCH 21ST, 2022
*/



#include <stdio.h>
#include <stdlib.h>

int main() {
    // DECLARE VARIABLES
    float rainfallValues[5][12] = { // INITIALIZE 2D MATRIX WITH HARD CODED
VALUES
        {11.5, 12.5, 4.3, 7.4, 7.2, 3.2, 7.6, 1.2, 5.4, 4.3, 5.3, 4.5}, // TOTAL = 74.4
        {1.2, 4.3, 5.4, 6.5, 3.4, 6.8, 5.3, 7.6, 6.4, 7.9, 0.5, 8.9}, // TOTAL = 64.2
        {0.3, 0.5, 3.2, 5.6, 7.5, 4.5, 3.5, 2.0, 2.1, 4.9, 9.1, 8.2}, // TOTAL = 51.4
        {12.4, 2.3, 5.3, 7.1, 1.0, 3.0, 5.0, 6.0, 9.3, 8.2, 7.9, 1.9}, // TOTAL = 69.4
        {8.5, 4.6, 3.7, 2.4, 3.7, 1.2, 5.3, 7.9, 2.1, 2.0, 3.4, 5.7} // TOTAL = 50.5
    }; //1ST COLUMN TOTAL = 33.9
    //2ND COLUMN TOTAL = 24.2
    //3RD COLUMN TOTAL = 21.9

    float totalYearlyRainfall[5];
    float totalMonthlyRainfall[12];

    printf("\n\n***TOTAL & AVERAGE YEARLY RAINFALL***");

    // LOOP TO FIND THE TOTAL YEARLY RAINFALL
    for(int i = 0; i < 5; ++i) {
        for(int j = 0; j < 12; ++j) {
            totalYearlyRainfall[i] += rainfallValues[i][j];
        }
        printf("\n\nThe total yearly rainfall is: %.2f",
totalYearlyRainfall[i]);
        printf("\n\nThe average yearly rainfall is: %.2f",
totalYearlyRainfall[i] / 5);
    }

    printf("\n\n=====");
    ==");
}

printf("\n\n***TOTAL & AVERAGE MONTHLY RAINFALL***");

// LOOP TO FIND THE AVERAGE MONTHLY RAINFALL
for(int k = 0; k < 12; ++k) {
    for(int l = 0; l < 5; ++l) {
        totalMonthlyRainfall[k] += rainfallValues[l][k];
    }
}
```

```
    printf("\n\nThe total monthly rainfall is: %.2f",
totalMonthlyRainfall[k]);
    printf("\n\nThe average monthly rainfall is: %.2f",
totalMonthlyRainfall[k] / 12);

printf("\n\n*****");
}
}

return 0;
}
```

64. (Demonstration) Create a simple Weather Program

- We can use constants to define the **months** and **years**.

-

Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>

#define MONTHS 12
#define YEARS 5

int main()
{
    // initialize rainfall data for 2011-2015
    float rain[YEARS][MONTHS] =
    {
        {4.3, 4.3, 4.3, 3.0, 2.0, 1.2, 0.2, 0.2, 0.4, 2.4, 3.5, 6.6},
        {8.5, 8.2, 1.2, 1.6, 2.4, 0.0, 5.2, 0.9, 0.3, 0.9, 1.4, 7.3},
        {9.1, 8.5, 6.7, 4.3, 2.1, 0.8, 0.2, 0.2, 1.1, 2.3, 6.1, 8.4},
        {7.2, 9.9, 8.4, 3.3, 1.2, 0.8, 0.4, 0.0, 0.6, 1.7, 4.3, 6.2},
        {7.6, 5.6, 3.8, 2.8, 3.8, 0.2, 0.0, 0.0, 0.0, 1.3, 2.6, 5.2}
    };

    int year, month;
    float subtotal, total;

    printf("YEAR\t\tRAINFALL (inches)\n");

    for (year = 0, total = 0; year < YEARS; year++)
    {
        for (month = 0, subtotal = 0; month < MONTHS; month++)
        {
            subtotal += rain[year][month];
        }
        printf("%5d \t%15.1f\n", 2010 + year, subtotal);
        total += subtotal;
    }

    printf("\nThe yearly average is %.1f inches.\n\n", total/YEARS);

    printf("MONTHLY AVERAGES:\n\n");
    printf(" Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec\n");

    for (month = 0; month < MONTHS; month++)
    {
        for (year = 0, subtotal = 0; year < YEARS; year++)
            subtotal += rain[year][month];

        printf("%4.1f ", subtotal/YEARS);
    }

    printf("\n");

    return 0;
}
```

Section 9: Functions

•

65. Basics

- **Overview:**

- ◊ A function is a self contained unit of program code designed to accomplish a particular task.
 - it's basically something that you can invoke that helps you reduce duplication of code.
- ◊ It's self contained inside the code itself and it has some task that is gonna accomplish.
- ◊ Syntax rules define the structure of a function and how it can be used.
 - this is dependent on the programming language you use.
 - most programming languages support this concept of a function.
- ◊ A function in C is the same as subroutines or procedures in other languages.
- ◊ Some functions cause an action to take place:
 - **printf()** causes data to be printed on your screen.
- ◊ Some functions find a value for a program to use:
 - **strlen()** tells a program how long a certain string is.
 - it doesn't perform an action, it just gives information back to you.

- **Advantages:**

- ◊ Allow for the **divide and conquer strategy**:
 - it is very difficult to write an entire program as a single large main function.
 - difficult to test, debug and maintain.
- ◊ Without functions you wouldn't have any cohesion.
- ◊ With **divide and conquer**, tasks can be divided into several independent subtasks:
 - reduces the overall complexity.
 - separate functions are written for each subtask.
 - we can further divide each subtask into smaller subtasks, further reducing the complexity.
- ◊ Your goal is to make your functions as cohesive as possible:
 - make sure each function is **doing just one task**.
 - don't have a function do 5 or 10 different tasks.
- ◊ If you have a function that reads input from the keyboard:
 - just have it read input from the keyboard:
 - don't have it check a string or display output or anything like that.

- ◊ **Functions need to be small and they need to be very cohesive.**
- ◊ Reduce duplication of code:
 - saves you time when writing, testing and debugging code.
 - reduces the size of the source code.
- ◊ If you have to do a certain task several times in a program, you only need to write an appropriate function once:
 - the program can then use that function wherever needed.
 - you can also use the same function in different programs (`printf`):
 - this is what constitutes a library.
 - `printf()`, is from the **stdio (standard input/output) library**
- ◊ Help with readability:
 - program is better organized.
 - easier to read and easier to change or fix.
- ◊ Creating functions is basically modularizing your program.
- ◊ The maintenance phase is the most expensive part of the software life cycle.
- ◊ The divide and conquer approach also allows the parts of a program to be developed, tested and debugged independently:
 - reduces the overall development time.
- ◊ The functions developed for one program can be used in another program:
 - software reuse.
- ◊ Many programmers like to think of a function as a "**black box**"
 - information that goes in (**its input**).
 - the value or action it produces (**its output**).
- ◊ Using this "**black box**" thinking helps you concentrate on the program overall design rather than the details:
 - what the function should do and how it relates to the program as a whole before worrying about writing the code.
- ◊ Design is how you're gonna write your program, how you're gonna write the code.
- ◊ **When you're thinking about writing code in functions, just think it as inputs and outputs to the function and it's gonna help you with your design.**

- **Examples:**

- ◊ You have already used built-in functions such as **printf()** and **scanf()**.
- ◊ You should have noticed how to invoke these functions and pass data to them:
 - **arguments between parentheses following the function name.**
- ◊ e.g:
 - printf()
 - the **first argument** is usually a **string literal**
 - and the **succeeding arguments (of which there may be none)** are a **series of variables** or **expressions whose values are to be displayed.**
- ◊ You also should have noticed how you can receive information back from a function in two ways:
 - **through one of the function arguments** (scanf).
 - the **input is stored in an address that you supply as an argument.**
 - **as a return value.**
- ◊ Functions can take data into them and then they can return data out of them.
- ◊ code example:

```
#define SIZE 50

int main(void) {
    float list[SIZE]

    readlist(list, SIZE);
    sort(list, SIZE);
    average(list, SIZE);

    return 0;
}
```

- this makes the program more organized because we know immediately what is doing.
 - this makes the program very readable and easier to maintain.

- **Implementing Functions:**

- ◊ Remember, just calling functions does not work unless we implement the function itself:
 - user defined functions (the ones you create).
 - we would have to write the code for the three functions readlist(), sort() and average() in our previous example.

- ◊ Always use descriptive function names to make it clear what the program does and how it is organized:
 - if you can make the functions general enough, you can reuse them in other programs.
- ◊ In the upcoming lectures we will understand:
 - how to define them.
 - how to invoke them.
 - how to pass and return data to them.

- **main()** function:

- ◊ As a reminder, the **main()** is a specially recognized name in the C system:
 - it indicates where the program is to begin execution.
 - all C programs must always have a **main()**.
 - you can pass data to it (command line arguments).
 - returning data is optional.
 - we've been using **return 0** in our programs, but that's used when you wanna return error codes.

66. Defining Functions

- Is saying that a function exists.

- **Defining a Function:**

- ◊ When you create a function, you specify the function header as the first line of the function definition:

- followed by a starting curly brace {
 - the executable code in between the starting and ending braces.
 - the ending curly brace }
 - the block of code between the braces following the function header is called the function body.

- ◊ The function header defines the name of the function:

- **parameters** (which specify the number and types of values that are passed to the function when it's called)

- parameters have to do with passing data.
 - the type for the value that the function returns.

- ◊ The function body contains the statements that are executed when the function is called:

- these have access to any values that are passed as arguments to the function.

- ◊ **Function Structure Example:**

```
Return_type Function_name(Parameters - separated by commas) {  
    // Statements...  
}
```

- ◊ The first line of a function definition tells the compiler (in order from left to right) three things about the function:

- **that type of value it returns.**
 - **its name.**
 - **the (data) arguments it takes.**

- ◊ Choosing meaningful function names is just as important as choosing meaningful variables names:

- greatly affects the program's readability.
 - name the function based on what it's going to do.

- **Example:**

- ◊ `void printMessage(void) {`

```
    printf("Programming is fun.\n");  
}
```

- ◊ The first line of the **printMessage()** function definition tells the compiler that the function returns no value:

- keyword **void**

- ◊ Next is the name: **printMessage**

- ◊ After that is that it takes no arguments (the second use of the keyword void)

- **Defining a Function (cont'd):**

- ◊ The statement in the function body can be absent, but the braces must be present.

- ◊ If **there are no statements in the body of a function, the return type must be void** and the **function will not do anything**:

- defining a function with an empty body is often useful during the testing phase of a complicated program.
 - allows you to run the program with only selected functions actually doing something.
 - you can then add the detail for the functions bodies step by step, testing at each stage, until the whole thing is implemented and fully tested.

- **Naming functions:**

- ◊ The name of a function can be any legal name:

- not a reserved word (such as int, double, sizeof and so on...)
 - is not the same name as another function in your program.
 - known as function overload.
 - overloading is not supported in C.
 - is not the same name as any of the standard library functions.
 - would prevent you from using the library function.

- ◊ A legal name has the same form as that of a variable:

- a sequence of letters and digits.
 - first character must be a letter.
 - underline character counts as letter.

- ◊ The name that you choose should be meaningful and relevant to what the function does.

- ◊ You will often define function names (and variable names, too) that consist of more than one word.
- ◊ There are three common approaches that you can adopt:
 - separate each of the words in a function name with an underline character.
 - capitalize the first letter of each word.
 - capitalize words after the first word (camelCase).
- ◊ You can pick any one you want, but, use the same approach throughout your program.

• Function prototypes:

- ◊ Declaring a function.
- ◊ A function prototype is a statement that defines a function:
 - defines its name, its return value type and the type of each of its parameters.
 - provides all the external specifications for the function.
- ◊ You can write a prototype for a function exactly the same as the function header:
 - only difference is that you add a semicolon at the end.
- ◊ Example:
 - `void printMessage(void);`
- ◊ A function prototype enables the compiler to generate the appropriate instructions at each point where you call the function.
 - it also checks that you are using the function correctly in each invocation.
- ◊ When you include a standard header file in a program, the header file adds the function prototypes for that library to your program:
 - the header file stdio.h contains function prototypes for printf(), among others.
- ◊ They generally appear at the beginning of a source file prior to the implementation of any function or in a header file:
 - they usually appear in source files if only that source file is using that function.
 - otherwise you put them in a header file if they're being reused by many source files.
- ◊ It allows any of the functions in the file to call any function regardless of where you have placed the implementation of the functions.

- ◊ Parameter names do not have to be the same as those used in the function definition:
 - not required to include the names of parameters in a function prototype.
- ◊ It's good practice to always include declarations for all of the functions in a program source file, regardless of where they are called:
 - will help keep your programs more consistent in design.
 - prevent any errors from occurring if, at any stage, you choose to call a function from another part of your program.

- ◊ code example:

```
// #include & #define directives

double Average(double data_values[], size_t count);
double Sum(double *x, size_t n);
size_t GetData(double*, size_t);

int main(void) {
    // Code in main()...
}

// Definitions/implementations for Average(), Sum() and GetData()...
```

- if you call the Average(), the Sum() and the GetData() functions inside the main() function:

→ without including the **function (headers) prototypes**:

⇒ if they are defined below the **main()** routine you'd have errors or warnings.

- Let's take a look at an example in our IDE:

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 8 - SIMPLE WEATHER PROGRAM - MY SOLUTION
DATE: MARCH 21ST, 2022
*/

#include <stdio.h>

// FUNCTION DECLARATION
void add(); // WE'RE TELLING THE COMPILER THAT WE'RE GONNA USE THE FUNCTION
            // SOMEWHERE,
            // BUT WE DON'T KNOW WHERE THE IMPLEMENTATION IS.

int main(){
    add();
}
```



```
    return 0;  
}  
  
// add FUNCTION  
void add() {  
}
```

- it's a good practice to declare your functions before the **main()** function.
 - or
- you gotta provide function prototypes before the **main()** function.

67. Arguments and Parameters

- These terms tend to be used interchangeably.
- Data that you pass to a function.
- A **parameter** is a variable in a **function (prototype) declaration** and **function definition/implementation** (the actual code.)
- When a function is called, **the arguments are the data you pass into the function's parameters:**
 - ◊ the actual value of a variable that gets passed to the function.
- Function parameters are defined within the function header:
 - ◊ they are placeholders for the arguments that need to be specified when the function is called.
- **Clear Definition:**
 - **Parameters:**
 - ◊ Have to do with the function definition and declaration.
 - **Arguments:**
 - ◊ These are when we call/invoke the function:
 - the data that you're passing.
 - **Arguments** are used when you invoke, these are the arguments that are going to the function:
 - ◊ and when the function receives that data they're referred to as **parameters**.
- The parameters for a function are a list of parameter names with their types:
 - ◊ each parameter is separated by a comma.
 - ◊ entire list of parameters is enclosed between the parentheses that follow the function name.
- **A function can have no parameters, in which case you should put *void* between the parentheses.**
 - ◊ no parameters means that no data is passed to it, it also means no return data.

- you can also leave it as empty, nothing inside the parentheses.
- Parameters provide the means to pass data to a function:
 - ◊ data passed from the calling function to the function that is called.
- The names of the parameters are local to the function:
 - ◊ they will assume the values of the arguments that are passed when the function is called.
- The body of the function should use these parameters in its implementation.
 - ◊ if it doesn't then the arguments shouldn't have been passed.
- A function may have additional locally defined variables that are needed by the function's implementation.
- When passing an array as an argument to a function:
 - ◊ you must also pass an additional argument specifying the size of the array.
 - ◊ the function has no means of knowing how many elements there are in the array.

• Example:

- ◊ When the **printf()** function is called, you always supply one or more values as arguments.
 - the first value being the format string.
 - the remaining values being any variables to be displayed.
 - this is a function that takes a variable number of arguments, but this is a more advanced topic.
- ◊ Parameters greatly increase the usefulness and flexibility of a function:
 - the **printf()** function displays whatever you tell it to display via the parameters and arguments passed.
- ◊ Passing data to a function makes it a lot more flexible and useful.
- ◊ It is a good idea to add comments before each of your own function definitions:
 - it helps explain what the function does and how the arguments are to be used.
 - these are usually multiline comments.
 - there you say the name of the function and the purpose of the function.
 - ⇒ also you can describe any data that's being passed to the function.
 - this is for maintenance and to be able to find and fix errors much quicker.

• Function example:

```
#include <stdio.h>

void multiplyTwoNumbers(int x, int y){
    int result = x * y;
    printf("The product of %d multiplied by %d is: %d\n", x, y, result);
}

int main(void) {
    multiplyTwoNumbers(10, 20);
    multiplyTwoNumbers(20, 30);
    multiplyTwoNumbers(50, 2);

    return 0;
}
```



- you can pass any data type into a function, such as:
 - arrays
 - pointers
 - structures

68. Returning data from functions

- **Returning Data:**

- ◊ In our prior example of multiply two numbers, our multiply function displayed the results of the calculation at the terminal.

- ◊ You might not always want to have the results of your calculations displayed:
 - the function becomes more useful, more flexible, more generic:
 - which in turn allows it to be used more if you can return data.
 - functions can return data using specific syntax.
 - we should be familiar with this from previous experience with the main function.

- ◊ Let's take a look at the general form of a function:

```
Return_type Function_name(List of parameters - separated by commas) {  
    // Statements  
}
```

- ◊ The **Return_type** specifies the type of the value returned by the function.
 - ◊ Functions can return any type of data.

- **The return type:**

- ◊ You can specify the type of value to be returned by a function as any of the legal types in C:
 - include **enum** types and **pointers**.
 - it makes it very flexible when you can return pointers.
 - ◊ The **return type** can also be type **void** which means no value is returned.
 - no data is gonna be returned.
 - if you see **void** inside the **parameters list** you're not passing any data.
 - you can also use **void** as a pointer as well.
 - ◊ **Void** is a way of saying nothing:
 - the reason why you include it in your programs is that it makes it more readable.
 - ◊ On the return type you have to have the word **void** in there, it can't be empty.

- **The return statement:**

- ◊ **The return statement provides the means of exiting from a function.**

- ◊ If you provide the **return** keyword with:
 - **no variable name or expression:** it means that you want to exit the function.
 - similar to a break in a loop.
 - you can provide this keyword in functions that are defined as **void**.
 - ⇒ it makes it clear when you're exiting the function.
 - you should have the **return** at the bottom of the function.
 - ◊ if you see it anywhere inside the function it means that you're jumping out of that function.
 - ◊ This form of the return statement is used exclusively in a function where the return type has been declared as void:
 - does not return a value.
 - ◊ The more general form of the return statement is:
 - `return expression;`
 - this form of the return statement must be used when the return value type for the function has been declared as some type other than void.
 - ◊ The value that is returned to the calling program is the value that results when **expression** is evaluated:
 - should be of the return type specified for the function.

• Returning Data:

- ◊ A function that has statements in the function body but does not return a value must have the return type as **void**:
 - you will get an error message if you compile a program that contains a function with a **void** return type that tries to return a value.
- ◊ A function that does not have a **void** return type must return a value of the specified return type:
 - you will get an error message from the compiler if return type is different than specified.
- ◊ If expression results in a value that's a different type from the return type in the function header, the compiler will insert a conversion from the type of expression to the one required:
 - if conversion is not possible then the compiler will produce an error message.
 - for example:
 - if you're trying to return an int when it's a string or when it's an enum, you'll get an error message.

- ◊ **There can be more than one return statement in a function:**

- each return statement must supply a value that is convertible to the type specified in the function header for the return value.
 - you should avoid doing this.

- **Invoking a function:**

- ◊ You call a function by using the function name followed by the arguments to the function between parentheses.

- ◊ When you call the function, the values of the arguments that you specify in the call will be assigned to the parameters in the function.

- ◊ When the function executes, the computation proceeds using the values you supplied as arguments.

- ◊ The arguments you specify when you call a function should agree in type, number and sequence with the parameters in the function header.

- **Invoking a function and assigning data returned:**

- ◊ If the function is used as the right side of an assignment statement, the return value supplied by the function will be substituted for the function:

- will also work with an expression.

- ◊ Example:

- ```
int x = myFunctionCall();
```

- whatever the previous function returns is gonna be assigned to x:
  - if it returns  $x + y$ , the result of  $x + y$  is gonna be assigned to x.

- ◊ The calling function doesn't have to recognize or process the value returned from a called function:

- it's up to you how you use any values returned from function calls.

- **Invoking a function and returning data from a function example:**

```
int multiplyTwoNumbers(int x, int y) {
 int result = x * y;
 return result;
}

int main(void){
 int result = 0;
 result = multiplyTwoNumbers(10,20);

 printf("result is %d\n", result)
 return 0;
}
```

- ❖ flexibility with data being passed and being returned is a big deal.

# **69. Local and Global Variables**

- This is the concept of **function scope** or in general **program scope**.
- It has to do with when you can use variables and when you cannot use them.

- **Local Variables:**

- ◊ Variables defined inside a function are known as **automatic local variables**:
  - they are automatically "created" each time the function is called.
  - their values are local to the function.
- ◊ Local variables has to do with anything that has a block of code.
- ◊ The value of a local variable can only be accessed by the function in which the variable is defined:
  - its value cannot be accessed by any other function.
- ◊ If an initial value is given to a variable inside a function, that initial value is assigned to the variable each time the function is called.
- ◊ If you wanna retain a value in a variable every subsequent call, there's a way you can do that in C, it's referred to as a:
  - **static variable** inside of a function.
- ◊ You can use the **auto** keyword to be more precise, but, not necessary, as the compiler adds this by default.
- ◊ **Local variables are also applicable to any code where the variable is created in a block (loops, if statements).**
  - blocks can be anywhere where there's brackets { }.

- **Global Variables:**

- ◊ This is the opposite of local variable.
- ◊ A global variable's value can be accessed by any function in the program.

- ◊ A global variable has the same lifetime as that of the program.
  - it's alive from when the program starts to when it stops, the variable is gonna be located in memory.
- ◊ Global variables are declared outside of any function:
  - does not belong to any particular function.
- ◊ Any variables defined in the **main function** are only accessible in the main function.
- ◊ If you want a variable that's used by all functions:
  - you put the variable outside of a function.
    - usually at the top of the source file.
    - you also mark it as saying "this is a global variable"
- ◊ Any function in the program can change the value of a global variable.
  - because they have access to it they can modify it.
- ◊ If there is a local variable declared in a function with the same name, then, within that function the local variable will mask the global variable:
  - global variable is not accessible and prevent it from being accessed normally.
  - **basically if the global and local variable have the same name, the local variable is gonna take precedence.**
  - don't use the same names with global and local variables because it is gonna be confusing.

- **Code example:**

```

int myGlobal = 0; // global variable

int main() {
 int myLocalMain = 0; // local variable
 // can access myGlobal and myLocal
 return 0;
}

void myFunction() {
 int x; // local variable
 // can access myGlobal and x, cannot access myLocalMain
}

```

- ◊ if you try to access a local variable from a function that doesn't have access to it, the compiler is gonna give you an error.

- **Avoid Using Global Variables:**

- ◊ In general, global variables are a "bad" thing and should be avoided:

- promoted coupling between functions (dependencies).

- example:

- if you have a global variable and it's being used by four functions, that means each one of those functions could be dependent on the other functions.

- ⇒ dependencies cause a problem to be much harder to debug and find errors.

- if you find the error and you isolate it, it might be very hard to fix it, because now when you fix it you may cause problems in other functions.

- hard to find the location of a bug in a program.

- hard to fix a bug once it's found.

- ◊ Global variables should be avoided as much as possible because of this **coupling**.

- ◊ There are times where you might need to use it, but you should avoid it.

- ◊ Modern OOP languages don't have the concept of a global variable.

- they have static variables, which can be class variables, but they're still encapsulated inside of a class.

- ◊ Use parameters in functions instead:

- pass data to functions.

- if a function needs access to some data pass it to it.

- if there's a lot of data and you don't wanna have a lot of parameters use a **struct**.

- **in theory you shouldn't have more than 5 parameters.**

# **70. (Challenge) Write some functions!**

- In this challenge you're gonna write a number of different functions.
- We need to get some practice writing functions.
  - ◊ better organized code.
  - ◊ more modular.
  - ◊ avoid duplication.

- **Requirements:**

- For this challenge you are to write three functions in a single program.
- Write a function which finds the **greatest common divisor** of **two non-negative integer values** and to return the result:
  - gcd**, takes two **ints** as **parameters**.
  - to find the **greatest common divisor** you can look online or any math book.
- Calculate a function to **calculate the absolute value of a number**:
  - it should take as a parameter a **float** and return a **float**.
  - test this function with **ints** and **floats**.
  - implementation:
    - check if it's less than zero make sure it's positive.
    - multiply the number by -1 or variable by its negative counterpart.
- Write a function to compute the square root of a number:
  - if a negative argument is passed then a message is displayed and -1.0 should be returned.
  - should use the **absoluteValue** function as a helper function in the above step.
    - ⇒ when one function uses another function it's sometimes referred to as a helper function.
  - In the square root function you have to invoke the **absoluteValue** function:
  - use it to test whether or not the arguments are negative.
- Use function prototypes at the top of the file.
- Return the correct data.
- Pass data to these functions.

# My Code

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 9 - WRITE SOME FUNCTIONS - MY SOLUTION
DATE: MARCH 24TH, 2022
*/

// INCLUDES STATEMENTS
#include <stdio.h>

//FUNCTION PROTOTYPES
void main(void); // MAIN FUNCTION
unsigned int gcd(unsigned int x, unsigned int y); // GREATEST COMMON DIVISOR
FUNCTION
float absoluteValue(float a); // ABSOLUTE VALUE OF A NUMBER FUNCTION
float squareRoot(float); // SQUARE ROOT OF A NUMBER FUNCTION

// MAIN FUNCTION
void main(void){
 // CALL THE GCD FUNCTION
 // gcd(128, 96);

 printf("\nThe GCD is: %f", gcd(128, 96));

 // CALL THE absoluteValue FUNCTION
 // absoluteValue(-2);

 printf("\nThe absolute value is: %f", absoluteValue(-2));

 // CALL THE SQUARE ROOT OF A NUMBER FUNCTION
 // squareRoot(25);

 return;
}

// GREATEST COMMON DIVISOR FUNCTION
unsigned int gcd(unsigned int x, unsigned int y) {
 // DECLARE LOCAL VARIABLE
 unsigned int z;

 do {
 if (x > y)
 z = (x / y);
 else
 z = (y / x);
 // (x > y ? z = (x / y) : z = (y / x));
 } while (z != 0);

 return z;
}

// ABSOLUTE VALUE OF A NUMBER FUNCTION
float absoluteValue(float a) {
```

```
if(a < 0)
 a *= -1;
else
 a = a;
// (a < 0 ? a *= -1 :);
return a;
}

// SQUARE ROOT OF A NUMBER FUNCTION
// float squareRoot(float) {
// // HERE WE HAVE TO INVOKE THE PREVIOUS absoluteValue FUNCTION
// absoluteValue(b)
// for(i = b; i < ;)
// sqrRoot = ;
// }
```

## **71. (Demonstration) Write some functions!**

- We also have to declare the variable names along with the data type in the function prototypes.
- To get the absolute value of a negative number we could also make it equal to its positive counterpart:
  - ◊ example:

```
if (x < 0)
 x = -x;
```

•

# Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>

int gcd(int u, int v);
float absoluteValue(float x);
float squareRoot(float x);

int main()
{
 int result = 0;

 float f1 = -15.5, f2 = 20.0, f3 = -5.0;
 int i1 = -716;
 float absoluteValueResult = 0.0;

 result = gcd(150, 35);
 printf("The gcd of 150 and 35 is %d\n", result);

 result = gcd(1026, 405);
 printf("The gcd of 1026 and 405 is %d\n", result);

 printf("The gcd of 83 and 240 is %d\n\n\n", gcd(83, 240));

 /* testing absolute Value Function */
 absoluteValueResult = absoluteValue (f1);
 printf ("result = %.2f\n", absoluteValueResult);
 printf ("f1 = %.2f\n", f1);

 absoluteValueResult = absoluteValue (f2) + absoluteValue (f3);
 printf ("result = %.2f\n", absoluteValueResult);

 absoluteValueResult = absoluteValue ((float) i1);
 printf ("result = %.2f\n", absoluteValueResult);

 absoluteValueResult = absoluteValue (i1);
 printf ("result = %.2f\n", absoluteValueResult);

 printf ("% .2f\n\n\n", absoluteValue (-6.0) / 4);

 printf ("% .2f\n", squareRoot(-3.0));
 printf ("% .2f\n", squareRoot(16.0));
 printf ("% .2f\n", squareRoot(25.0));
 printf ("% .2f\n", squareRoot(9.0));
 printf ("% .2f\n", squareRoot(225.0));

 /* testing square root */

 return 0;
}

int gcd(int u, int v)
{
 int temp;

 while(v != 0)
 {
```

```

 temp = u % v;
 u = v;
 v = temp;
 }

 return u;
}

float squareRoot(float x)
{
 const float epsilon = .00001;
 float guess = 1.0;
 float returnValue = 0.0;

 if (x < 0)
 {
 printf ("Negative argument to squareRoot.\n");
 returnValue = -1.0;
 }
 else
 {
 while (absoluteValue (guess * guess - x) >= epsilon)
 guess = (x / guess + guess) / 2.0;

 returnValue = guess;
 }

 return returnValue;
}

float absoluteValue(float x)
{
 if (x < 0)
 x = -x;

 return x;
}

```

## **72. (Challenge) Create a Tic Tac Toe Game**

- This challenge is related to functions, but it's gonna be more divide and conquer.
- It's gonna be more about organizing your program around modules, more about problem solving.
- This is gonna help you become a better problem solver.
- You're gonna utilize an array.
- You're gonna have to figure out how to format certain data.
- It's gonna be a simulation of a tic tac toe game, it's gonna be player 1 vs player 2.

- **Requirements:**

- Write a program that plays a tic-tac-toe:
  - game is played on a 3x3 grid, the game is played by two players, who take turns.
- You should create an array to represent the board:
  - can be of type **char** and consist of 10 elements (do not use zero).
  - each element represents a coordinate on the board that the user can select.
- Some functions that you should probably create:
  - checkForWin** → checks to see if a player has won or the game is a draw.
  - drawBoard** → redraws the board for each player turn.
  - markBoard** → sets the char array with a selection and checks for an invalid selection.
- The game can look something like this:
  -

```
C:\Users\jfedin\Downloads\Test\bin\Debug\Test.exe Something
```

Tic Tac Toe

Player 1 (X) - Player 2 (O)

|   |   |   |
|---|---|---|
|   |   |   |
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

Player 1, enter a number:

```
C:\Users\jfedin\Downloads\Test\bin\Debug\Test.exe Something
```

Tic Tac Toe

Player 1 (X) - Player 2 (O)

|   |   |   |
|---|---|---|
|   | X | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

Player 2, enter a number: 10

Invalid move

Tic Tac Toe

Player 1 (X) - Player 2 (O)

|   |   |   |
|---|---|---|
| 1 | X | O |
| 4 | O | O |
| X | X | X |

==>Player 1 win



# My Code

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 10 - CREATE A TIC TAC TOE GAME - MY SOLUTION
DATE: MARCH 28TH, 2022
*/

// INCLUDES STATEMENTS
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// FUNCTION PROTOTYPES
void main(void); // MAIN FUNCTION
int checkForWin(char); // CHECK FOR WIN OR DRAW FUNCTION
char drawBoard(char); // DRAW BOARD FUNCTION
char markBoard(char); // MARK BOARD FUNCTION

// MAIN FUNCTION
void main(void) {
 // DECLARE VARIABLES AND ARRAYS

 // GAME NAME
 printf("\t\t\tTic-Tac-Toe\n");

 // EACH PLAYER'S CHARACTER
 printf("\nPlayer 1 (X) - Player 2 (O)\n");

 return;
}

// CHECK FOR WIN OR DRAW FUNCTION
int checkForWin(char) {
 // DECLARE VARIABLES
 enum whoWins {0, 1, 2, 3}; /* = 1 --> PLAYER 1 WINS
 = 2 --> PLAYER 2 WINS
 = 3 --> DRAW
 = 0 --> GAME NOT YET CONCLUDED
 */
 // ITERATE THROUGH THE BOARD TO CHECK WHO WINS OR IF IT'S A DRAW
}
```

```

for(i = 0; i < board; ++i) {
 for(j = 0; j < board; ++j) {
 // SWITCH CASE TO CHECK WHO WINS OR IF IT'S A DRAW
 switch (board) {
 // FIRST ROW
 case (board[0][0] == 'X' && board[0][1] == 'X' && board[0]
[2] = 'X') :
 whoWins = 1;
 break;
 case (board[0][0] == 'O' && board[0][1] == 'O' && board[0]
[2] = 'O') :
 whoWins = 2;
 break;
 // SECOND ROW
 case (board[1][0] == 'X' && board[1][1] == 'X' && board[1]
[2] = 'X') :
 whoWins = 1;
 break;
 case (board[1][0] == 'O' && board[1][1] == 'O' && board[1]
[2] = 'O') :
 whoWins = 2;
 break;
 // THIRD ROW
 case (board[2][0] == 'X' && board[2][1] == 'X' && board[2]
[2] = 'X') :
 whoWins = 1;
 break;
 case (board[2][0] == 'O' && board[2][1] == 'O' && board[2]
[2] = 'O') :
 whoWins = 2;
 break;
 // FIRST COLUMN
 case (board[0][0] == 'X' && board[1][0] == 'X' && board[2]
[0] = 'X') :
 whoWins = 1;
 break;
 case (board[0][0] == 'O' && board[1][0] == 'O' && board[2]
[0] = 'O') :
 whoWins = 2;
 break;
 // SECOND COLUMN
 case (board[0][1] == 'X' && board[1][1] == 'X' && board[2]
[1] = 'X') :
 whoWins = 1;
 break;
 case (board[0][1] == 'O' && board[1][1] == 'O' && board[2]
[1] = 'O') :
 whoWins = 2;
 break;
 // THIRD COLUMN
 case (board[0][2] == 'X' && board[1][2] == 'X' && board[2]
[2] = 'X') :
 whoWins = 1;
 break;
 case (board[0][2] == 'O' && board[1][2] == 'O' && board[2]
[2] = 'O') :
 whoWins = 2;
 break;
 /* LEFT DIAGONAL --> \ */
 case (board[0][0] == 'X' && board[1][1] == 'X' && board[2]
[2] = 'X') :
 whoWins = 1;
 break;
 case (board[0][0] == 'O' && board[1][1] == 'O' && board[2]
[2] = 'O') :
 whoWins = 2;
 break;
 }
 }
}

```

```

 whoWins = 2;
 break;
 /* RIGHT DIAGONAL --> / */
 case(board[0][2] == 'X' && board[1][1] == 'X' && board[2]
[0] = 'X'):
 whoWins = 1;
 break;
 case(board[0][2] == 'O' && board[1][1] == 'O' && board[2]
[0] = 'O'):
 whoWins = 1;
 break;
 default:
 // whoWins = ;
 break;
 }
}
}

return whoWins;
}

// DRAW BOARD FUNCTION
// REDRAWS THE BOARD FOR EACH PLAYER TURN
char drawBoard(char array) {

 for(i = 0; i < board; ++i){
 for(j = 0; j < board; ++j) {
 board[i][j];
 }
 }

 printf("");
}

// MARK BOARD FUNCTION
// SETS THE CHAR ARRAY WITH A SELECTION AND CHECKS FOR AN INVALID SELECTION
char markBoard(void) {
 // DECLARE VARIABLES AND ARRAYS
 char board[3][3] = {
 {1, 2, 3},
 {4, 5, 6},
 {7, 8, 9}
 };
 char player1Move;
 char player2Move;
 bool winCheck = false;

 // REQUEST USER MOVE
 do {

 // REQUEST USER'S 1 MOVE
 printf("Player 1's turn.\nEnter a number: ");
 scanf("%c", &player1Move);

 // LOOP TO ITERATE THROUGH THE BOARD AND MARK PLAYER 1'S SELECTION
 for(i = 0; i < board; ++i){
 for(j = 0; j < board; ++j) {
 board[i][j] = player1Move;
 }
 }
 }
}
```

```
// INVOKE THE drawBoard() FUNCTION
drawBoard(board);

// LOOP TO ITERATE THROUGH THE BOARD AND MARK PLAYER 2'S SELECTION
printf("Player 2's turn.\nEnter a number: ");
scanf("%c", &player2Move);

// LOOP TO ITERATE THROUGH THE BOARD AND MARK PLAYER 2'S SELECTION
for(k = 0; k < board; ++k) {
 for(l = 0; l < board; ++l) {
 board[k][l] = player2Move;
 }
}

// INVOKE THE drawBoard() FUNCTION
drawBoard(board);
checkForWin(winCheck);
} while(winCheck == false)

}

return board;
```

## 73. (Demonstration) Create a Tic Tac Toe Game

- There are many solutions to a given problem, this is gonna be the instructor's solution.
- If your solution is different, it's fine if you follow good programming practices.
- Tie the modules as much as he can.
- Here a global variable is used because the concept of **pass by reference** hasn't been explained yet.
  - ◊ to pass the (local variables) data to functions, we'd need to use **pass by reference**, pass around pointers.
- An array that contains **characters** must have **single quotes around them** when being declared:

```
char board[3][3] = {
 {'1', '2', '3'},
 {'4', '5', '6'},
 {'7', '8', '9'}
};
```

- To print out the board, we could do the following:

The screenshot shows a code editor window with a file named "main.c". The code prints a Tic Tac Toe board using printf statements. The board is represented by a 3x3 grid of squares. The code uses square brackets to index into the board array. The printf statements are as follows:

```
*main.c x|
71 printf("Player 1 (X) - Player 2 (O)\n\n");
72
73
74 printf(" | | \n");
75 printf(" %c | %c | %c \n", square[1], square[2], square[3]);
76
77 printf(" | | \n");
78 printf(" | | \n");
79
80 printf(" %c | %c | %c \n", square[4], square[5], square[6]);
81
82 printf(" | | \n");
83 printf(" | | \n");
84
85 printf(" %c | %c | %c \n", square[7], square[8], square[9]);
86
87 printf(" | | \n\n");
88 }
```

- The **getch()** pause the program and forces the user to press the enter button.

•

# Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

char square[10] = { 'o', '1', '2', '3', '4', '5', '6', '7', '8', '9' };
int choice, player;

int checkForWin();
void displayBoard();
void markBoard(char mark);

int main()
{
 int gameStatus;

 char mark;

 player = 1;

 do
 {
 displayBoard();

 // change turns
 player = (player % 2) ? 1 : 2;

 // get input
 printf("Player %d, enter a number: ", player);
 scanf("%d", &choice);

 // set the correct character based on player turn
 mark = (player == 1) ? 'X' : 'O';

 // set board based on user choice or invalid choice
 markBoard(mark);

 gameStatus = checkForWin();

 player++;
 }while (gameStatus == -1);

 if (gameStatus == 1)
 printf("==>\aPlayer %d win ", --player);
 else
 printf("==>\aGame draw");

 return 0;
}

FUNCTION TO RETURN GAME STATUS
1 FOR GAME IS OVER WITH RESULT
-1 FOR GAME IS IN PROGRESS
0 GAME IS OVER AND NO RESULT
*****/
int checkForWin()
```

```

{
 int returnValue = 0;

 if (square[1] == square[2] && square[2] == square[3])
 {
 returnValue = 1;
 }
 else if (square[4] == square[5] && square[5] == square[6])
 returnValue = 1;

 else if (square[7] == square[8] && square[8] == square[9])
 returnValue = 1;

 else if (square[1] == square[4] && square[4] == square[7])
 returnValue = 1;

 else if (square[2] == square[5] && square[5] == square[8])
 returnValue = 1;

 else if (square[3] == square[6] && square[6] == square[9])
 returnValue = 1;

 else if (square[1] == square[5] && square[5] == square[9])
 returnValue = 1;

 else if (square[3] == square[5] && square[5] == square[7])
 returnValue = 1;

 else if (square[1] != '1' && square[2] != '2' && square[3] != '3' &&
 square[4] != '4' && square[5] != '5' && square[6] != '6' && square[7]
 != '7' && square[8] != '8' && square[9] != '9')
 returnValue = 0;
 else
 returnValue = -1;
}

return returnValue;
}

FUNCTION TO DRAW BOARD OF TIC TAC TOE WITH PLAYERS MARK

void displayBoard()
{
 system("cls");

 printf("\n\n\tTic Tac Toe\n\n");

 printf("Player 1 (X) - Player 2 (O)\n\n");

 printf(" %c | %c | %c \n", square[1], square[2], square[3]);
 printf("-----|-----|-----\n");
 printf(" %c | %c | %c \n", square[4], square[5], square[6]);
 printf("-----|-----|-----\n");
 printf(" %c | %c | %c \n", square[7], square[8], square[9]);
}

```

```

 printf(" | |\n\n");
}

set the board with the correct character,

x or o in the correct spot in the array

void markBoard(char mark)
{
 if (choice == 1 && square[1] == '1')
 square[1] = mark;

 else if (choice == 2 && square[2] == '2')
 square[2] = mark;

 else if (choice == 3 && square[3] == '3')
 square[3] = mark;

 else if (choice == 4 && square[4] == '4')
 square[4] = mark;

 else if (choice == 5 && square[5] == '5')
 square[5] = mark;

 else if (choice == 6 && square[6] == '6')
 square[6] = mark;

 else if (choice == 7 && square[7] == '7')
 square[7] = mark;

 else if (choice == 8 && square[8] == '8')
 square[8] = mark;

 else if (choice == 9 && square[9] == '9')
 square[9] = mark;
 else
 {
 printf("Invalid move ");
 player--;
 getch();
 }
}

```

## ***Section 10: Character Strings***

## 74. Overview

- In this section we're gonna talk about character strings.
- We're gonna talk about how we're gonna store strings in character arrays.

- **Strings:**

- ◊ We have learned all about the **char** data type.
  - contains a single character.
- ◊ To assign a single character to a **char variable**, the character is enclosed **within a pair of single quotation marks**.
  - `plusSign = '+'`
- ◊ You have also learned that there is a distinction made between the single quotation and double quotation marks:
  - `plusSign = "+"; // INCORRECT IF plusSign IS A char`
- ◊ A **string constant** or **string literal** is a sequence of characters or symbols **between a pair of double-quote characters**:
  - **anything between a pair of double quotes is interpreted by the compiler as a string.**
  - **includes any special characters and embedded spaces.**
- ◊ Every time you have displayed a message using the `printf()` function, you have defined the message as a string constant.
- ◊ Understand the difference between single quotation and double quotation marks:
  - both are used to create two different types of constants in C.
- ◊ A backslash in a string always signals the start of an escape sequence to the compiler.
- ◊ **How a string is represented in memory:**

## String in Memory

|                                                                                 |
|---------------------------------------------------------------------------------|
| "This is a string."                                                             |
| T h i s i s a s t r i n g . \0                                                  |
| 84 104 105 115 32 105 115 32 97 32 115 116 114 105 110 103 46 0                 |
| "This is on\ntwo lines."                                                        |
| T h i s i s o n \n t w o \i n l i n e s . \0                                    |
| 84 104 105 115 32 105 115 32 111 110 10 116 119 111 32 108 105 110 101 115 46 0 |
| "For \" you write \\\"."                                                        |
| F o r " w r i t e \\" = . \0                                                    |
| 70 111 114 32 34 32 119 114 105 116 101 32 92 34 46 0                           |

### ◊ How a string is represented in memory:

- the backslash 0 character:

- \0

- is the null terminator.

- even the \n which represents a new line is in memory.

- you should notice that the string in memory always has at least one more byte for that \0 null terminator.

- the values underneath are the **ASCII values**.

### • Null character:

- ◊ A special character with code value 0 is added to the end of each string to mark where it ends (so the compiler knows where it ends):

- this character is known as the **null character** and you write it as \0.

- ◊ This is a very important character.

- ◊ **A string is always terminated by a null character**, so the **length of a string is always one greater than the number of characters in the string**.

- $\text{stringLength} = \text{sizeOfString} + 1$

- ◊ When we define string arrays, it always has to be one more than how many characters you wanna store.

- ◊ Don't confuse the **null character** with **NULL**:

- **null character** is a string terminator.

- **NULL** is a symbol that represents a memory address that doesn't reference anything.

- this symbol usually comes into play when we talk about pointers.

- ◊ You can add a \0 character to the end of a string explicitly:

- **this will create two strings.**

- ◊ code example:

```
#include <stdio.h>

int main(void) {
 printf("The character \0 is used to terminate a string.");
 return 0;
}
```

- if you compile and run this program, you'll get this output:
  - The character.
  - only the first part of the string has been displayed.
  - output ends after the first two words because the function stops outputting the string when it reaches the first null character.
  - the second **\0** at the end of the string will never be reached.

- ◊ The first **\0** that's found in a character sequence always marks the end of the string (it's there even though it's not explicitly defined).

# 75. Defining a String

- In this lecture we're gonna talk about creating a character string, using a character array.
- The instructor tends to use the terms **string** and **character string** interchangeably, but **a character string is the array we're gonna talk about.**
- Now we're gonna be able to utilize words and text in our programs.
- C doesn't have a data type that represents a string.

- **Character Strings:**

- ◊ C has no special variable type for strings:
  - this means there are no special operators in the language for processing strings.
  - the standard library provides an extensive range of functions to handle strings.
- ◊ You can't use operators on strings, no plusing or minusing strings.
- ◊ Strings in C are stored in an array of type char:
  - characters in a string are stored in adjacent memory cells, one character per cell.
- ◊ To declare a string in C, simply use the **char type** and the brackets to indicate the size:
  - code example:
    - `char myString[20];`
      - when declaring a string like this there's no actual data associated with it:
      - ⇒ you're just specifying the size.
- ◊ This variable can accommodate a string that contains up to 19 characters:
  - **you must allow one element for the termination character (null character).**
- ◊ **When you specify the dimension of an array that you intend to use to store a string, it must be at least one greater than number of characters in the string that you want to store:**
  - the compiler automatically adds `\0` to the end of every string constant.

- **Initializing a String:**

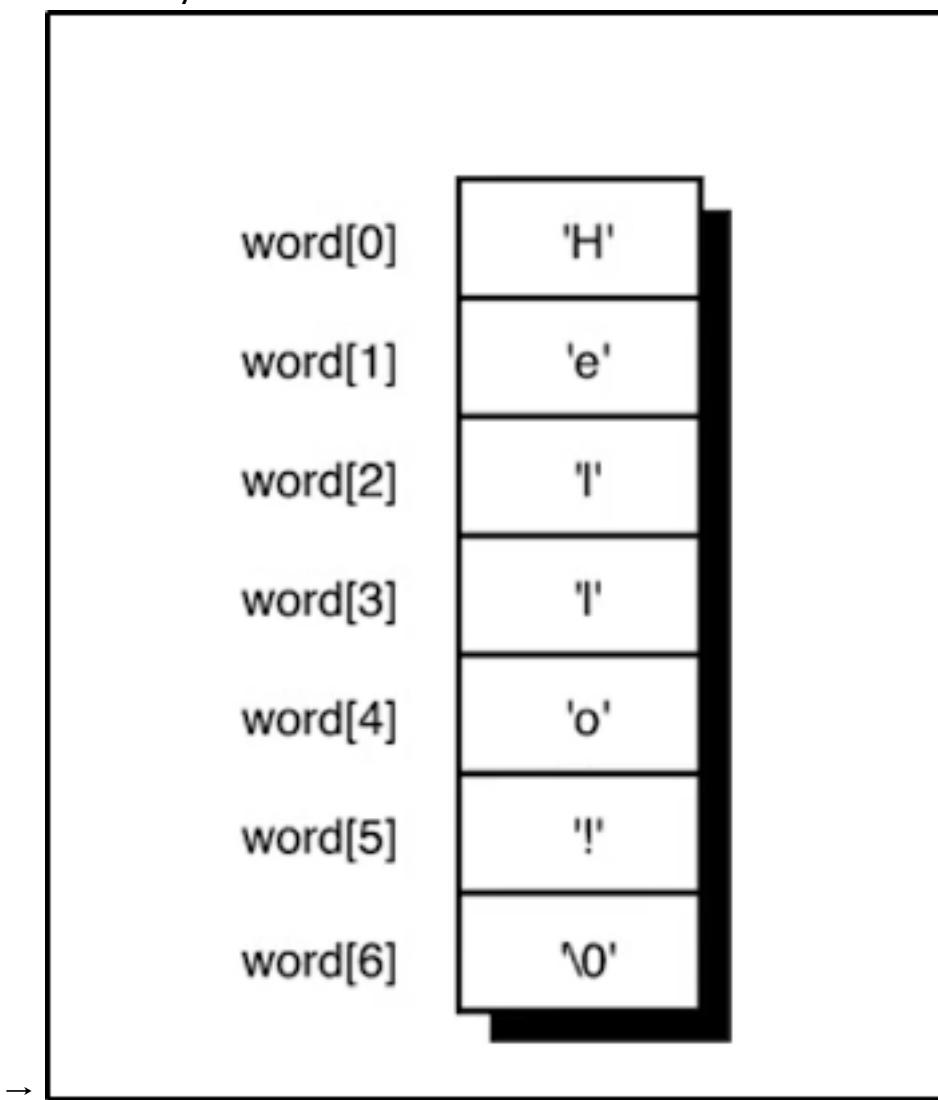
- ◊ You can initialize a string variable when you declare it.

◊ code example:

```
- char word[] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
```

◊ To initialize a string, it is the same as any other array initialization:

- in the absence of a particular array size, the C compiler automatically computes the number of elements in the array.
  - based upon the number of initializers.
  - this statement (on the previous example) reserves space in memory for exactly seven characters.
  - automatically adds the null terminator.



◊ It'd be really annoying if you always had to initialize your character arrays with single quotes followed by commas.

◊ You can specify the size of the string explicitly, just make sure you leave enough space for the terminating null character.

▪ code example:

```
- char word[7] = {"Hello!"};
```

→ **what you should really do is let the compiler do it for you.**

⇒ **don't put any numbers inside the brackets for the size.**

◊ If the size specified is too small, then the compiler can't fit a terminating null character at the end of the array, and it doesn't put one there (and it doesn't complain about one either).

▪ code example:

```
- char word[6] = {"Hello!"};
```

→ this will cause a bug in the program and you won't even know.

◊ So... do not specify the size, let the compiler figure out, you can be sure it will be correct.

◊ You can initialize just part of an array of elements of type char with a string:

▪ code example:

```
- char str[40] = "To be";
```

→ The compiler will initialize the first five elements, **str[0]** to **str[4]**, with the characters of the string constant:

⇒ **str[5]** will contain the null character '**/0**' .

⇒ space is allocated for all 40 elements of the array.

## • Assigning a value to a string after initializing:

◊ Since you can not assign arrays in C, you can not assign strings either.

◊ The following is an error:

```
char s[100]; // DECLARE

s = "hello"; //INITIALIZE - DOESN'T WORK ('lvalue required' error)
```

◊ You are performing an assignment operation, and you cannot assign one array of characters to another array of characters like this:

▪ you have to use **strncpy()** to assign a value to a char array after it has been declared or initialized.

◊ The below is perfectly valid to assign values to a **character array**:

```
s[0] = 'h';
s[1] = 'e';
s[2] = 'l';
s[3] = 'l';
s[4] = 'o';
s[5] = '\0';
```

- **Displaying a string:**

- ◊ When you want to refer to a string stored in an array, **you just use the array name by itself.**

- you don't have to specify indices:
      - because this will specify a single character.
        - **if you want the entire string you just use the array name.**

- ◊ To display a string as output using the **printf()** function, you do the following:

- ```
printf("\nThe message is: %s", message);
```

- for a **character string variable** we use the following format specifier:
 - **%s**
 - you don't treat like other arrays when you wanna access it.

- ◊ the **%s** format specifier is for outputting a null-terminated string.

- ◊ The **printf()** function assumes when it encounters the **%s** format character that the corresponding argument is a character string that is terminated by a null character.

- **Inputting a String:**

- ◊ To input a string via the keyboard, use the **scanf()** function:

- code example:

```
char input[10];
printf("Please input your name: ");
scanf("%s", input);
```

- ◊ **The %s format specifier is for inputting strings.**

- ◊ Notice something here, there's no **ampersand symbol (&)**:

- **there's no need to use the & (address of operator) on a string.**

- the reason for this will be more clear when we talk about pointers.

- char input even though it's a character array, it's a pointer and **scanf()** as its second argument takes a pointer.

- ⇒ when you declare **integers** and **doubles** those are not pointers.

- so if you want to read input from the keyboard using **scanf()** the second argument

expects a pointer you have to pass in an address.

- ◊ With the **scanf()** when you do a string the **scanf()** is only gonna read up until a space:
 - so if you do **scanf()** and type several words in there separated by spaces:
 - the only thing that's gonna be stored in input is the first word or character:
 - because once it hits that space it's not gonna read the rest of the string.
 - there's other functions you can use to read the entire string, like:

- `fgets();`

- ◊ which is another way to read input from the keyboard, and it'll read the entire string regardless of spaces.

- **Testing if two strings are equal:**

- ◊ You cannot directly test two strings to see if they are equal with a statement such as:

- `if(string1 == string2)`
- ...

- ◊ The **equality operator can only be applied to simple variable types**, such as:

- **floats**
- **ints**
- **chars**
- doesn't work on structures or arrays.
 - you can't compare two arrays.

- ◊ To determine if two strings are equal, you must explicitly compare the two character strings character by character:

- we will discuss an easier way with the **strcmp()** function.

- ◊ **Reminder:**

- the string constant "x" is not the same as the character constant 'x':
 - 'x' is a basic type (char).
 - "x" is a derived type, an array of type char.
 - "x" really consists of two characters:
 - 'x' and '\0', the null character

- **String declaration code example:**

```
#include <stdio.h>

int main(void) {
    char str1[] = "To be or not to be";
    char str2[] = ", that is the question";
    unsigned int count = 0;           // Stores the string length

    while(str1[count] != '\0')        // Increment count till we reach the
                                    // terminating character
        ++count;

    printf("The length of the string \"%s\" is %d characters.\n", str1, count);

    count = 0;                      // Reset count for next string
    while(str2[count] != '\0')        // Count characters in second string
        ++count;

    printf("The length of the string \"%s\" is %d characters.\n", str2, count);
    return 0;
}
```

- you'll never have to do this (create code to check the size of a char array) in the real world because instead we could make use of the **strlen()** helper function.

76. Constant Strings

- A string constant is inside double quotes.

- **Constant Strings:**

- ◊ Sometimes you need to use a constant in a program:
 - the values in your variables will never change.

- ◊ Example:

- `circumference = 3.14159 * diameter;`

- ◊ The constant (in the previous example) represents the world-famous constant pi.

- ◊ There are good reasons to use a symbolic constant instead of just typing in the number:

- a name tells you more than a number does.
 - example:

- `owed = 0.015 * housevalue;`
 - `owed = taxrate * housevalue;`

- ◊ If you read through a long program, the meaning of the second version is plainer.

- ◊ Suppose you have used a constant in several places, and it becomes necessary to change its value:

- you only need to alter the definition of the symbolic constant, rather than find and change every occurrence of the constant in the program.

- **#define:**

- ◊ The preprocessor lets you define constants.
 - this is a preprocessor directive.

- ◊ Code example:

- `#define TAXRATE 0.015`

- ◊ When your program is compiled, the value 0.015 will be substituted everywhere you have used **TAXRATE**:

- **compile-time substitution.**

- the pre processor runs before the compiler.

◊ A **defined name** is **not a variable**:

- you cannot assign any other values to it.
- if it were a variable you'd be able to assign data to it, but not in this case because it's a pre processor directive.

◊ Notice that the **#define** statement has a special syntax:

- no equals sign used to assign the value **0.015** to **TAXRATE**.
- no semicolon.
- it's not a statement.

◊ **#define** statements can appear anywhere in the program:

- no such thing as a local define.
- most programmers group their **#define** statements at the beginning of the program (or inside an include file) where they can be quickly referenced and shared by more than one source file.
 - because it's running at the pre processor it doesn't have any scope, it doesn't have any local or global.
 - you can just put it anywhere in the program, and anywhere where the program can see it, even if it's being used before it's defined.

◊ The **#define** statement helps to make programs more portable:

- it might be necessary to use constant values that are related to the particular computer on which the program is running.

◊ The **#define** statement can be used for **character and string constants**, not just for numbers:

- code example:

```
#define BEEP '\a'  
#define TEE 'T'  
#define ESC '\033'  
#define OOPS "Now you have done it!"
```

→ **you just enclose the single characters in single quotes.**

→ **if you want to put an entire string you enclose it in double quotes.**

• **const:**

◊ **C90** added a **second way to create symbolic constants**:

- using the const keyword to convert a declaration for a variable into a declaration for a constant:

- it's called the constant modifier.

◊ This will allow never to change that variable:

- so all you do is the following:

```
const int MONTHS = 12; // MONTHS IS A  
SYMBOLIC CONSTANT FOR 12
```

→ use the word:

⇒ **const**

→ then the data type:

⇒ **int, double**, etc.

→ and the name of the variable.

◊ We're not calling it a variable, because a variable can change, that's why it's called variable.

◊ You either have constants or variables.

◊ In the previous example:

- **const** makes **MONTHS** into a read-only value:

- you can display **MONTHS** and use it in calculations.
- you cannot alter the value of **MONTHS**.

◊ **const** is a newer approach and is more flexible than using **#define**:

- it lets you declare a type.
- it allows better control over which parts of a program can use the constant.

◊ **C has yet a third way to create symbolic constants:**

- **enums**

◊ Initializing a **char array** and declaring it as a constant is a good way of handling standard messages:

- code example:

```
- const char message[] = "The end of the world is nigh.;"
```

→ because you declare messages as **const**, it's protected from being modified explicitly (by doing a **strcpy()**) within the program:

⇒ any attempt to do so will result in an error message from the compiler.

◊ This technique for defining standard messages is particularly useful if they are used in many places within a program:

- it prevents accidental modification of such constants in other parts of the program.

- **In summary:**
- Constants are important if you do not want data to change.
- They also can change magic numbers.

77. Common String Functions

- **String functions:**

- ◊ The standard library comes with a lot of functions to perform different operations on strings.
- ◊ C provides many functions specifically designed to work with strings.
- ◊ Some of the more commonly performed operations on character strings include:
 - getting the length of a string:
 - **strlen()**
 - copying one character string to another:
 - **strcpy()** and **strncpy()**
 - combining two character strings together (concatenation):
 - **strcat()** and **strncat()**
 - determining if two character strings are equal:
 - **strcmp()** and **strncmp()**
- ◊ The C library supplies these string-handling function prototypes in the **string.h** header file:
 - so you must include this file when calling or invoking any of these functions.

- **Getting the length of a string:**

- ◊ The **strlen()** function finds the length of a string:
 - returned as a **size_t** variable:
 - you can use the **size_t** variable interchangeably with an int.
- ◊ code example:

```
#include <stdio.h>
#include <string.h>

int main() {
    char myString[] = "my string";

    printf("The length of the string is: %d", strlen(myString));

    return 0;
}
```

- we could use this to validate, for example to avoid buffer overflows.
- ◊ This function does change the string:
 - the function header does not use const in declaring the formal parameter string.

- **Copying strings:**

- ◊ This is either modifying an existing string (an existing character array that has some data in it), or if it doesn't have anything in it, you wanna put something in there.

- ◊ Since you cannot assign arrays in C, you cannot assign strings either.

- ◊ You can use the **strcpy()** function to copy a string to an existing string:

- the string equivalent of the assignment operator.

- ◊ code example:

```
char src[50], dest[50];  
  
strcpy(src, "This is source");  
strcpy(dest, "This is destination");
```

- first parameter:

- where you're copying to.

- second parameter:

- the string that you're copying.

- ◊ In the previous example **src** would have the text **This is source** inside of it after passing it to the **strcpy()** function.

- same applies for **dest**.

- ◊ You can also put variables inside the string array as well.

- **Strncpy():**

- ◊ The **strcpy()** function does not check to see whether the source string actually fits in the target string:

- safer way to copy strings is to use **strncpy()**

- ◊ **strncpy()** takes a third argument:

- the maximum number of character to copy.

- ◊ code example:

```
char src[40];  
char src[12];  
  
memset(dest, '\0',  
      sizeof(dest));
```

```

strcpy(src, "Hello how are
you doing"); // THIS DOESN'T
CHECK IF THE ACTUAL

        // STRING IS
LARGER THAN src

strncpy(dest, src, 10); //
THIS WILL COPY src, BUT
IT'LL ONLY COPY THE FIRST
        //
10 CHARACTERS.
        //
THE THIRD ARGUMENT SHOULD
CORRESPOND TO THE SIZE OF
        //
THE DESTINATION.

```

- **memset()**:

- is setting the size of the data inside **dest** to null terminators:
- ⇒ that way when you add something it will automatically be there.

• Code example:

```

/*
    AUTHOR: JFITECH
    PURPOSE: strlen() FUNCTION EXAMPLE
    DATE: APRIL 1ST, 2022
*/



#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){

    char myString[] = "My name is Jason"; // ARRAY OF SIZE 17, THE CHAR
PLUS THE null character

    printf("\nThe length is: %d\n", strlen(myString));

    return 0;
}

```

◊ **strlen()**

◊ **strncpy()**:

```

/*
    AUTHOR: JFITECH
    PURPOSE: strncpy() FUNCTION EXAMPLE
    DATE: APRIL 1ST, 2022
*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {

    char myString[] = "My name is Jason"; // ARRAY OF SIZE 17, THE CHARACTERS
PLUS THE null character
    char temp[50];

    strncpy(temp, myString, sizeof(temp) - 1); // THE FIRST PARAMETER IS WHERE
WE WANNA COPY TO
                                                // SECOND PARAMETER IS WHAT WE
WANNA COPY
                                                // THIRD PARAMETER IS THE
DESTINATION
                                                // sizeof(temp) IS TO SAY THAT
WE DON'T WANNA COPY MORE THAN 50 BYTES
                                                // THIS IS A VERY IMPORTANT
CHARACTER BECAUSE THAT WAY WE'RE NEVER GONNA HAVE BUFFER OVERFLOWS

    printf("\nThe string is: %s\n", temp);

    return 0;
}

```

• String Concatenation:

- ◊ The **strcat()** function takes two strings for arguments:
 - a copy of the second string is tacked onto the end of the first.
 - this combined version becomes the new first string.
 - the second string is not altered.

- ◊ It returns the value of its first argument:
 - the address of the first character of the string to which the second string is appended.

- ◊ code example:

```

char myString[] = "my string";
char input[80];

printf("Enter a string to be concatenated: ");
scanf("%s", input);

printf("\nThe string %s concatenated with %s is::::\n", myString, input);
printf("%s", strcat(input, myString));

```

- ◊ The **strcat()** function does not check to see whether the second string will fit in the first array:
 - if you fail to allocate enough space for the first array, you will run into problems as excess

characters overflow into adjacent memory locations.

- **strncat()**

- ◊ Use **strncat()** instead:
 - takes a second argument indicating the maximum number of characters to add.

- ◊ For example:

- ```
strncat(bugs, addon, 13);
```

- will add the contents of the **addon** string to **bugs**, stopping when it reaches 13 additional characters or the null character, whichever comes first.

```
char src[50], dest[50];

strcpy(src, "This is source");
strcpy(dest, "This is destination");

strncat(dest, src, 15);

→ printf("Final destination string: %s", dest);
```

- **Comparing Strings:**

- ◊ Suppose you wanna compare someone's response to a stored string:
  - cannot use `==`, will only check (for references) to see if the string has the same address.
- ◊ There is a function that compares string contents, not string addresses:
  - it is the **strcmp()** (for string comparison) function.
  - does not compare arrays, so it can be used to compare strings stored in arrays of different sizes.
  - does not compare characters:
    - you can use arguments such as "**apples**" and "**A**", but you cannot use character arguments such as '**A**'.
- ◊ This function does for strings what relational operators do for numbers:
  - it **returns 0** if its **two string arguments are the same and nonzero otherwise**.
  - it returns **value < 0** when it indicates **str1** is less than **str2**.
  - it returns **value > 0** when it indicates **str1** is greater than **str2**.
- ◊ Example:
  -

```

int main() {

 printf("strcmp(\"A\", \"A\") is ");
 printf("%d\n", strcmp("A", "A"));

 printf("strcmp(\"A\", \"B\") is ");
 printf("%d\n", strcmp("A", "B"));

 printf("strcmp(\"B\", \"A\") is ");
 printf("%d\n", strcmp("B", "A"));

 printf("strcmp(\"C\", \"A\") is ");
 printf("%d\n", strcmp("C", "A"));

 printf("strcmp(\"Z\", \"a\") is ");
 printf("%d\n", strcmp("Z", "a"));

 printf("strcmp(\"apples\", \"apple\") is ");
 printf("%d\n", strcmp("apples", "apple"));
}

```

- the output for this should be:

```

C:\Users\jfedin\Downloads\Test\bin\Debug\Test.exe Something

strcmp("A", "A") is 0
strcmp("A", "B") is -1
strcmp("B", "A") is 1
strcmp("C", "A") is 1
strcmp("Z", "a") is -1
strcmp("apples", "apple") is 1

Process returned 0 (0x0) execution time : 3.707 s
Press any key to continue.

```

⇒ capital letters are always less than lowercase ones.

- ◊ The **strcmp()** function compares strings until it finds corresponding characters that differ:
  - could take the search to the end of one of the strings.
  
- ◊ The **strncmp()** function compares the strings until they differ or until it has compared a number of characters specified by a third argument:

- if you wanted to search for strings that begin with "astro", you could limit the search to the first five characters.

- code example:

```
if(strncmp("astronomy", "astro", 5) == 0) {
 printf("Found: astronomy");
}

if(strncmp("astounding", "astro", 5) == 0) {
 printf("Found: astounding");
}
```

→ we use **== 0** to say that it's equal.

→ in the second case, when comparing to the string **astounding**:

- it's not gonna find it, it's actually gonna say that **astounding is greater than astro**:

**astro**:

- ◊ so it's gonna display something greater than zero, which is a 1:
  - it's not gonna display any data.

- ◊ Use **strcmp()**:

- when comparing strings for all the contents.

- ◊ Use **strncmp()**:

- when you only wanna compare a pre-fix to a string.

# **78. Searching, Tokenizing, and Analyzing Strings**

- **Overview:**

- ◊ Let's discuss some more string functions.
- ◊ Searching a string:
  - the **string.h** header file declares several string-searching functions for finding **a single character or a substring**:
    - **strchr()**
      - &
    - **strstr()**
  - ◊ Tokenizing a string:
    - a token is a sequence of characters within a string that is bounded by a delimiter (space, comma, period, etc.).
    - breaking a sentence into words is called **tokenizing**.
    - **strtok()**
  - ◊ Analyzing strings:
    - **islower()**
    - **isupper()**
    - **isalpha()**
    - **isdigit()**
    - etc.
      - these are mainly done on characters (characters inside strings), you can also apply them to entire strings.

- **Concept of a pointer:**

- ◊ We are going to discuss in detail, the concept of a pointer in an upcoming section.
  - however, in order to understand some of these string functions, I want to give you a quick peek on this concept.
- ◊ C provides a remarkably useful type of variable called a pointer:
  - **a variable that stores an address.**
  - its value is the address of another location in memory that can contain a value.
  - we have used addresses in the past with the **scanf()** function.
- ◊ Some other languages will automatically manage memory for you:

- it'll allocate memory for you and delete it → **garbage collector**

◊ C doesn't do garbage collection for you, so when you create memory for using pointers, you have to delete that memory.

◊ The pointer just points to an address and that address contains a value.

◊ To create an integer by itself is not a pointer but getting the address of it, it now becomes a pointer.

◊ code example:

```
int Number = 25;
int *pNumber = &Number;
```

- above, we declared a variable, **Number**, with the value 25.

→ if we get the ampersand of it (**&Number**) we're getting its address, so then we can assign it to a pointer.

- we declared a pointer, **pNumber**, which contains the address of the variable **Number**:

→ asterisk is used in declaring a pointer.

◊ In the previous example to get the value of the variable **pNumber**, you can use the asterisk to dereference the pointer:

- if you wanna get the value of what the address is pointing to you have to dereference it.

```
*pNumber = 25;
```

- \* is the dereference operator, and its effect is to access the data stored at the address specified by a pointer.

◊ A pointer has special syntax in C:

- the asterisk character \* represents a pointer.

◊ A variable can have an address and the address can then have a value.

◊

```
int Number = 25;
int *pNumber = &Number;
```



- ◊
  - the value of **&Number** is the address where **Number** is located:
    - this value is used to initialize **pNumber** in the second statement.
  - many of the string functions return pointers:
    - this is why I wanted to briefly mention them.
    - do not worry if this concept does not sink in right now, we are going to cover pointers in a ton of detail in an upcoming section.
- ◊ You have to specify the type of the pointer, even though pointer is an address, **you still have to specify the type of where the address is pointing to**.
- ◊ In the previous image in order to get the value of **pNumber** you'd have to dereference the **1000** which would give you **25**.
- ◊ In the past when we used certain functions (strcpy and strcmp) we passed in pointers but it never returned pointers, we never had to use that pointer, now that we have to use the pointer it just points to an address:
  - and then we can use that address to do parsing or whatever we need to do.

- **Searching a string for a character:**

- The **strchr()** function searches a given string for a specified character:
  - first argument to the function is the **string to be searched** (which **will be the address of a char array**)
  - second argument is the **character that you are looking for.**
- The function will search the string starting at the beginning and return a pointer to the first position in the string where the character is found:
  - here's where pointers become important because we're not returning an entire value:
    - we're returning a pointer to a character inside of a string.
    - the address represents the index of an array.
  - the address of this position in memory is of type **char\*** (pronounced: **character star**) described as the "**pointer to char**".
    - **pointer to char** → is basically a single element in memory.
- To store the value that's returned, you must create a variable that can store the address of a character:
  - so you have to create a pointer.
- If the character is not found, the function returns a special value **NULL**:
  - **NULL** is the equivalent of **0** for a pointer and **represents a pointer that does not point to anything.**
    - **NULL** is just saying that it didn't find anything, so it's not pointing to anything.
    - we use **NULL** when we talk about pointers.
- **strchr()**
  - you can use the **strchr()** function like this:

```
char str[] = "The quick brown fox"; // THE STRING TO BE SEARCHED
char ch = 'q'; // THE CHARACTER WE ARE LOOKING FOR
char *pGot_char = NULL; // POINTER INITIALIZED TO NULL
- pGot_char = strchr(str, ch); // STORES ADDRESS WHERE ch IS FOUND
```
  - we create a pointer because it allows us to see where on the string it found the character.
    - we initialized the pointer to **NULL** because right now is not pointing to anything.
    - on the last line:
      - ⇒ we call **strchar()**
      - ⇒ we pass in the string "**The quick brown fox**".
      - ⇒ we pass in the single character which is '**q**'
      - ⇒ what's gonna happen is:
        - it's gonna look inside the string and the first time it finds a '**q**', it's gonna return the address of that character.

⇒ **pGot\_char** is gonna **be pointing to the fourth index of str[]**

- ◊ The first argument to **strchr()** is the address of the first location to be searched:
  - second argument is the character that is sought (**ch**, which is of type **char**).
    - expects its second argument to be of type **int**, so the compiler will convert the value of **ch** to this type.
      - it'll get the ASCII value
    - could just as well define **ch** as type **int**:
      - `int ch = 'q';`
    - **pGot\_char** will point to the value:
      - "quick brown fox"

- **Searching for a substring:**

- ◊ The **strstr()** function is probably the most useful of all the searching functions:
  - searches one string for the first occurrence of a substring.
  - returns a pointer to the position in the first string where the substring is found
  - if no match, returns **NULL**.
- ◊ The first argument is the string that is to be searched.
- ◊ The second argument is the substring you're looking for.
- ◊ code example:

```
char text[] = "Every dog has his day";
char word[] = "dog";
char *pFound = NULL;
pFound = strstr(text, word);
```
- ◊ Searches text for the first occurrence of the string stored in **word**:
  - the string "**dog**" appears starting at the seventh character in text.
  - **pFound** will be set to the address of **text + 6** ("dog has his day"):
    - it'll start there then it's gonna go to the **null terminator**.
  - search is case sensitive, "**Dog**" will not be found.

- **Tokenizing a string:**

- ◊ A token is a sequence of characters within a string that is bound by a delimiter.
- ◊ A delimiter can be anything, but, should be unique to the string:
  - spaces
  - commas
  - and a period are good examples.
- ◊ Breaking a sentence into words is called tokenizing.
- ◊ The **strtok()** function is used for tokenizing a string.
- ◊ It requires two arguments:
  - string to be tokenized.
  - a string containing all the possible delimiter characters.
- ◊ code example:

```
int main() {
 char str[80] = "Hello how are you - my name is - jason";
 const char s[2] = "-";
 char *token;

 /* GET THE FIRST TOKEN */
 token = strtok(str, s);

 /* WALK THROUGH OTHER TOKENS */
 while(token != NULL) {
 printf("%s\n", token);

 token = strtok(NULL, s)
 }

 return(0);
}
```

- **Analyzing strings:**

- ◊

| Function    | Tests for                                                           |
|-------------|---------------------------------------------------------------------|
| islower ()  | Lowercase letter                                                    |
| isupper ()  | Uppercase letter                                                    |
| isalpha ()  | Uppercase or lowercase letter                                       |
| isalnum ()  | Uppercase or lowercase letter or a digit                            |
| iscntrl ()  | Control character                                                   |
| isprint ()  | Any printing character including space                              |
| isgraph ()  | Any printing character except space                                 |
| isdigit ()  | Decimal digit ('0' to '9')                                          |
| isxdigit () | Hexadecimal digit ('0' to '9', 'A' to 'F', 'a' to 'f')              |
| isblank ()  | Standard blank characters (space, '\t')                             |
| isspace ()  | Whitespace character (space, '\n', '\t', '\v', '\r', '\f')          |
| ispunct ()  | Printing character for which isspace () and isalnum () return false |

◊

- ◊ The argument to each of these functions is the character to be tested.
  - ◊ All these functions return a nonzero value of type **int** if the character is within the set that's being tested for.
  - ◊ These return values convert to **true** and **false**, respectively, so you can use them as **Boolean** values.
  - ◊ code example:
- ```

char buff[100]; //INPUT BUFFER
int nLetters = 0; // NUMBER OF LETTERS IN INPUT
int nDigits = 0; // NUMBER OF DIGITS IN INPUT
int nPunct = 0; // NUMBER OF PUNCTUATION CHARACTERS

printf("Enter an interesting string of less than %d characters:\n", 100);
    
```

```
scanf("%s", buff); // READ A STRING INTO BUFFER

int i = 0;          // BUFFER INDEX
while(buff[i]) {
    if(isalpha(buff[i]))
        ++nLetters; // INCREMENT LETTER COUNT
    else if(isdigit(buff[i]))
        ++nDigits;   // INCREMENT DIGIT COUNT
    else if(ispunct(buff[i]))
        ++nPunct;
    ++i;
}

printf("\nYour string contained %d letters, %d digits and %d punctuation
characters.\n", nLetters, nDigits, nPunct);
```

79. Converting Strings

- In this lecture we're gonna talk about converting strings, some more string functions.

- **Converting Strings:**

- ◊ It is very common to convert character case:
 - to all upper or lower case.
- ◊ The **toupper()** function converts from **lowercase** to **uppercase**.
- ◊ The **tolower()** function converts from **uppercase** to **lowercase**.
- ◊ Both functions return either the converted character or the same character for characters that are already in the correct case or are not convertible such as punctuation characters.
- ◊ This is how you convert a string to uppercase:
 - `for(int i = 0; (buf[i] = (char)toupper(buf[i])) != '\0'; ++i);`
 - this are utilized on **character by character** basis.
- ◊ The previous loop will convert the entire string in the buf array to uppercase by stepping through the string one character at a time:
 - the loop stops when it reaches the string termination character '\0'.
 - **the cast to type char is there because toupper() returns type int.**
- ◊ You can use the **toupper()** function in combination with the **strstr()** function to find out whether one string occurs in another, ignoring case.
- ◊ Case conversion example:

```
char text[100]; // INPUT BUFFER FOR STRING TO BE SEARCHED
char substring[40]; // INPUT BUFFER FOR STRING SOUGHT

printf("Enter the string to be searched (less than %d characters):\n", 100);
scanf("%s", substring);

printf("\nEnter the string sought (less than %d characters):\n", 40);
scanf("%s", substring);

printf("\nFirst string entered:\n%s\n", text);
printf("Second string entered:\n%s\n", substring);

// CONVERT BOTH STRING TO UPPERCASE
for(i = 0; (text[i] = (char)toupper(text[i])) != '\0'; ++i);
for(i = 0; (substring[i] = (char)toupper(substring[i])) != '\0'; ++i);
```

```
printf("The second string %s found in the first.\n", (strstr(text, substring)
== NULL) ? "was not": "was");
```

- ◊ one of the limitations of the **strstr()** function is that is case sensitive.

• Converting Strings to Numbers:

- ◊ The **stdlib.h** header file declares functions that you can use to convert a string to a numerical value:

| Function | Returns |
|----------|--|
| atof() | A value of type double that is produced from the string argument. Infinity as a double value is recognized from the strings "INF" or "INFINITY" where any character can be in uppercase or lowercase and 'not a number' is recognized from the string "NAN" in uppercase or lowercase. |
| atoi() | A value of type int that is produced from the string argument. |
| atol() | A value of type long that is produced from the string argument. |
| atoll() | A value of type long long that is produced from the string argument. |

- for all four functions, the leading whitespace is ignored:

```
char value_str[] = "98.4";
→ double value = atof(value_str)
```

- **atof()** is basically an **ASCII** character to a float.

- ◊ When you have character strings that **have numbers** you can convert them using these **ASCII to** functions.

- ◊ This is all declared in the **stdlib.h** header file.

- ◊ You'll also see the opposite of it for creating **integers to strings**, like:

- **itoa()**

- which is **int to ASCII**.

→ so if you had a number like **98** and you want to **convert it to a string**, you would use the previous function.

- ◊ There's also some newer functions:
 - these ones are specifically for **string to double**:

| Function | Returns |
|-----------|--|
| strtod() | A value of type <code>double</code> is produced from the initial part of the string specified by the first argument. The second argument is a pointer to a variable, <code>ptr</code> say, of type <code>char*</code> in which the function will store the address of the first character following the substring that was converted to the <code>double</code> value. If no string was found that could be converted to type <code>double</code> , the variable <code>ptr</code> will contain the address passed as the first argument. |
| strtof() | A value of type <code>float</code> . In all other respects it works as <code>strtod()</code> . |
| strtold() | A value of type <code>long double</code> . In all other respects it works as <code>strtod()</code> . |

- ◊ The **ASCII** ones are the traditional ones, the others:

- **strtod()**
- **strtof()**

- were added in later standards of C.

- ◊ Here's an example of doing some conversions:

```
double value = 0;
char str[] = "3.5 2.5 1.26"; // THE STRING TO BE CONVERTED
char *pstr = str;           // POINTER TO THE STRING TO BE CONVERTED
char *ptr = NULL;           // POINTER TO THE CHARACTER POSITION AFTER CONVERSION

while(true) {
    value = strtod(pstr, &ptr); // CONVERT, STARTING AT pstr
    if(pstr == ptr)           // pstr STORED IF NO CONVERSION...
        break;
    else {
        printf(" %f", value); // OUTPUT THE RESULTANT VALUE
        pstr = ptr;           // STORE START FOR THE NEXT CONVERSION
    }
}
```

- on line 7, we use **pstr** instead of **str** because we can't compare an address against a string.

80. (Challenge) Understanding char arrays

- In this challenge you are going to write a program that tests your understanding of char arrays.

- **Requirements:**

- 1)
 - Write a function to count the number of character in a string (length):
 - cannot use the `strlen` function.
 - function should take a **character array as a parameter**.
 - should return an **int** (the length).

- ◊ You don't have to worry about pointers at all in this assignment.

- 2)
 - Write a function to concatenate two character strings:
 - cannot use the **`strcat()`** library function.
 - function should take **3 parameters**:
 - char result[]**
 - const char str1[]**
 - const char str2[]**
 - can return void**

- 3)
 - Write a function that determines if two strings are equal:
 - cannot use the **`strcmp()`** function
 - function should take **two const char arrays** as parameters and return a **Boolean** of **true** if they are equal and **false otherwise**.
 - You're gonna have to iterate through the arrays and compare each character.

- You should probably look for the **null terminator** for the equal strings because it'll determine when you wanna stop, this is due to the fact that you don't have the sizes of these arrays.

- Compare the second string to the first.

My Code

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 11 - UNDERSTANDING CHAR ARRAYS - MY SOLUTION
DATE: APRIL 11TH, 2022
*/

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// FUNCTION PROTOTYPES
void main(void); // MAIN FUNCTION
int numOfChars(const char stringX[]); // FUNCTION TO COUNT THE NUMBER OF
CHARACTERS IN A STRING
char concatStrings(const char str1[], const char str2[]); // FUNCTION TO
CONCATENATE TWO CHARACTER STRINGS
bool areStringsEqual(const char string1[], const char string2[]); // FUNCTION
TO DETERMINE IF TWO STRINGS ARE EQUAL

// MAIN FUNCTION
void main(void){
    // VARIABLE DECLARATION
    const char myString[] = "Hello, this is a test"; // CONTAINS 21 CHARACTERS
+ NULL TERMINATOR '\0' = 22 CHARACTERS
    const char myString2[] = "of the concat function";

    // FUNCTION CALLS
    // numOfChars(myString);
    // concatStrings(myString, myString2);
    // areStringsEqual(myString, myString2);

    // SCREEN OUTPUT
    // numOfChars FUNCTION
    printf("\n1. The amount of character in the string \"%s\" is: %d",
myString, numOfChars(myString), "\n");

    // concatStrings FUNCTION
    // printf("\n2. The result of the concatenated strings is: \n%s",
concatStrings(myString, myString2));

    // areStringsEqual FUNCTION
    printf("\n3. The result of the comparisson between the strings: \n %s",
myString, "\nand %s", myString2, "\nis: %u", areStringsEqual(myString,
myString2));
}

// FUNCTION TO COUNT THE NUMBER OF CHARACTERS IN A STRING
int numOfChars(const char stringX[]){
    // VARIABLE DECLARATION
    int counter = 0;

    // LOOP TO ITERATE THROUGH THE ARRAY
    for(int i = 0; i < stringX[i]; ++i){
```

```

    ++counter;
}

return counter;
}

// FUNCTION TO CONCATENATE TWO CHARACTER STRINGS
char concatStrings(const char str1[], const char str2[]){
    // VARIABLE DECLARATION

    // FUNCTION CALL
    int str1Length = numOfChars(str1); // CALLS THE numOfChars() FUNCTION ON
str1 AND STORES THE RETURNED VALUE ON str1Length
    int str2Length = numOfChars(str2); // CALLS THE numOfChars() FUNCTION ON
str2 AND STORES THE RETURNED VALUE ON str2Length */

    char result[str1Length + str2Length];

    return (int)result;
}

// FUNCTION TO DETERMINE IF TWO STRINGS ARE EQUAL
bool areStringsEqual(const char string1[], const char string2[]){
    // VARIABLE DECLARATION
    bool areEqual = false;

    // LOOP TO ITERATE THROUGH THE ARRAYS AND COMPARE EACH CHARACTER
    for(i = 0; i < string1; ++i){
        for(j = 0; j < string2; ++j){
            if(string[i] == string[j])
        }
    }

    return areEqual;
}

```

81. (Demonstration) Understanding char arrays

- The reason we're using **const** variables here is because we don't wanna modify the string inside the function, we just wanna parse it and return the length.

Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int stringLength(const char string[]);
void concat(char result[], const char str[], const char str1[]);
bool equalStrings(const char s1[], const char s2[]);

int main()
{
    const char word1[] = "jason";
    const char word2[] = "ok";
    const char word3[] = "whatever";
    char result[50];

    printf("%d      %d\n", stringLength(word1), stringLength(word2),
stringLength(word3));

    concat(result, word1, word2);

    printf("\n%s", result);

    printf("\n%d\n", equalStrings("Jason", "Jason"));
    printf("%d\n", equalStrings("Jasons", "Jason"));

    return 0;
}

int stringLength(const char string[])
{
    int count = 0;

    while (string[count] != '\0')
        ++count;

    return count;
}

void concat(char result[], const char str1[], const char str2[])
{
    int i, j;

    for (i = 0; str1[i] != '\0'; ++i)
    {
        result[i] = str1[i];
    }

    for (j = 0; str2[j] != '\0'; ++j)
        result[i + j] = str2[j];

    result[i+j] = '\0';
}

bool equalStrings(const char s1[], const char s2[])
{
    int i = 0;
    bool isEqual = false;
```

```
while ( s1[i] == s2[i] &&
        s1[i] != '\0' &&
        s2[i] != '\0' )
    ++i;

if ( s1[i] == '\0' && s2[i] == '\0' )
    isEqual = true;
else
    isEqual = false;

return isEqual;
}
```

82. (Challenge) Utilizing common string functions

- This challenge mainly has to do with string functions.
- It will get you familiar with **strlen()** and **strcmp()**.
- This challenge will help you better understand how to use the most common string functions in the string library.

• Requirements:

- 1)
 - Write a program that displays a string in reverse order:
 - should read input from the keyboard.
 - need to use the **strlen()** function.
 - once you have the size of the string then you can start reversing the characters.
- 2)
 - Write a program that sorts the strings of an array using **bubble sort**:
 - research the **bubble sort algorithm**.
 - need to use the **strcmp()** and **strcpy()** functions.
 - What it could look like:
 -

Input number of strings :3

Input string 3 :

zero

one

two

Expected Output :

The strings appears after sorting :

one

two

zero

My Code

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 12 - COMMON STRING FUNCTIONS - MY SOLUTION
DATE: APRIL 15TH, 2022
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// FUNCTION PROTOTYPES
char reverseString(const char string[]); // REVERSE STRING FUNCTION

// MAIN FUNCTION
void main(void){
    // VARIABLES DECLARATION
    char string1[30];

    // REQUEST USER INPUT
    printf("\nPlease enter the string to be reversed: ");
    scanf("%s", string1);

    // int stringLength = strlen(string1);

    // printf("\n--> The size of the string is: %d", stringLength, "---\n");

    printf("\nThe string %s in reverse order is: ", reverseString(string1));

    return;
}

// REVERSE STRING FUNCTION
char reverseString(const char string[]){
    // VARIABLES DECLARATION
    int i = 0;
    int stringLength = strlen(string);
    char reversedString[stringLength];

    while (string[i] != '\0'){
        reversedString[stringLength] = string[i];
        reversedString[stringLength - 1];
    }

    return reversedString;
}
```

83. (Demonstration) Utilizing common string functions

- In:

- ◊ `char name[25][50];`

- we're creating 25 strings of size 50.
 - it's a 2D array with 25 rows and 50 columns.

- We use the following to avoid **Buffer Overflows**:

```
strncpy(temp, name[j], sizeof(temp) - 1);
strncpy(name[j], name[j+1], sizeof(temp) - 1);
strncpy(name[j+1], temp, sizeof(temp) - 1);
```

-

Instructor's Code

□

Section 11: Debugging

84. Overview

- This is a very important concept that you need to understand.
- Specifically you need to understand some of the tools and some of the methods that you can employ to find and fix bugs.

- **Overview:**

- ◊ Debugging is the process of finding and fixing errors in a program (usually logic errors, but, can also include compiler/syntax errors)
 - for syntax errors, understand what the compiler is telling you.
 - **always focus on fixing the first problem detected.**
 - the first error could be causing the other errors, so the other errors can be very misleading.
- ◊ Can range in complexity from fixing simple errors to collecting large amounts of data for analysis.
 - sometimes you don't have any information provided for you, except logging.
 - you can have bugs that aren't repeatable, because they happen in such rare occasions.
- ◊ The ability to debug by a programmer is an essential skill (problem solving) that can save you tremendous amounts of time (and money).
- ◊ This skill is developed with practice.
- ◊ Maintenance phase is the most expensive phase of the software life cycle.
- ◊ Understand that bugs are unavoidable.
 - all software have bugs.
 - nobody can write perfect code.
- ◊ Just because a software has bugs it doesn't mean it's poorly written code.

- **Common Problems:**

- ◊ Common problems when you're trying to debug are:
 - Logic errors:
 - functionality is broken but for the compiler everything looks good.

- Syntax errors
 - directly related to the compiler.
- Memory corruption.
 - this is also a common problem specially in C.
 - if you're not freeing your memory after allocating it correctly, you can have things like dangling references.
 - you can also have memory issues related to memory that's not being used.
 - you can have memory leaks.
- Performance / scalability:
 - this can be a problem where your functionality is intended as it is and your code syntactically is correct, but it's not very efficient.
 - or you don't have the hardware supported.
 - if you're in a server and you don't have any hardware to support other requests it's gonna run really really slow.
- Lack of Cohesion:
 - this can be a problem when your code that reduces quality, related to your code is doing too much or your function is doing too much, it's not focusing on one single thing.
 - or even an application has too much functionality, has features that are never used.
 - this would probably require a redesign.
- Tight coupling (dependencies):
 - this also has to do with quality of code.
 - this is where you have a lot of dependencies.
 - like using global variables in different functions, it can cause dependencies between those functions.
- **What you wanna strive for when you write code and you're writing high quality code is:**
 - you wanna have very high cohesion.**
 - and very low coupling.**

• Debugging Process:

- ◊ Understand the problem (sit down with tester, understand requirements):
 - don't dive directly into the code, understand what's going on.
- ◊ Reproduce the problem:
 - once you understand the bug try to reproduce it.

- sometimes is very difficult as problems can intermittent or only happen in very rare circumstances.
 - parallel processes or threading problems.
- ◊ Simplify the problem / Divide and conquer / isolate the source:
 - remove parts of the original test case.
 - comment out code / back out changes.
 - what you wanna do essentially though is turn a large program into a lot of small programs (unit testing).
 - unit testing might tell you where the problem is.
 - ◊ Identify origin of the problem (in the code).
 - use debugging tools if necessary.
 - ◊ The debugger that we're gonna use for C program debugging is **GDB**.
 - ◊ After you've identified the origin of the problem then you'll have to fix it:
 - solve the problem:
 - experience and practice.
 - sometimes include redesign or refactor of code.
 - ◊ Test!, Test!, Test!

• Techniques and Tools:

- ◊ Tracing / using print statements:
 - output values of variables at certain points of a program.
 - show the flow of execution.
 - can help isolate the error.
 - this is also useful when your program crashes, it can help you identify where it crashes.
- ◊ Debuggers - monitor the execution of a program, stop it, restart it, set breakpoints and watch variables in memory.
- ◊ Log files - can be used for analysis, add "good" log statements to your code.
 - these are very useful specially for bugs that cannot be recreated.
 - you can analyze them at a later date to see exactly what was going on when the bug happened.
 - this means that you'd have to provide good logging statements in your code itself in order for this to be useful.
 - ⇒ you may create your own log file and write to that file while the program is running,

so that if you do have an error you can look at those logs.

- ◊ Monitoring software - run-time analysis of memory usage, network traffic, thread and object information.
 - this can tell you maybe if you're allocating too much memory and not deleting if you have a tool for memory usage.
 - network traffic: whether you're sending too many packets out or if your software is scalable.

- **Common debugging tools:**

- ◊ **Exception Handling** helps a great deal to identify catastrophic errors:
 - many programming languages have exception handling, unfortunately C does not.
 - an exception handler will actually prevent crashes and do a lot of things at run-time.
 - things like **try, catch** blocks (these don't apply to C).
- ◊ **Static Analyzers** - analyze source code for specific set of known problems:
 - analyzes the program without running it.
 - **semantic checker**, does not analyze syntax.
 - can detect things like uninitialized variables, memory leaks, unreachable code, deadlocks or race conditions.
 - can also detect code that's not being executed / non used code.
- ◊ Test suites - run a set of comprehensive system end-to-end tests.
 - you can automate tests to run for example every time you do a new build.
- ◊ Debugging the program after it has crashed:
 - analyze the call stack.
 - analyze memory dump (core file).
 - the core file is generated when the program crashes (it depends on the kind of crash).
 - you can analyze the core file to look at the call stack to see exactly where that program died, it'll tell you the line of code where the program died.

- **Preventing Errors:**

- ◊ **Write high quality code** (follow good design principles and good programming practices):
 - things like:
 - using meaningful variable names.
 - writing inefficient code:

- using more loops than you need to.
- using an if statement where you don't need to do that.
- using a function that's doing a hundred things and it's not cohesive.
- organizing your code in a module fashion.
- having many dependencies between functions.

◊ **Unit tests** - automatically executed when compiling:

- helps avoid regression.
 - regression is when you write new code and you introduce a new bug in something that was previously working (you broke old code).
- finds errors in new code before it is delivered.
- **TDD** (Test Driven Development):
 - this is where you write your tests first and then you develop your code after.
 - the goal here is when you write your first test, your test is failing and then you write the code to get it to pass.
 - this will make you work in really small chunks, hence you will write very cohesive code, your code won't have a lot of dependencies.

◊ Provide good documentation and proper planning (write down design on paper and utilize pseudocode).

◊ Work in steps and constantly test after each step:

- avoid too many changes at once.
- when making changes, apply them incrementally. Add one change, then test thoroughly before starting the next step.
- helps reduce the possible sources of bugs, limits problem set.

85. Understanding the call stack

- How to read the call stack and further help debug your program.

- **Overview:**

- ◊ A stack trace (call stack) is generated whenever your app crashes because of a fatal error:
 - it uses the word **stack** because it's using a data structure.
- ◊ A stack trace shows a list of the function calls that lead to the error:
 - includes the filenames and line numbers of the code that caused the exception or error to occur.
 - the top of the stack contains the last call that causes the error (there could be nested calls).
 - bottom of the stack contains the first call that started the chain of calls to cause the error.
 - you need to find the call in your program that is causing the crash.
- ◊ The call stack contains a lot of good information.
 - ◊ When a C program crashes it may generate a **Segmentation Fault** and that doesn't tell you a lot of information, it just means something bad happened.
 - ◊ You don't always need to look at that core file, you can also analyze the call stack in GDB.
 - ◊ A programmer can also dump the stack trace.
 - ◊ If you don't actually know when the error is gonna happen you can always dump the **call stack** to see the current traces that led up to a particular call.
- When seeing the **call stack** specially in **GDB** in **Code Blocks**, it's gonna look something like this:
 - ◊

| Nr | Address | Function |
|----|----------------|--|
| 0 | | GcmdGtkFoldview::Model::iter_refresh(this=0xb442b8, _iter=0x7fff24e606f0) |
| 1 | 0x4b332f | GcmdGtkFoldview::control_refresh(this=0xb44000, ctxdata=0xb2ad50) |
| 2 | 0x4b3352 | GcmdGtkFoldview::Control::refresh(menu_item=0xaed540, ctxdata=0xb2ad50) |
| 3 | 0x7f521b69fe9d | IA__q_closure_invoke(closure=0xb6d800, return_value=0x0, n_param_values=1, param_values=0x7fff24e60990, invocation_hint=0x7fff24e60990) |
| 4 | 0x7f521b6b2bfd | signal_emit_unlocked_R(node=0x988f00, detail=0, instance=0xaed540, signal_id=619054832, emission_return=0x0, instance_and_params=0x7fff24e60990) |
| 5 | 0x7f521b6b40ee | IA__q_signal_emit_valist(instance=0xaed540, signal_id=619054832, var_args=0x7fff24e60bf0) |
| 6 | 0x7f521b6b45f3 | IA__q_signal_emit(instance=0xb442b8, signal_id=619054832, detail=0, var_args=0x7fff24e60bf0) |
| 7 | 0x7f521c8e2ceb | gtk_widget_activate() |
| 8 | 0x7f521c7d63ad | gtk_menu_shell_activate_item() |
| 9 | 0x7f521c7d8085 | ??() |
| 10 | 0x7f521c7c9848 | ??() |
| 11 | 0x7f521b69fe9d | IA__q_closure_invoke(closure=0x971dc0, return_value=0x7fff24e60f20, n_param_values=2, param_values=0x7fff24e60fe0, invocation_hint=0x7fff24e60990) |
| 12 | 0x7f521b6b28dc | signal_emit_unlocked_R(node=0x768180, detail=0, instance=0x9a12b0, emission_return=0x7fff24e611e0, instance_and_params=0x7fff24e60990) |
| 13 | 0x7f521b6b3f71 | IA__q_signal_emit_valist(instance=0x9a12b0, signal_id=<value optimized out>, detail=0, var_args=0x7fff24e61240) |
| 14 | 0x7f521b6b45f3 | IA__q_signal_emit(instance=0xb442b8, signal_id=619054832, detail=11900256) |
| 15 | 0x7f521c8de4de | ??() |
| 16 | 0x7f521c7c23d3 | gtk_propagate_event() |
| 17 | 0x7f521c7c341b | gtk_main_do_event() |
| 18 | 0x7f521c424fac | ??() |
| 19 | 0x7f521b0027ab | IA__q_main_context_dispatch(context=0x78e6c0) |
| 20 | 0x7f521b005f7d | q_main_context_iterate(context=0x78e6c0, block=1, dispatch=1, self=<value optimized out>) |
| 21 | 0x7f521b0064ad | IA__q_main_loop_run(loop=0xb00460) |
| 22 | 0x7f521c7c3837 | gtk_main() |

- this particular **call stack** has a lot of different calls so it must've been doing a lot of nesting and different function calls.
- but at the very bottom of the stack on **line 22**, it says **gtk_main()**:
 - it probably means that it executed the **main()** function
 - and then on **line 21** it looks like it executed a statement:
 - **main_loop_run**
 - the question marks on line 18 mean that there's no debugging information, it doesn't know what happened there:
- **?? ()**
 - ⇒ and that's why when you build your program you can either build for release or build it with debugging information.
 - when you build it with debugging information it contains extra information when it compiles so that when you run you can look at call stacks.
 - ⇒ when you build something for release the executable's a lot smaller and may not contain this information, it may contain more question marks.
 - this will also tell you if you invoked any libraries, any additional programs, it will call those and it will tell you the **call stack** from there.
 - at the very top is what's important because that will signify what cause the crash:
 - on this particular program the **iter_refresh** was causing the problem.

- You can also look at **call stacks** while the program is running to just see information so you can trace it back.
 - ◊ so if you put a break point inside of a function:
 - you run the debugger on that
 - the debugger's gonna stop at that breakpoint
 - and then you can bring up the call stack and it will tell you how you got to that line of code.

86. Code Blocks Debugger

- We're gonna learn how to use the debugger in the **Code Blocks IDE**.

- **Debugging Correct Settings:**

- ◊ If you wanna be able to debug your project you wanna make sure that you have the correct settings.

- you program needs to have debugging symbols:

- -g

- you'll also want to have **all common compiler warnings enabled**.

- these are the two options that you wanna have enabled for debugging.

- ◊ You also wanna make sure that none of the other settings are doing the optimizations switches, removing any symbols or doing anything like that.

- this is the **-s** option:

- you wanna make sure it's not enabled.

- ◊ By default nothing is really enabled so you should be good to go.

- ◊ None of the optimizations checks should be checked.

- Once you have the correct debugging settings you're good to go.

- **Debugging Mode:**

- ◊ The way you can debug your code is you start your program in debugging mode.

- ◊ You still wanna build it like normal, do a normal build.

- and that'll compile and add your debugging information.

- ◊ And when you wanna run it in the debugger you say **start / continue**

- ◊ If you don't have any breakpoints set the program's gonna run like normal and you won't see anything.

- **Breakpoints:**

- ◊ Breakpoints are a way to stop your program at a particular statement:

- so that you can look at the value of variables

- and also if you want to **step line by line** to see what's going on with each statement

and its logic.

- ◊ If you don't set any breakpoints your program is never gonna stop, so you're not gonna be able to view any variables and you're not gonna be able to look at the call stack.
- ◊ When you click **start / continue** it's gonna run just like any other program.
- ◊ If you don't have a breakpoint you could click on "**run to cursor**" and it'll set a breakpoint at wherever your cursor is.
- ◊ When you click on "**step out**":
 - you're stepping out of a function:
 - if you're right inside of a function you'll step out.
- ◊ When you click "**step into**":
 - you'll step into a function:
 - so if you have a breakpoint at a **printf()** you'd actually go inside the **printf()** function.
- ◊ When you click "**next instruction**":
 - you're gonna go to the next line of code (this is usually the most common).
- ◊ You can also **step into** an instruction as well if there's any functions inside of a function.

- **Debugging Windows Toolbar (CodeBlocks IDE):**

- ◊ These are also important.
- ◊ The main ones the instructor uses are:
 - **call stack**
 - **watches**
- ◊ You can also look at the memory that's being allocated (if there's any) on:
 - **Memory dump**
- ◊ Also you can look at the current running thread on:
 - **Running threads**
- ◊ You can look at the CPU registers on:
 - **CPU registers**
- ◊ And you can also look at the breakpoints that you have set, if you have set any:

- **Breakpoints**

- ◊ You can also set a breakpoint by clicking on the red dot that appears next to the line numbers:
 - the program when executed will run until it hits the breakpoint.
- ◊ There's no point in using the call stack if you're not inside of a function or no functions have been invoked.

- **Watches Window:**

- ◊ It displays all of the variables in your program and the data
- Use the debugger to solve problem that you wouldn't be able to solve with normal techniques.
 - ◊ or if your program crashes go line by line, once it crashes you'll know what line made it crash.

87. Common C Mistakes

- We'll look at some common mistakes while programming in C.

- **Misplacing or missing a semicolon:**

```
if (j == 100);  
◊     j = 0;
```

- the value of **j** will always be set to **0** due to the misplaced semicolon after the closing parenthesis:
 - semicolon is syntactically valid (it represents the null statement), and, therefore, no error is produced by the compiler.
 - this will cause a problem because **j** will never be set to **0**
 - same type of mistake is frequently made in while and for loops.

- **Confusing the operator = with the operator ==:**

```
if (a = 2)  
◊   printf("Your turn.\n")
```

- **==** when comparing strings in C we can't use this operator, otherwise we would be comparing references.
 - usually made inside an **if**, **while**, or **do** statement.
 - perfectly valid and has the effect of assigning **2** to **a** and then executing the **printf()** call.
 - **printf()** function will always be called because the value of the expression contained in the **if** statement will always be nonzero.

- **Omitting prototype declarations:**

```
◊ result = squareRoot(2);
```

- if **squareRoot** is defined later in the program, or in another file, and is not explicitly declared otherwise.
 - the compiler assumes that the function returns an int.
 - it's always safest to include a prototype declaration for all functions that you call (either explicitly yourself or implicitly by including the correct header file in your program).

- **Failing to include the header file that includes the definition for a C-programming library function being used in the program:**

```
◊ double answer = sqrt(value1);
```

- this will generate a linking problem with that function.
- if this program does not **#include** the **<math.h>** file, this will generate an error that **sqrt()** is undefined.

- **Confusing a character constant and a character string:**

- ◊ `text = 'a';`
 - a single character is assigned to text.
- ◊ `text = "a";`
 - a pointer to the character string "a" is assigned to text.
 - this isn't valid in C either.
- ◊ in the first case, text is normally declared to be a **char** variable.
- ◊ in the second case, it should be declared to be of type "**pointer to char**".

- **Using the wrong bounds for an array:**

```
int a[100], i, sum = 0;
...
for(i = 1; i <= 100; ++i)
    sum += a[i];
```

- array out of bounds exception.
- valid subscripts of an array range from 0 through the number of elements minus one:
 - the preceding loop is incorrect because the last valid subscript of **a** is **99** and not **100**.
 - also probably intended to start with the first element of the array; therefore, **i** should have been initially set to 0.
- forgetting to reserve an extra location in an array for the terminating null character of a string:
 - when declaring character arrays they need to be large enough to contain the terminating null character.
 - the character string "**hello**" **would require six locations in a character array** if you wanted to store a null at the end.

- **Confusing the operator -> with the operator . when referencing structure members (this has to do with pointers):**

- ◊ the operator **.** is used for structure variables.
- ◊ the operator **->** is used for structure pointer variables.

- **Omitting the ampersand (&) before nonpointer variables in a `scanf()` call:**

```
int number;
...
◊ scanf("%i", number);
```

- all arguments appearing after the format string in a **scanf()** call must be pointers.

- **Using a pointer variable before it's initialized:**

```
char *char_pointer;  
◇ *char_pointer = 'X';
```

- you can only apply the indirection operator (which is the * before) to a pointer variable after you have set the variable pointing somewhere.
 - **char_pointer** is never set pointing to anything, so the assignment is not meaningful.
 - this would actually cause a segmentation fault.

- **Omitting the break statement at the end of a case in a switch statement:**

- ◇ if a break is not included at the end of a case, then execution continues into the next case.

- **Inserting a semicolon at the end of a preprocessor definition:**

- ◇ usually happens because it becomes a matter of habit to end all statements with a semicolon.

- ◇ this leads to a syntax error if used in an expression such as:

```
#define END_OF_DATA 999;  
- if(value == END_OF_DATA)  
  ...
```

→ the compiler will see the following statement after preprocessing:

```
if(value == 999;)  
⇒ ...
```

- this will cause an error.

- **Omitting a closing parentheses or closing quotation marks on any statement:**

```
total_earning = (cash + (investements * (inv_interest)) + (savings *  
sav_interest);  
◇ printf("Your total money to date is %.2f, total_earning);
```

- the use of embedded parentheses to set apart each portion of the formula makes for a more readable line of code.

- however, there is always the possibility of missing a closing parentheses (or in some occasions, adding one too many).

- the second line is missing a closing quotation mark for the string being sent to the **printf()** function.

- both of these will generate a compiler error:

- sometimes the error will be identified as coming on a different line.

- depending on whether the compiler uses a parentheses or quotation mark on a subsequent line to complete the expression which moves the missing character to a place later in the program.

88. Understanding Compiler Errors

- Common compiler errors and warnings.

- **Overview:**

- ◊ It is sometimes very hard to understand what the compiler is complaining about:
 - you need to understand compiler errors in order to fix them.
 - it is sometimes difficult to identify the true reason behind a compiler error.
- ◊ The compiler is doing half the job for you, it's checking the syntax telling you exactly what's wrong:
 - you should use that to your advantage.
- ◊ The compiler makes decisions about how to translate the code that the programmer has not written in the code:
 - is convenient because the programs can be written more succinctly (only expert programmers take advantage of this feature).
- ◊ You should use an option for the compiler to notify all case where there are implicit decision:
 - this option is:
 - **-Wall**
 - this is turned on by default in Code Blocks.
 - this will notify you of all the cases where the compiler is doing things you don't know about.
 - it'll also turn on all warnings.

- ◊ **The compiler shows two types of problems:**

- **Errors:**

- a condition that prevents the creation of a final program.
 - no executable is obtained until all the errors have been corrected.
 - the first errors shown are the most reliable because the translation is finished but there are some errors that may derive from previous ones.

- fix the errors that are first, it is recommended to compile again and see if other later errors also disappeared.

- **Warnings:**

- messages that the compiler shows about "special" situation in which an anomaly has been detected.
 - non-fatal errors.
 - the final executable program may be obtained with any number of warnings.
 - a warning should be fixed:

→ you're either doing something incorrectly or it could cause a problem in the future.

- ◊ Compile always with the **-Wall** option and do not consider the program correct until all warnings have been eliminated.

- **Most common compiler messages:**

- ◊ **'Variable' undeclared (first use in this function):**

- this is one of the most common and easier to detect.
 - the symbol shown at the beginning of the message is used but has not been declared.

- ◊ **Warning: implicit declaration of function '...':**

- this warning appears when the compiler finds a function used in the code but no previous information has been given about it.

- need to declare a function prototype.

- ◊ **Warning: control reaches end of non-void function:**

- this warning appears when a function has been defined as returning a result (non **void**) but no return statement has been included to return this result.

- either the function is incorrectly defined or the statement is missing.

- ◊ **Warning: unused variable '...':**

- this warning is printed by the compiler when a variable is declared but not used in the code.
 - message disappears if the declaration is removed.
 - you shouldn't be declaring a variable if you don't use it.

- ◊ **Undefined reference to '...':**

- appears when there is a function invoked in the code that has not been defined anywhere.
 - the compiler is telling us that there is a reference to a function with no definition.
 - check which function is missing and make sure its definition is compiled.

- ◊ **Error: conflicting types for '...':**

- two definitions of a function prototype have been found.
 - one is the prototype (the result type, name, parentheses including the parameters, and a semicolon).
 - the other is the definition with the function body.
 - the information in both places is not identical, and a conflict has been detected.
 - the compiler shows you in which line the conflict appears and the previous definition that caused the contradiction.

- **Run-time errors:**

- ◊ The execution of C programs may terminate abruptly (**crash**) when a run-time error is detected.
 - C programs only print the succinct message **Segmentation Fault** (this doesn't really tell you anything.)
 - usually results in a core file depending on the signal that has been thrown.
 - can analyze the core file and the call stack.
 - first analyze the **call stack**.
 - or
 - run the **debugger** line by line to see which line makes the program crash.

Section 12: Pointers

89. Overview

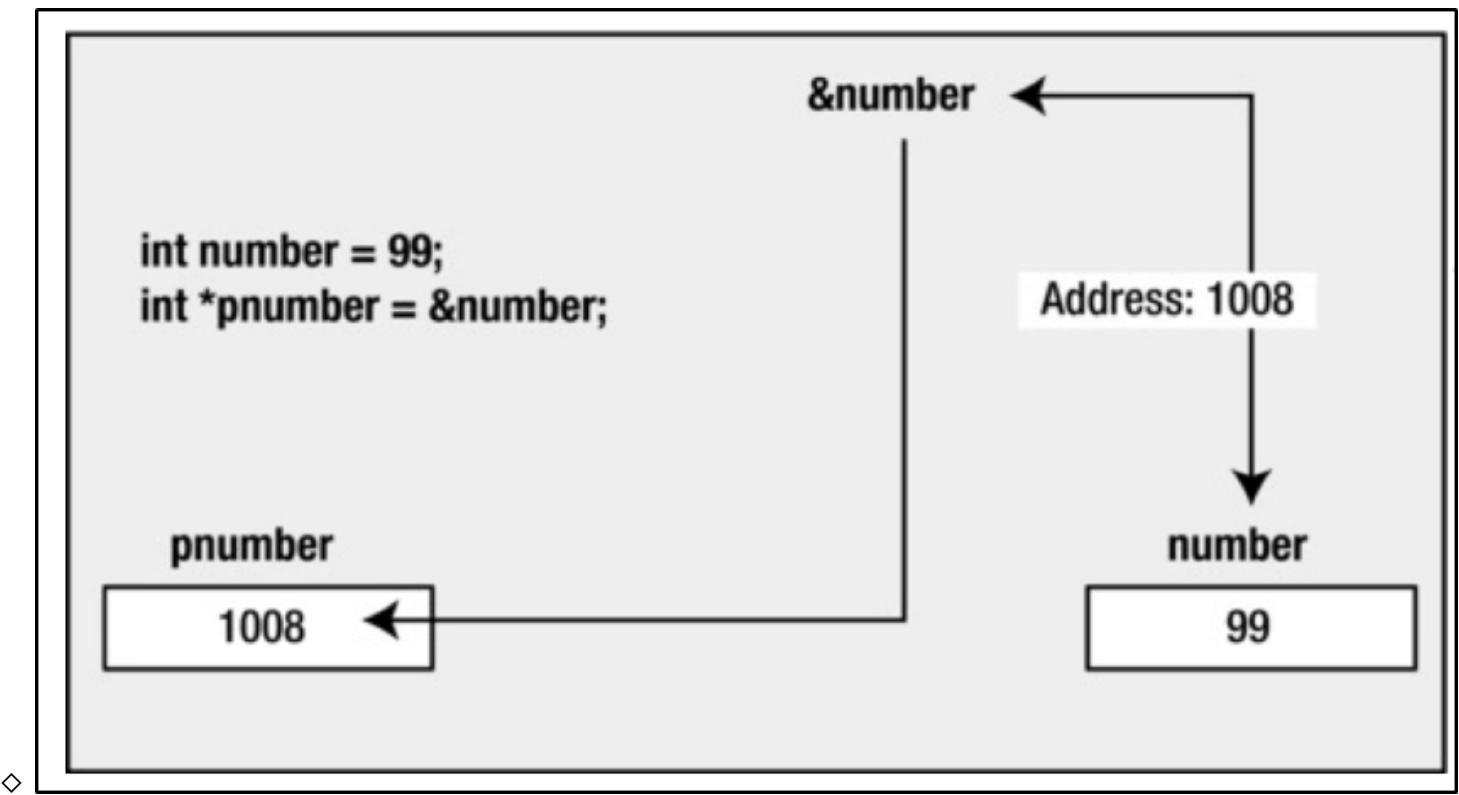
- This is the most important section of the course.
- It's also probably the most difficult section of the course:
 - ◊ the concept of pointers in C is a tricky concept.
- You have to understand pointers if you wanna take full advantage of the C language.

• Indirection:

- ◊ Pointers are very similar to the concept of indirection that you employ in your everyday life:
 - suppose you need to buy a new ink cartridge for your printer at the company that you work for.
 - all purchases are handled by the purchasing department:
 - you call Joe in purchasing and ask him to order the new cartridge for you.
 - Joe then calls the local supply store to order the cartridge
 - you are not ordering the cartridge directly from the supply store yourself (indirection).
- ◊ **In programming languages, indirection is the ability to reference something using a name, reference, or container, instead of the value itself:**
 - we use some form of indirection by using variable names, we don't use the memory address.
- ◊ The most common form of indirection is the act of manipulating a value through its memory address.
- ◊ A pointer provides an indirect means of accessing the value of a particular data item:
 - **a variable whose value is a memory address.**
 - the memory address is gonna be represented as an integer, a hexadecimal number.
 - **its value is the address of another location in memory that can contain a value.**
- ◊ It's either gonna be the value of an address on another location of memory (if it's an existing variable).
 - or
- ◊ A new location that you dynamically allocate.

• Overview:

- ◊ Just as there are reasons why it makes sense to go through the purchasing department to order new cartridges (you don't have to know which particular store the cartridges are being ordered from):
 - there are good reasons why it makes sense to use pointers in C.
- ◊ Using pointers in your program is one of the most powerful tools available in the C language.
- ◊ Pointers are also one of the most confusing topics of the C language:
 - it is important you get this concept figured out in the beginning and maintain a clear idea of what is happening as you dig deeper.
- ◊ The compiler must know the type of data stored in the variable to which it points:
 - it needs to know how much memory is occupied or how to handle the contents of the memory to which it points.
 - every pointer will be associated with a specific variable type.
 - it can be used only to point to variables of that type.
- ◊ Pointers of type "**pointer to int**" can point only to variables of type int.



- ◊ the value of **&number** is the address where **number** is located:
 - this value is used to initialize **pnumber** in the second statement.
- ◊ Pointers can only store addresses.

- **Why Use Pointers:**

- ◊ Accessing data by means of only variables is very limiting:
 - with pointers, you can access any location (you can treat any position of memory as a variable for example) and perform arithmetic with pointers.
- ◊ Pointers in C make it easier to use arrays and strings.
- ◊ Pointers allow you to refer to the same space in memory from multiple locations:
 - means that you can update memory in one location and the change can be seen from another location in your program.
- **you can also save memory space by being able to share components in your data structures.**
- ◊ Pointers allow functions to modify data passed to them as variables:
 - **pass by reference** - passing arguments to a function in way they can be changed by the function.
- ◊ **Can also be used to optimize a program to run faster or use less memory than it would otherwise.**
 - this is very important in embedded real-time programs.
- ◊ Pointers allow us to get multiple values from the function:
 - a function can return only one value but by passing arguments as pointers we can get more than one value from the pointer.
 - by using pointers we can return multiple values from that function.
 - this is a big advantage.
- ◊ With pointers **dynamic memory** can be created according to the program use:
 - we can save memory from static (compile time) declarations.
 - right now when we create memory for a program it's done implicitly by the compiler when you declare a variable:
 - when you declare an **int variable** it automatically create (allocates) **4 bytes** of memory for that variable.
 - when you declare the size for an array, that size is allocated at compile-time:
 - ⇒ so if you're trying to allocate more memory than that, your program is not gonna work.
- ◊ Pointers allow you to create memory at run-time:
 - **dynamic memory allocation.**
- ◊ Pointers allow us to design and develop complex data structures like:

- **stack**
- **queue**
- **linked list**

◊ Pointers provide direct memory access:

- efficient
- fast

90. Defining Pointers

- The syntax for declaring a pointer and then using the **addressof** operator to assign a memory address to a pointer.

- **Declaring pointers:**

- ◊ Pointers are not declared like normal variables:

- `pointer ptr; // NO THE WAY TO DECLARE A POINTER`

- ◊ It is not enough to say that a variable is a pointer:

- you also have to specify the kind of variable which the pointer points.
 - different variable types take up different amounts of storage.
 - some pointer operations require knowledge of that storage size.

- ◊ You declare a pointer to a variable of type **int** with:

- `int *pnumber;`

- the type of the variable with the name **pnumber** is **int***

- it can store the address of any variable of type int.

- `int * pi; // pi IS A POINTER TO AN INTEGER VARIABLE`
 - `char * pc; // pc IS A POINTER TO A CHARACTER VARIABLE`
 - `float * pf, * pg; // pf, pg ARE POINTERS TO FLOAT VARIABLES.`

- ◊ **The space between the * and the pointer name is optional:**

- programmers use the space in a declaration and omit it when dereferencing a variable.**

- ◊ The value of a pointer is an address, and it is represented internally as an **unsigned integer** on most systems:

- however, you shouldn't think of a pointer as an integer type.
 - there are things you can do with integers that you can not do with pointers, and vice versa.
 - you can multiply one integer by another, but you can not multiply one pointer by another.

- ◊ A pointer really is a new type, not an integer type:

- %p represents the format specifier for pointers.**

- ◊ The previous declarations create the variable but does not initialize it (there's no assignment there):

- it's very dangerous when not initialized.

- you should always initialize a pointer when you declare it:
 - if you don't know what to initialize it to, use a **NULL pointer**.

- **NULL pointers:**

- ◊ You can initialize a pointer so that it does not point to anything:

- ```
int *pnumber = NULL;
```

- **NULL** is a constant that is defined in the standard library.

- **is the equivalent of zero for a pointer.**

- ⇒ it means that it doesn't point to anything.

- ◊ **NULL** is a value that is guaranteed not to point to any location in memory:

- means that it implicitly prevents the accidental overwriting of memory by using a pointer that does not point to anything specific.

- this will prevent **buffer overflows**.

- ◊ By initializing it to **NULL**, it limits the mistakes that you can make.

- ◊ If you wanna use the **NULL** value, you need to add an **#include** directive for **stddef.h** to your source file:

- ```
#include <stddef.h>
```

- **Address of Operator:**

- ◊ If you want to initialize your variable with the address of a variable you have already declared:

- use the **address of operator**, **&**

- ```
int number = 99;
int *pnumber = &number;
```

- ◊ The initial value of **pnumber** is the address of the variable **number**:

- the declaration of **number** must precede the declaration of the pointer that stores its address.

- the compiler must have already allocated space and thus an address for **number** to use it to initialize **pnumber**.

- **Be Careful:**

- ◊ There is nothing special about the declaration of a pointer:
  - you can declare regular variables and pointers in the same statement.
- ◊ You could say:
  - `double value, *pVal, fnum;`
    - only the second variable, **pVal**, is a pointer.
- ◊ `int *p, q;`
  - the above declares a pointer, **p** of type **int\*** and a variable, **q**, that is of type **int**:
    - a common mistake is to think that both **p** and **q** are pointers.
- ◊ **Also, it is a good idea to use names beginning with p as pointer names.**
  - this will tell you that that variable is a pointer by just looking at it.

# 91. Accessing Pointers

- In this lecture we're gonna talk about accessing the value that a pointer is pointing to.

- **Accessing Pointer Values:**

- ◊ You use the indirection operator, `*`, to access the value of the variable pointed to by a pointer:

- also referred to as the deference operator because you use it to "**dereference**" a pointer.

- ◊ Example:

- ```
int number = 15;
int *pointer = &number;
int result = 0;
```

- the **pointer** variable contains the address of the variable **number**.

- you can use this in an expression to calculate a new value for result:

- ```
⇒ result = *pointer + 5;
```

- ◊ In the previous example the expression `*pointer` will evaluate the value stored at the address contained in the pointer:

- the value stored in **number**, **15**, so **result** will be set to **15 + 5**, which is **20**.

- ◊ The **indirection operator**, `*`, is also the symbol for multiplication, and **it is used to specify pointer types**:

- depending on where the asterisk appears, the compiler will understand whether it should interpret it as an **indirection operator**, as a **multiplication sign** or as a **part of a type specification**.

- **context determines what it means in any instance.**

- ◊ Code example:

- ```
int main(void){
    int count = 10, x;
    int *int_pointer;

    int_pointer = &count;
    x = *int_pointer;

    printf("count = %i, x = %i\n", count, x);

    return 0;
}
```

- **Displaying a pointers value:**

- ◊ To output the address of a variable, you use the output format specifier:
 - **%p**
 - it outputs a pointer value as a memory address in hexadecimal form.

- ◊ Code example:

```
int number = 0; // A VARIABLE OF TYPE int INITIALIZED TO 0
int *pnumber = NULL; // A POINTER THAT CAN POINT TO TYPE int

number = 10;
pnumber = &number;
printf("pnumber's value: %p\n", pnumber);           // OUTPUT THE VALUE (AN ADDRESS)
```

- ◊ Pointers occupy **8 bytes** in memory and the addresses have 16 hexadecimal digits:
 - if a machine has a **64-bit** operating system and my compiler supports **64-bit** addresses, the addresses will have **16 hexadecimal digits**.
 - some compilers only support **32-bit** addressing, in which case the addresses will be **32-bit** addresses, the addresses will have **8 hexadecimal digits**.

- **Displaying an Address:**

- ◊ Code example:

```
...
printf("number's address: %p\n", &number);           // OUTPUT THE ADDRESS
printf("pnumber's address: %p\n", (void*)&pnumber); // OUTPUT THE ADDRESS
```

- we use **(void*)** to get around a warning.

- ◊ Remember, a pointer itself has an address, just like any other variable:
 - you use **%p** as the conversion specifier to display an address.
 - you use the **&** (address of) operator to reference the address that the **pnumber** variable occupies.
 - the cast to **void*** is to prevent a possible warning from the compiler:
 - the **%p** specification expects the value to be some kind of **pointer** type, but the type of **&number** is "**pointer to pointer to int**"

- **Displaying the number of bytes a pointer is using:**

- ◊ You use the **sizeof** operator to obtain the number of bytes a pointer occupies:
 - on the instructor's machine this shows that a pointer occupies **8 bytes**.
 - a memory address on his machine is **64 bits**.

- ◊ You may get a compiler warning when using **sizeof** this way:

- **size_t** is an implementation-defined integer type.
 - to prevent the warning, you could cast the argument to type **int** like this:

- `printf("pnumber's size: %d bytes\n", (int)sizeof(pnumber)); // OUTPUT THE SI`

- ◊ Code example:

```
int main(void){  
    int number = 0;      // A VARIABLE OF TYPE int INITIALIZED TO 0  
    int *pnumber = NULL; // A POINTER THAT CAN POINT TO TYPE int  
  
    number = 10;  
    printf("number's address: %p\n", &number); // OUTPUT THE ADDRESS  
    printf("number's value: %d\n\n", number); // OUTPUT THE VALUE  
  
    pnumber = &number; // STORE THE ADDRESS OF number on pnumber  
  
    printf("pnumber's address: %p\n", (void*)&pnumber); // OUTPUT THE ADDRESS  
    printf("pnumber's size: %zd bytes\n", sizeof(pnumber)); // OUTPUT THE SIZE  
    printf("pnumber's value: %p\n", pnumber); // OUTPUT THE VALUE (AN ADDRESS)  
    printf("value pointed to: %d\n", *pnumber); // VALUE AT THE ADDRESS  
  
    return 0;  
}
```

- you use **%p** whenever you're displaying addresses
 - you use **%d** when displaying values of the integer pointer or the number of bytes.

92. (Challenge) Pointer Basics

- This is gonna be a challenge that gets you acquainted with the pointer syntax.
- In this challenge you are going to learn how to **create**, **initialize**, **assign** and **access** a pointer.

- **Requirements:**

- Write a program that creates an **integer** variable with a hard-coded value.
- Assign** that **variable's address** to a **pointer variable**:
 - using the address of **&** operator.
- Creating the pointer is just declaring it.
- Initializing it will be to **NULL** because it's not gonna point to anything.
- Assigning it will be assigning the value of another variable:
 - use the **dereference** ***** operator.
- Display as output:
 - address of the pointer**
 - value of the pointer**
 - value of what the pointer is pointing to.**

My Code

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 13 - POINTER BASICS
DATE: MAY 1ST, 2022
*/



#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

void main(void){
    int number = 7;
    int *pnumber = NULL;

    pnumber = &number; // ASSIGN number'S ADDRESS TO pnumber

    printf("\nThe pointer's address is: %p", (void*)&pnumber); // OUTPUT
pnumber'S ADDRESS
    printf("\nThe pointer's value is: %p", pnumber);
    printf("\nThe value of what the pointer is pointing to is: %d\n",
*pnumber);

    return;
}
```

93. (Demonstration) Pointer Basics

- All good.

94. Using Pointers

- We're gonna talk about the use cases for pointers.

- **Overview:**

- ◊ C offers several basic operations you can perform on pointers.
- ◊ You can assign an address to a pointer:
 - the assigned value can be an **array name**, a variable preceded by the **address of operator (&)**, or **another second pointer**.
- ◊ You can **dereference** a pointer:
 - the `*` operator gives the value stored in the pointed-to location.
- ◊ You can take a pointer address:
 - the `&` operator tells you where the pointer itself is stored.
- ◊ You can perform **pointer arithmetic**:
 - use the `+` operator to **add an integer to a pointer** or a **pointer to an integer (integer is multiplied by the number of bytes in the pointed-to type and added to the original address)**.
 - this allows you to jump around memory quicker, specially if you're using arrays.
 - **increment a pointer by one** (useful in arrays when moving to the next element).
 - use the `-` operator to **subtract an integer from a pointer (integer is multiplied by the number of bytes in the pointed-to type and subtracted from the original address)**.
 - this allows you to jump around memory quicker, specially if you're using arrays.
 - **decrement a pointer by one (useful in arrays when going back to the previous element)**.
- ◊ You can find the **difference between two pointers**:
 - you do this for two pointers to elements that are in the same array to find out how far apart the elements are.
- ◊ You can use the **relational operators** to **compare the values of two pointers**:
 - pointers must be the same type.
- ◊ **Remember, there are two forms of subtraction**:
 - you can **subtract one pointer from another to get an integer**.
 - you can **subtract an integer from a pointer to get a pointer**.

◊ Be careful when incrementing or decrementing pointers and causing an array "out of bounds" or "under bounds" error:

- the computer does not keep track of whether a pointer still points to an array element.
- most of the stuff that you do with pointer arithmetic is when you're operating on arrays:
 - because it allows you to move quickly.

• Pointers used in expressions:

- ◊ The value referenced by a pointer can be used in an arithmetic expression:
 - if a variable is designed to be of type "**pointer to integer**" then it is evaluated using the rules of integer arithmetic.

- code example:

```
int number = 0; // A VARIABLE OF TYPE int INITIALIZED TO 0
int *pNumber = NULL; // A POINTER THAT CAN POINT TO TYPE int

number = 10;
pNumber = &number; // STORE THE ADDRESS OF number IN pNumber

*pNumber += 25; // THIS IS NOT CHANGING WHAT IS POINTING TO, WE'RE
                 // CHANGING THE VALUE THAT IT'S POINTING TO
```

→ increments the value of the **number** variable by **25**.

⇒ * indicates you are accessing the contents to which the variable called **pnumber** is pointing to.

◊ If a pointer points to a variable **x**:

- that pointer has been defined to be a pointer to the same data type as is **x**.
- use of ***pointer** in an expression is identical to the use of **x** in the same expression.

◊ A variable defined as a "**pointer to int**" can store the address of any variable of type **int**:

- code example:

```
int value = 999;
pNumber = &value;
*pNumber += 25;
```

→ the statement will operate with the new variable, **value**:

⇒ the new contents of **value** will be **1024**.

• this is the concept of changing the variable in one spot (the pointer) and you're changing it in multiple variables.

◊ A pointer can contain the address of any variable of the appropriate type:

- you can use one pointer variable to change the values of many different variables.
- as long as they are of a type compatible with the pointer type.

◊ Code example:

```
void main(void){  
    long num1 = 0L;  
    long num2 = 0L;  
    long *pNum = NULL;  
  
    pNum = &num1;          // GET ADDRESS OF num1  
    *pNum = 2L;           // SET num1 to 2  
    ++num2;              // INCREMENT num2  
    num2 += *pNum;        // ADD num1 TO num2  
  
    pNum = &num2;          //GET ADDRESS OF num2  
    ++*pNum;             // INCREMENT num2 DIRECTLY  
  
    printf("num1 = %ld\n num2 = %ld\n *pNum = %ld\n *pnum + num2 = %ld\n",  
        num1, num2, *pNum, *pNum + num2);  
  
    return;  
}
```

- in this **main()** we're doing some pointer arithmetic.

• When receiving input:

- ◊ How to use pointers in this way.
- ◊ When we have used **scanf()** to input values, we have used the **&** operator to obtain the address of a variable:
 - on the variable that is to store the input (second argument).
 - **scanf()**'s second argument is a **pointer**.
- ◊ When you have a pointer that already contains an address, you can use the pointer name as an argument for **scanf()**.
- code example:

```
int value = 0;  
int *pValue = &value;      // SET POINTER TO REFER TO VALUE  
  
printf("Input an integer: ");  
scanf("%d", pValue);     // READ INTO value VIA THE POINTER  
  
printf("You entered %d.\n", value)           // OUTPUT THE VALUE ENTERED
```

• Testing for NULL:

- ◊ There is one rule you should burn into your memory:
 - do no dereference an uninitialized pointer (a pointer that's pointing to nothing or a pointer that doesn't have memory allocated for it).

- code example:

```
int * pt; // AN UNINITIALIZED POINTER
→ *pt = 5; // A TERRIBLE ERROR
```

⇒ the second line means that you wanna store the value **5** in the location to which **pt** points:

- **pt** has a **random value**, there is no knowing where the **5** will be placed.

◊ This might go somewhere harmless, it might overwrite data or code, or it might cause the program to crash.

◊ Creating a pointer only allocates memory to store the pointer itself:

- it does not allocate memory to store data.
- before you use a pointer, it should be assigned a memory location that has already been allocated:
 - assign the address of an existing variable to the pointer.
 - or you can use the **malloc()** (dynamic memory allocation) function to allocate memory first.

◊ We already know that when declaring a pointer that does not point to anything, we should initialize it to **NULL**.

- code example:

```
- int *pValue = NULL;
```

◊ **NULL** is a special symbol in C that represents the pointer equivalent to 0 with ordinary numbers:

- the code below also sets a pointer to **NULL** using 0:

```
- int *pValue = 0;
```

◊ Because **NULL** is the equivalent of zero, if you want to test whether **pValue** is **NULL**, you can do this:

```
▪ if (!pValue) ...
```

- or you can do it explicitly by using **== NULL**

◊ You want to check for **NULL** before you dereference a pointer:

- you often do this when pointers are passed to functions.
 - because functions will take pointers as parameters.

→ and if the caller is passing in bad data or passing in a pointer that doesn't have any memory allocated, the program could crash.

95. Pointers and const

- How to use the **const** modifier with pointers.

- **Overview:**

- ◊ When we use the **const** modifier on a variable or an array it tells the compiler that the contents of the variable/array will not be changed by the program.

- ◊ With pointers, we have to consider **two things** when using the **const** modifier:

- whether the **pointer will be changed**:

- we may not wanna change the address ever.

- whether the **value that the pointer points to will be changed**:

- we may not wanna change the value that the pointer is pointing to.

- ◊ **In either of the previous cases we have to be specific with our syntax.**

- ◊ You can use the **const** keyword when you declare a pointer to indicate that the **value pointed to must not be changed**:

- code example:

- ```
long value = 999L;
const long *pValue = &value; // DEFINES A POINTER TO A CONSTANT
```

- you have declared the value pointed to by **pValue** to be **const**:

- ⇒ the compiler will check for any statements that attempt to modify the value pointed by **pValue** and flag such statements as errors.

- **the const keyword goes before the data type**

- ◊ Using the previous example:

- the following statement will now result in an error message from the compiler:

- ```
*pValue = 8888L;          // ERROR - ATTEMPT TO CHANGE const LOCATION
```

- **Pointers to Constants:**

- ◊ Using the previous example you can still modify **value** (you have only applied **const** to the pointer).

- ◊ code example:

- ```
value = 7777L;
```

- ◊ The value pointed to has changed, but you did not use the pointer to make the change.
- ◊ The pointer itself is not constant, so you can still change what it points to:
  - code example:

```
long number = 8888L;
pValue = &number; // OK - CHANGING THE ADDRESS IN pValue
```

→ **we're changing the address of the value of the pointer, we're not changing the value of what the pointer was pointing to.**

- ◊ If you don't wanna change what the pointer's pointing to:
  - **const** has to be that first keyword in the declaration of the pointer itself.

- ◊ **Using this example you will change the address stored in *pValue* to point to *number*:**

- **you still cannot use the pointer to change the value that is stored.**
- **you can change the address stored in the pointer as much as you like.**
- **using the pointer to change the value pointed to is not allowed, even after you have changed the address stored in the pointer.**

## • Constant Pointers:

- ◊ You might also want to ensure that the address stored in a pointer cannot be changed.
- ◊ You can do this by using the **const** keyword in the declaration of the pointer.

- ◊ Code example:

```
int count = 43;
int *const pCount = &count;
```

- **you put the **const** keyword after the **data type** and the asterisk (\*)**
- **this is saying that the pointer itself cannot change its value.**
- **if you put it before the **int** you're saying the value pointed to by the pointer cannot change.**

- ◊ **The above ensures that a pointer always points to the same thing:**
  - **indicates that the address stored must not be changed.**
  - **the compiler will check that you do not inadvertently attempt to change what the pointer points to elsewhere in your code.**

◊ Code example:

- ```
int item = 34;
pCount = &item; // ERROR - ATTEMPT TO CHANGE A CONSTANT POINTER
```

◊ **It is all about where you place the *const* keyword, either *before* the type or *after* the type:**

- ```
const int * ... // VALUE POINTED TO CANNOT BE CHANGED
int *const ... // POINTER ADDRESS CANNOT CHANGE
```

◊ You can still change the value that **pCount** points to using **pCount**:

- ```
*pCount = 354; // OK - CHANGES THE VALUE OF COUNT
```

◊ References the value stored in **count** through the **pointer** and changes its value to **354**.

◊ You can create a constant pointer that points to a value that is also constant:

- code example:

```
int item = 25;
const int *const pItem = &item;
```

→ the **pItem** is a constant pointer to a constant so everything is fixed:

⇒ cannot change the address stored in **pItem**

⇒ cannot use **pItem** to modify what it points to.

◊ You can still change the value of **item** directly:

- if you wanted to make everything not change, you could specify **item** as **const** as well.

96. void pointers

- Overview:

- ◊ The type name **void** means:
 - **absence of any type**
- ◊ A pointer of type **void*** can contain the address of a **data item of any type**:
 - so it has a lot of flexibility.
- ◊ If you don't know what you're gonna store as a pointer you can use **void*** :
 - it's usually used in functions for function parameters.
- ◊ **void*** is often used as a **parameter type** or **return value type** with functions that deal with data in a type-independent way:
 - **you have to cast it when you dereference it.**
- ◊ Any kind of pointer can be passed around as a value of type **void***:
 - the **void pointer** does not know what type of object it is pointing to, so, it cannot be dereferenced directly.
 - the **void pointer** must first be explicitly cast to another pointer type before it is dereferenced.
- ◊ The address of a variable of type **int** can be stored in a pointer variable of type **void*** :
 - when you want to access the **integer value** at the address stored in the **void*** pointer, you **must first cast the pointer to type int***

- ◊ Code example:

```
int i = 10;
float f = 2.34;
char ch = 'k';

void *vptr;

vptr = &i;
printf("Value of i = %d\n", *(int *)vptr);

vptr = &f;
printf("Value of f = %.2f\n", *(float *)vptr);

vptr = &ch;
printf("Value of ch = %c\n", *(char *)vptr);
```

- the advantage here is that we only have to create one variable type and we can store multiple different addresses in it.

- but whenever you wanna use it you have to cast the pointer:
 - ⇒ **(data_type *)**

- and the dereference it:
 - ◊ **`*(data_type *)pointer_name`**

97. Pointers and Arrays

- We're gonna talk about the relationships between pointers and arrays.
- We need to understand this relationship before understanding pointer arithmetic.

- **Overview:**

- ◊ An array is a collection of objects of the same type that you can refer to using a single name.
- ◊ A pointer is a variable that has as its value a memory address that can reference another variable or constant of a given type:
 - you can use a pointer to hold the address of different variables at different times (must be of the same type).
- ◊ Arrays and pointers seem quite different, but, they are very closely related and can sometimes be used interchangeably.
- ◊ One of the most common uses of pointers in C is as pointers to arrays:
 - a **character pointer** is the same as a character array.
- ◊ The **main reason** for using pointers to arrays are ones of **notational convenience** and **of program efficiency**:
 - you can use either one, an **array** or a **pointer** in many cases:
 - except when you're talking about dynamic memory management:
 - but, in most cases you can, the only difference is its syntax.
 - ◊ It's also more efficient to use character pointers.
 - ◊ Pointers to arrays generally results in code that uses less memory and executes faster.
 - **you wanna use a pointer for characters as much as possible.**
 - and a pointer to other data types as well.
 - because it is faster and more efficient (uses less memory).
 - ⇒ **use pointers over arrays in most parts.**

- **Arrays and Pointers Example:**

◊ If you have an array of 100 integers:

- code example:

- `int values[100];`

→ you can define a pointer called **valuesPtr**, which can be used to access the integers contained in this array:

⇒ `int *valuesPtr;`

◊ When you define a pointer that is used to point to the elements of an array, you do not designate the pointer as type "**pointer to array**" :

- **you designate the pointer as pointing to the type of element that is contained in the array.**

◊ To set **valuesPtr** to point to the first element in the **values** array, you write:

- code example:

- `valuesPtr = values;`

→ you don't have to put the address or anything because **values** array is a pointer (**underneath the hood all arrays are pointers**).

⇒ you have to get this previous line in your head.

◊ So if you want to assign the first element of an array to a pointer all you have to do is:

- specify the array on the right hand side of the equals (=) operator :

- and on the left hand side you just have the **pointer's name**.

- this is gonna point to the first address (the first element) in the array.

◊ The C **address of** operator is not used:

- the C compiler treats the appearance of an array name without a subscript as a pointer to the array.

- specifying values without a subscript has the effect of producing a pointer to the first element of values.

- you can then access subsequent elements in the array by just **incrementing** the pointer to go to the next address:

- this is just a notation and efficiency thing.

◊ An equivalent way of producing a pointer to the start of **values** is to apply the **address of operator** to the first element of the array:

- code example:

- `valuesPtr = &values[0];`

→ so you can do it two ways, **either way is fine**.

⇒ it's a notational thing.

- **Summary:**

- ◊ The two expressions:

- `ar[i]`
 - and
 - `*(ar+i)`

- ◊ are equivalent in meaning:

- **both work if `ar` is the name of an array, and both work if `ar` is a pointer variable.**

- **using an expression such as `ar++` only works if `ar` is a pointer variable:**

- you can't increment an array by saying `++` an expect it to go to the next element:
→ you can only do that on array pointers or pointers.

- ◊ **`ar[i]` is an array and an array is a pointer:**

- so you can **dereference** it by using:

- `*(ar+i)`

- `ar` is a pointer, so both are the same thing.

- ◊ Pointer arithmetic relates stronger to the relationship between **arrays** and **pointers**.

98. Pointer Arithmetic

- Now that we understand the concept between pointers and arrays, this concept is gonna be easier to understand.
- We say "**arithmetic**" but all it really meas is that you can either do:
 - + which is **incrementation**
 - or
 - which is **decrementation**
- ◇ on a pointer itself.

• Pointer Arithmetic:

- ◇ The real power of using pointers to arrays comes into play when you want to sequence through the elements of an array.
 - ◇ Because pointers use addresses and **each element in an array is just another address**
 - ◇ Code example:

```
*valuesPtr      /* CAN BE USED TO ACCESS THE FIRST INTEGER
                  /* OF THE values ARRAY, THAT IS, values[0]
```
 - ◇ To reference **values[3]** through the **valuesPtr** variable, you can **add 3** to **valuesPtr** and then apply the indirection operator:
 - code example:

```
* (valuesPtr + 3)
```

 - you put the * outside of the parentheses to make the **dereference** clear.
 - ◇ The expression, ***(valuesPtr + i)** can be used to access the value contained in **values[i]** :
 - to set **values[10]** to **27**, you could do the following:
 - code example:

```
values[10] = 27;
```

 - ⇒ or, using **valuesPtr**, you could:

```
* (valuesPtr + 10) = 27;
```
 - ⇒ **this second option is more efficient**
 - ◇ To set **valuesPtr** to point to the **second element** of the values array:
 - you can apply the **address of operator** to **values[1]** and assign the result to **valuesPtr**

- code example:

→ `valuesPtr = &values[1];`

⇒ **you can do this if you want, but it's much more convenient to do pointer arithmetic.**

- ◊ If **valuesPtr** points to **values[0]**:

- you can set it to point to **values[1]** by simply adding **1** to the value of **valuesPtr**.

- code example:

→ `valuesPtr += 1;`

⇒ this is a perfectly valid expression in C and can be used for pointers to any data type.

- ◊ The **increment** and **decrement** operators **++** and **--** are particularly useful when dealing with pointers.

- using the **increment** operator on a pointer has the **same effect as adding one to the pointer**.

- using the **decrement** operator on a pointer has the **same effect as subtracting one from the pointer**.

- ◊ Code example:

- `++valuesPtr;`

- the previous example sets **valuesPtr** pointing to the next integer in the values array (**values[1]**)

- `--valuesPtr;`

- sets **valuesPtr** pointing to the **previous integer in the values array**, assuming that **valuesPtr** was not pointing to the beginning of the values array.

→ validate when using the **increment** and **decrement** operators to avoid "**out of bounds**" errors.

- Code example:

```
/*
    AUTHOR: JFITECH
    PURPOSE: POINTER ARITHMETIC
    DATE: MAY 2ND, 2022
*/

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

// FUNCTION PROTOTYPES
int arraySum(int array[], const int n);
```

```

// main FUNCTION
void main(void){
    // VARIABLE DECLARATION
    int values[10] = {3, 7, -9, 3, 6, -1, 7, 9, 1, -5};

    printf("The sum is %i\n", arraySum(values, 10)); // THE SECOND ARGUMENT ON
THE FUNCTION IS THE ARRAY'S SIZE
}

// arraySum FUNCTION
int arraySum(int array[], const int n){
    // VARIABLE DECLARATION
    int sum = 0, *ptr;

    int * const arrayEnd = array + n; // int * const MEANS THAT POINTER
ADDRESS CANNOT BE CHANGED
    // WE DO THIS BECAUSE IT MAKES OUR
LOOP MORE EFFICIENT
    // arrayEnd IS POINTING TO THE ADDRESS
OF THE END OF THE ARRAY

    for (ptr = array; ptr < arrayEnd; ++ptr) // ptr = array ASSIGNS THE
POINTER TO POINT TO THE FIRST ELEMENT
        // IN THE ARRAY.
        // ptr AND arrayEnd ARE BOTH
        // ADDRESS OF ptr IS LESS THAN
ADDRESSES, SO WE'RE SAYING THAT IF THE
arrayEnd THEN WE WANNA EXIT THE LOOP
    sum += *ptr;

    return sum;
}

```

- the previous program sums all the elements in an array.

- ◊ To **pass an array to a function**, you **simply specify the name of the array**.
- ◊ To **produce a pointer to an array**, you need only to **specify the name of the array**

- ◊ The previous lines of text imply that in the call to the **arraySum()** function, what was passed to the function was actually a pointer to the array values:
 - this explains why you are able to change the elements of an array from within a function.
 - the only way that elements get changed outside of a function is if you're **passing pointers to a function**.
 - and because the elements are changed inside of a function this implies that you're passing a pointer.
 - ⇒ this explain that indeed arrays are pointers.

 - ◊ So, you might wonder why the formal parameter inside the function is not declared to be a pointer.
 - well, it could, we'd just have to do something like this:
 - `int arraySum(int *array, const int n)`
 - the above is perfectly valid:
 - ⇒ **pointers** and **arrays** are intimately related in C.

⇒ this is why you can declare the **array** to be of type:

- **array of ints**

- ◊ `int arraySum(int array[], const int n)`

- inside the **arraySum** function or to be of type:

- **pointer to int**

- ◊ `int arraySum(int *array, const int n)`

- ◊ If you are going to be using index numbers to reference the elements of an array that is passed to a function, **declare the corresponding formal parameter to be an array**:

- more correctly reflects the use of the array by the function.
 - (this is just notational convenience)

- ◊ If you are going to use the argument as a pointer to the array, declare it to be of type pointer in the formal parameter:

- example with pointer notation:

```
/*
    AUTHOR: JFITECH
    PURPOSE: POINTER ARITHMETIC
    DATE: MAY 2ND, 2022
*/

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

// FUNCTION PROTOTYPES
int arraySum(int *ptr, const int n);

// main FUNCTION
void main(void) {
    // VARIABLE DECLARATION
    int values[10] = {3, 7, -9, 3, 6, -1, 7, 9, 1, -5};

    printf("The sum is %i\n", arraySum(values, 10)); // THE SECOND ARGUMENT
    // THE FUNCTION IS THE ARRAY'S SIZE
}

// arraySum FUNCTION
int arraySum(int *ptr, const int n){
    // VARIABLE DECLARATION
    int sum = 0;

    int * const arrayEnd = ptr + n; // int * const MEANS THAT POINTER ADDRESS
    // CANNOT BE CHANGED
    // WE DO THIS BECAUSE IT MAKES OUR
    // LOOP MORE EFFICIENT
    // arrayEnd IS POINTING TO THE ADDRESS
    // OF THE END OF THE ARRAY

    for ( ; ptr < arrayEnd; ++ptr)
        sum += *ptr;

    return sum;
}
```

- the function becomes smaller this way and the program will run faster.

- **Summary:**

- ◊ Here are some things that are **valid** with pointers and some things that are **invalid**:
 - so if we first declare the following variables:

```
int
urn[3]; // AN ARRAY
int * ptr1, * ptr2; // TWO POINTERS
```

| Valid | Invalid |
|------------------|---------------------|
| ptr1++; | urn++; |
| ptr2 = ptr1 + 2; | ptr2 = ptr2 + ptr1; |
| ptr2 = urn + 1; | ptr2 = urn * ptr1; |

→

- ◊ Functions that process arrays actually use pointers as arguments.
- ◊ You have a choice between **array notation** and **pointer notation** for writing array-processing functions:
 - **using pointers is less code and more efficient.**
- ◊ Using **array notation** makes it more obvious that the function is working with arrays:
 - array notation has a more familiar look to programmers versed in **FORTRAN, Pascal, Modula-2, or BASIC.**
- ◊ Other programmers might be more accustomed to working with pointers and might find the pointer notation more natural:
 - it's closer to machine language and, with some compilers, leads to more efficient code.

99. Pointers and Arrays Example

- We're gonna go through some examples to demonstrate pointer arithmetic:

- **1st code example:**

```
/*
    AUTHOR: JULS
    PURPOSE: POINTER ARITHMETIC EXAMPLE
    DATE: MAY 2ND, 2022
*/

#include <stdio.h>
#include <string.h>

int main(void){
    char multiple[] = "a string";
    char *p = multiple;

    for(int i = 0; i < strlen(multiple); ++i)
        printf("multiple[%d] = %c *(p+%d) = %c &multiple[%d] = %p p+%d = %p\n",
i, multiple[i], i, *(p+i), i, &multiple[i], i, p+i);

    return 0;
}
```

```
$ ./pointer_arithmetic_example1.exe
multiple[0] = a *(p+0) = a &multiple[0] = 0xfffffc07 p+0 = 0xfffffc07
multiple[1] = * (p+1) = &multiple[1] = 0xfffffc08 p+1 = 0xfffffc08
multiple[2] = s *(p+2) = s &multiple[2] = 0xfffffc09 p+2 = 0xfffffc09
multiple[3] = t *(p+3) = t &multiple[3] = 0xfffffc0a p+3 = 0xfffffc0a
multiple[4] = r *(p+4) = r &multiple[4] = 0xfffffc0b p+4 = 0xfffffc0b
multiple[5] = i *(p+5) = i &multiple[5] = 0xfffffc0c p+5 = 0xfffffc0c
multiple[6] = n *(p+6) = n &multiple[6] = 0xfffffc0d p+6 = 0xfffffc0d
multiple[7] = g *(p+7) = g &multiple[7] = 0xfffffc0e p+7 = 0xfffffc0e

Type char occupies: 1 bytes
```

- **2nd code example concerning pointer arithmetic with *longs*:**

```
/*
    AUTHOR: JULS
    PURPOSE: POINTER ARITHMETIC EXAMPLE
    DATE: MAY 2ND, 2022
*/

#include <stdio.h>
```

```

int main(void) {
    long multiple[] = {15L, 25L, 35L, 45L};
    long *p = multiple;

    for(int i = 0; i < sizeof(multiple)/sizeof(multiple[0]); ++i)
        printf("address p+%d (&multiple[%d]): %llu      *(p+%d) value: %d\n",
i, i, (unsigned long long)(p + i), i, *(p+i));

    printf("\n  Type long occupies: %d bytes\n", (int)sizeof(long));
}

return 0;
}

```

- if we have a long pointer and we add 1 to it, we're actually adding 4 bytes to it, because a long is a little bit bigger.
 - we have to divide the **sizeof(multiple)** by each spot in the array because these are addresses that are incrementing by **4 bytes** each.
 - we can see that when executing the program the address sizes' increment by **8 bytes** (in our system, remember this is system dependent).

```

$ ./pointer_arithmetic_example2.exe
address p+0 (&multiple[0]): 4294953984  *(p+0) value: 15
address p+1 (&multiple[1]): 4294953992  *(p+1) value: 25
address p+2 (&multiple[2]): 4294954000  *(p+2) value: 35
address p+3 (&multiple[3]): 4294954008  *(p+3) value: 45

Type long occupies: 8 bytes

```

100. Pointers and Strings

- We're gonna talk about character pointers, mainly how you can use character pointers to represent strings.

- **Overview:**

- ◊ We now know how arrays relate to pointers and the concept of pointer arithmetic.

- ◊ These concepts can be very useful when applied to character arrays (strings).
 - because we can now easily traverse a string character by character.

- ◊ **One of the most common applications of using a pointer to an array is as a pointer to a character string:**

- the reasons are one of notational convenience and efficiency.
 - using a variable of type **pointer to char** to reference a string gives you a lot of flexibility.
 - so you're gonna try to use character pointers as much as you can over character arrays:
 - specifically using character pointers to pass data to a function, and all sort of things like that.

- ◊ Lets look at an example that uses an array to copy a string:
 - we're gonna compare it to a function that takes a character pointer:

```
// CHARACTER ARRAY FUNCTION
void copyString(char to[], char from[]) {
    int i;

    for(i = 0; from[i] != '\0'; ++i)
        to[i] = from[i]

    to[i] = '\0'
}

// CHARACTER POINTER FUNCTION
void copyString(char *to, char *from) {
    for(; *from != '\0'; ++from, ++to)
        *to = *from;
    *to = '\0';
}
```

- we can see that the character pointer function is more efficient.

- **Char Arrays as Pointers:**

- ◊ If you have an array of characters called `text`, you could similarly define a pointer to be used to point to elements in `text`.
 - code example:

```
- char *textPtr;
```

→ if **textPtr** is set pointing to the beginning of an **array of chars** called **text** then you can just increment it by one:

```
⇒ ++textPtr;
```

- the above sets **textPtr** pointing to the next character in **text**, which is **text[1]**.

```
◊ --textPtr;
```

- the above sets **textPtr** pointing to the previous character in **text**, assuming that **textPtr** was not pointing to the beginning of **text** prior to the execution of this statement.

◊ Let's look at the previous example **optimized**:

```
int main(void){  
    char string1[] = "A string to be copied.";  
    char string2[50];  
  
    copyString(string2, string1);  
    printf("%s\n", string2);  
  
}  
  
void copyString(char *to, char *from) {  
    while(*from) // THE null CHARACTER IS EQUAL TO THE VALUE 0, SO IT'  
    JUMP OUT THEN  
        *to++ = *from++;  
}
```

- if we were to use pre-increment operators we'd have a problem because we'd miss the first character in the string.

101. (Challenge) Counting characters in a String

- In this challenge we're going to learn all about pointer arithmetic.
- In this challenge you are going to write a program that tests your understanding of **pointer arithmetic** and the **const** modifier.

- **Requirements:**

- Write a function that calculates the length of a string:
 - the function should take as a **parameter** a **const char pointer**:
 - ⇒ this way you cannot change the value of what the pointer is pointing to.
 - the function can only determine the **length of the string using pointer arithmetic**
 - use the incrementation operator (**+ +pointer**) to get to the end of the string.
 - you are required to use a **while loop** using the value of the pointer to exit.
 - while(dereferenced pointer)**
 - the function should **subtract two pointers (one pointing to the end of the string and one pointing to the beginning of the string)**.
 - the function should **return** an **int** that is the **length of the string that was passed into the function**.
 - subtract the two pointers (you are actually subtracting two addresses to know how far both elements are in memory)

My Code

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 14 - COUNTING CHARACTERS IN A STRING
DATE: MAY 2ND, 2022
*/

#include <stddef.h>
#include <stdio.h>

// FUNCTION PROTOTYPES
int strnLen(const char*);

// MAIN FUNCTION
void main(void){
    char string[] = "Hallo, jeg snakker litt norsk!";
    printf("\nThe length of the string is: %d", strnLen(string));
    return;
}

// strnLen FUNCTION
int strnLen(const char *strnArray){
    // DECLARE VARIABLES
    int length;
    char * const arrayEnd = strnArray + n;

    while(*strnArray < *arrayEnd)
        *strnArray++;

    length = arrayEnd - strnArray;
    return length;
}
```

- Correct solution after watching the demonstration and fixing my code:

```
/*
AUTHOR: JULS
PURPOSE: CHALLENGE 14 - COUNTING CHARACTERS IN A STRING
DATE: MAY 2ND, 2022
*/

#include <stddef.h>
#include <stdio.h>

// FUNCTION PROTOTYPES
int strnLen(const char *strnArray);
```



```
// MAIN FUNCTION
void main(void){
    char string[] = "Hallo, jeg snakker litt norsk!";

    printf("\nThe length of the string is: %d", strnLen(string));

    return;
}

// strnLen FUNCTION
int strnLen(const char *strnArray) {
    // DECLARE VARIABLES
    int length;
    const char *arrayEnd = strnArray;

    while(*arrayEnd)
        arrayEnd++;

    return arrayEnd - strnArray;
}
```

102. (Demonstration) Counting characters in a string

•

Instructor's Code:

```
#include <stdio.h>
#include <stdlib.h>

int stringLength(const char *string);

int main()
{
    printf("%d \n", stringLength("stringLength test"));
    printf("%d \n", stringLength(""));
    printf("%d \n", stringLength("jason"));

    return 0;
}

int stringLength(const char *string)
{
    const char *lastAddress = string;

    while (*lastAddress)
        ++lastAddress;

    return lastAddress - string;
}
```

103. Pass by reference

- A concept called pass by reference.
- One of the advantages of using pointers is that you don't have to use global variables as much:
 - ◊ because when you pass a pointer to a function, the function can use that parameter and when it modifies it, the data will be modified outside the function.
 - this is a very valuable concept.
- This is more of a concept, C doesn't really employ this.
 - ◊ we're gonna see how this works and some example.

• Pass by Value:

- ◊ There are a few different ways you can pass data to a function:
 - **pass by value**
 - this has to do with functions
 - whenever we call a function besides the **scanf()** function, every other time we called a function when you modify that data inside the function, it didn't get modified outside of the function.
 - C does it exclusively by passing by value.
 - **pass by reference**
 - C simulates pass by reference because you copy addresses.
 - ◊ **Pass by value** is when a function copies the actual value of an argument into the formal parameter of the function:
 - changes made to the parameter inside the function have no effect on the argument.
 - the argument being passed in does not get modified.
 - it acts like a local variable because it's a copy.
 - under the hood it's being copied and passed to the function.
 - ◊ **C programming uses call by value to pass arguments:**
 - call by value and pass by value are used interchangeably.
 - it means the code within a function cannot alter the arguments used to call the function.
 - there are no changes in the values, though they had been changed inside the function.
 - ◊ Code example, pass by value:

```
/* function definition to swap the values*/  
  
void swap(int x, int y) {  
    int temp;
```

```

temp = x; // SAVE THE VALUE OF x
x = y;     //PUT y INTO x

return;
}

```

- this swap routine is often used to demonstrate **pass by value**.
- **x** and **y** are integers and they're not addresses, that's significant.

◊ Code example continued:

```

int main() {
    // LOCAL VARIABLE DEFINITION
    int a = 100;
    int b = 200;

    printf("Before swap, value of a: %d\n", a);
    printf("Before swap, value of b: %d\n", b);

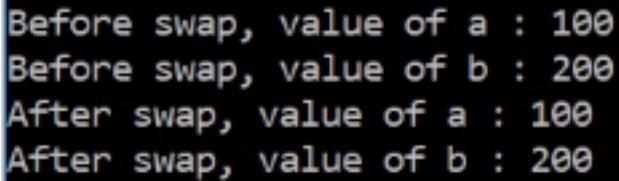
    // CALLING A FUNCTION TO SWAP THE VALUES
    swap(a,b);

    printf("After swap, value of a: %d\n", a);
    printf("After swap, value of b: %d\n", b);

    return 0;
}

```

- the variables **a** and **b** never change because we're doing **pass by value**:



```

Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 100
After swap, value of b : 200
→

```

• **Passing data using copies of pointers:**

- ◊ Pointers and functions get along quite well together:
 - you can pass a pointer as an argument to a function and you can also have a function return a pointer as its result.
- ◊ Pass by reference copies the address of an argument into the formal parameter:
 - the address is used to access the actual argument used in the call.
 - it means the changes made to the parameter inside the function affect the passed argument.
 - even though under the hood it's not technically **pass by reference** it mimics it.
- ◊ To pass a value by reference, argument pointers are passed to the functions just like any other value:

- you need to declare the function parameters as pointer types.
- changes inside the functions are reflected outside the function as well.
- unlike call by value where the changes do not reflect outside the function.
 - we're still passing by value but it's just a value of a copy of the address.

◊ Code example using pointers to pass data:

```
// FUNCTION DEFINITION TO SWAP THE VALUES
void swap(int *x, int *y) {
    int temp;
    temp = *x; // SAVE THE VALUE AT ADDRESS x
    *x = *y // PUT y INTO x
    *y = temp; // PUT TEMP INTO y

    return;
}

int main() {
    // LOCAL VARIABLES DEFINITION
    int a = 100;
    int b = 200;

    printf("Before swap, value of a: %d\n", a);
    printf("Before swap, value of b: %d\n", b);

    swap(&a, &b);

    printf("After swap, value of a: %d\n", a);
    printf("After swap, value of b: %d\n", b);

    return 0;
}
```

- **a** and **b** are not pointer so we have to pass in the address.

→ because the function takes pointers.

⇒ whenever you have a function that takes pointers and you don't declare a pointer, you have to use the **address of** operator.

- it's the same thing how we use **scanf()**

- now you're gonna notice when we invoke the function, the variables **a** and **b** are gonna be swapped:

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
→
```

⇒ we're able to change the values of variables inside the function because we're making a copy of their addresses.

- this is one of the big advantages of pointers.

• Summary of Syntax:

- ◊ You can communicate two kinds of information about a variable to a function:

- `function1(x);`

- you transmit the value of **x** and the function must be declared with the same type as **x**:

- `int function1(int num)`

- `function2(&x);`

- you transmit the address of **x** and requires the function definition to include a pointer to the correct type:

- `int function2(int * ptr)`

⇒ **so whenever you use pointers you have to make sure that if you're passing in data and that data is not a pointer, you have to get the address of it.**

• const pointer Parameters:

- ◊ You can qualify a function parameter using the **const** keyword:

- it indicates that the function will treat the argument that is passed for this parameter as a **constant**.

- it's only useful when the parameter is a pointer.

◊ **The only reason you make a pointer constant is because you don't wanna have values changing outside of the function.**

◊ You apply the **const** keyword to a parameter that is a pointer to specify that **a function will not change the value to which the argument points**.

- code example:

```
bool SendMessage(const char* pMessage) {
    // CODE TO SEND THE MESSAGE
    return true;
}
```

- the type of the parameter, **pMessage**, is a pointer to a **const char**:

- ⇒ it is the **character** value that's **constant**, not its address.

- ⇒ you could specify the pointer itself as **const** too, but this makes little sense because the address is passed by value:

- you cannot change the original pointer in the calling function.

- ◊ The compiler knows that an argument that is a pointer to constant data will be safe.
- ◊ **You might wanna use this method if you don't want the changes to be reflected outside of the function.**
- ◊ If you pass a pointer to constant data as the argument for a parameter, then the parameter must be used like in the above example.
 - so when you invoke the function and you pass in a constant, then you have to have the function declaration also take a constant pointer.

- **Returning Pointers from a Function:**

- ◊ Returning a pointer from a function is a particularly powerful capability:
 - **it provides a way for you to return not just a single value, but a whole set of values.**
- ◊ With pointers you can now return more data from a function:
 - because the pointer could be pointing to a structure or other things, you can now return multiple values.
- ◊ You would have to declare a function returning a pointer:
 - so the function prototype and its declaration have to match.
- ◊ Declaration is done in the following way:

```
int * myFunction() {
    ...
}
```

- - use the asterisk * when you define it
→ and
 - have the return type of what (data type) that pointer is pointing to.

- ◊ **Be careful though, there are special hazards related to returning a pointer:**
 - **use local variables to avoid interfering with the variable that the argument points to.**
 - it is somewhat problematic when you're returning pointers, because if you modify that pointer inside the function and then return it, it could be something that you don't intended to do.

104. (Challenge) Using Pointers as parameters

- In this challenge we're gonna test your understanding of the concept of **pass by reference**.
 - ◊ even though C is doing everything **pass by value**.

- **Requirements:**

- Write a **function** that **squares a number by itself**
 - the function should **define as a parameter** an **int pointer**
 - int ***
 - All you're gonna have to do is dereference the pointer and multiply it by itself.
 - You don't have to return any data from this function.
 - The last thing this program should do is print out the value of the variable (pointer) that you passed to the function.
 - You don't have to create a pointer variable, all you have to do is create a primitive data type like an **int** or a **float** and when you pass it in as an argument, you use the **address of** operator.
- ◊ What you're gonna notice here is that the variable's value is gonna change.

My Code

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 15 - USING POINTERS AS PARAMETERS
DATE: MAY 3RD, 2022
*/



#include <stdio.h>

// FUNCTION PROTOTYPES
void square(int *ptr);

// MAIN FUNCTION
void main(void){
    int myInt = 5;

    printf("\nThe original value is: %d\n", myInt);

    // PASS BY REFERENCE FUNCTION
    square(&myInt);

    printf("\nThe value after 'pass by reference' is: %d\n", myInt);

    return;
}

// SQUARE FUNCTION
void square(int *ptr){
    int squaredValue = *ptr;

    squaredValue *= squaredValue;

    *ptr = squaredValue;
}
```

105. (Demonstration) Using Pointers as parameters

•

Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>

void square(int * x);

int main()
{
    int num = 5;

    square(&num);
    printf("The square of the given number is %d\n", num);

    return 0;
}

void square(int * x)
{
    *x = (*x) * (*x);
}
```

106. Dynamic Memory Allocation

- Dynamic memory allocation is directly related to pointers.
- You can only employ this on pointers.

- **Overview:**

- ◊ Whenever you define a variable in C, the compiler automatically allocates the correct amount of storage for you base on the data type.
- ◊ When you create a pointer memory is allocated for you so that it stores a pointer type, an address:
 - that doesn't mean that memory is allocated for that address.
 - up until this point whenever we have assigned data to a pointer, we've always assigned an existing variable:
 - so if we want to assign data to a pointer we've use the **address of** operator on variable that already had memory allocated for it.
- ◊ But what if you wanted to create a pointer and you didn't wanna assign it the **address of** operator.
 - you wanted to dynamically allocate memory for it.
- ◊ It is frequently desirable to be able to dynamically allocate storage while a program is running:
 - allocate memory as you need it.
- ◊ If you have a program that is designed to read in a set of data from a file into an array in memory, you have three choices:
 - define the array to contain the maximum number of possible elements at compile time.
 - use a variable-length array to dimension the size of the array at runtime.
 - allocate the array dynamically using one C's memory allocation routines.
 - **this is the best choice**
- ◊ There's different functions that you can call to do this.

- **Dynamic Memory Allocation:**

- ◊ With the first approach, you have to define your array to contain the maximum number of

elements that would be read into the array.

- this is what we've been doing until this point.

- ◊ You'd do something like this:

- ```
int dataArray[1000];
```

- the data file cannot contain more than 1000 elements, if it does, your program will not work:

- if it is larger than 1000 you must go back to the program, change the size of the array to be larger and recompile it.

- no matter what value you select, you always have the chance of running into the same problem again in the future.

- ◊ **Using the dynamic memory allocation functions, you can get storage as you need it:**

- **this approach enables you to allocate memory as the program is executing.**

- the advantages of this are:

- you don't create a ton of memory that you don't use, you're only gonna create as much as you need.

- and if you do need more you can allocate more, you don't have to go back, change the size and recompile.

- ◊ Dynamic memory allocation depends on the concept of a pointer and provides a strong incentive to use pointers in your code.

- this is one of the big reasons to use pointers.

- this is all about being an efficient language and not using a ton of memory.

- ◊ C runs on embedded systems, real time programming and this is one of the reasons why:

- because you can limit how much memory you use based on the need.

- ◊ Dynamic memory allocation allows memory for storing data to be allocated dynamically when your program executes:

- **allocating memory dynamically is possible only because you have pointers available.**

- ◊ The majority of production programs will use dynamic memory allocation:

- if you're writing code in C, you're gonna use this.

- even if you're a beginner you're gonna use it, most useful programs will employ this.

- ◊ Allocating data dynamically allows you to create pointers at runtime that are just large enough to hold the amount of data you require for the task so you're not wasting memory.

- this is vital

- **Heap vs. Stack:**

- ◊ There are two ways that you can store memory in a program (two different data structures):
  - **Stack**
  - **Heap**
- ◊ **Dynamic memory allocation reserves space in a memory area called the *heap*.**
- ◊ When you store data in the heap it allows you for more change:
  - you can arbitrarily change the size of data objects and sticks around a lot longer, it sticks around the entirety of your program.
- ◊ The stack is another place where memory is allocated (it is more limiting):
  - **function arguments and local variables in a function are stored here.**
  - **when the execution of a function ends, the space allocated to store arguments and local variables is freed.**
    - so memory located in the stack doesn't stick around for very long.
      - that's one of the reasons when you dynamically allocate memory it's on the heap,
      - ⇒ because you control when you use it and when you don't use it, you allocate it and you delete it.
  - ◊ Stuff on the stack you don't control, it automatically gets allocated and deleted.
  - ◊ The memory in the **heap** is different in that it is controlled by you:
    - when you allocate memory on the heap, it is up to you to keep track of when the memory you have allocated is no longer required.
    - you must free the space you have allocated to allow it to be reused.
      - this is very important because memory is very scarce resource.
      - for programs that embedded running on devices you don't have as much memory as on desktops or laptops
        - so you wanna control how much memory you use and you also wanna release the memory when you're done using it, so that other applications can use it.
        - ⇒ **this is more important in low level embedded type systems.**
    - ◊ This is a freedom that many languages don't have:
      - this can make the program much more efficient and faster.

## 107. *malloc*, *calloc*, and *realloc*

- Now that we understand dynamic memory allocation, we have to implement it.
- We're gonna talk about three really important functions:
  - ◊ **malloc()**
  - ◊ **calloc()**
  - ◊ **realloc()**
- **malloc()** :
  - ◊ The simplest standard library function that allocates memory at runtime is called **malloc()**:
    - you need to include the **stdlib.h** header file
    - you need to specify the number of bytes of memory that you want allocated as the argument.
    - it returns the address of the first byte of memory that it allocated.
    - **because you get an address returned, a pointer is the only place to put it.**
      - this is why dynamic memory allocation is only possible via pointers.
  - ◊ Here's how you do it:
    - ```
int *pNumber = (int*)malloc(100);
```

 - you could create your pointer and set it to **NULL** and then in the next line you could **malloc()**.
 - or you could do it like the previous example.
 - we're gonna create 100 bytes for **malloc()** and then we're gonna **cast it to an int pointer**.
 - because **malloc()** only returns an address, you have to cast it to an **int** pointer, that address is stored in the **int** pointer.
 - ◊ In the previous example, you have requested 100 bytes of memory and assigned the address of this memory block to **pNumber**:
 - **pNumber** will point to the first **int** location at the beginning of the 100 bytes that were allocated.
 - the instructor can hold **25 int** values on his computer, because they require **4 bytes** each.
 - assuming that type **int** requires **4 bytes**.
 - ◊ The problem with this code is that it's not gonna work on different systems:
 - because you're allocating based on the assumption of the **int** size.
 - so it's not good to use **malloc()** in this manner, with a constant number, 100 in this case.

- ◊ What you wanna do is use the **sizeof()** operator:
 - it would be better to remove the assumption that ints are 4 bytes:
 - ```
int *pNumber = (int*)malloc(25*sizeof(int));
```

      - in this example we're allocating memory for 25 integers.
      - ⇒ this will make the code much more portable, it'll work on different systems.
      - you still cast the return value because it has to be pointing to a type int.
- ◊ You always wanna use the **sizeof()** operator when dynamically allocating memory.
- ◊ The argument to **malloc()** above is clearly indicating that sufficient **bytes** for accommodating **25 values** of **type int** should be made available:
- ◊ Also notice the cast:
  - **(int\*)**
  - which converts the address returned by the function to the type **pointer to int**:
  - **malloc()** returns pointer of type **pointer to void**, so you have to cast.
- ◊ You can request any number of bytes:
  - however it is dependent on how much memory you have on your system.
- ◊ If the memory that you requested can not be allocated for any reason:
  - **malloc()** returns a pointer with the value **NULL**
    - this is good because the pointer won't be pointing to anything so you can't corrupt or change memory inadvertently.
    - it is always a good idea to check any dynamic memory requests immediately using an **if statement** to make sure the memory is actually there before you try to use it.
    - code example:
 

```
int *pNumber = (int*)malloc(25*sizeof(int));
if (!pNumber) {
 // CODE TO DEAL WITH MEMORY ALLOCATION FAILURE
}
```
    - ⇒ remember:
      - the **NULL** is a zero so you can just use a **not (!)** on it
      - ⇒ if you're not able to allocate memory in your program, you're in trouble.
      - so what you should do in the code to deal with the memory allocation failure is:
        - you should abort, you call exit, get out of your program.
  - ◊ You can at least display a message and terminate the program:
    - this is much better than allowing the program to continue and crash when it uses a **NULL** address to store something.

- **Releasing Memory:**

- ◊ When you allocate memory dynamically, **you should always release the memory when it is no longer required.**

- ◊ Memory that you allocate on the heap will be automatically released when your program ends:

- but it's better and cleaner to explicitly release the memory when you are done with it, even if it's just before you exit the program.

- this is a **strict rule**:

- **always release memory when you allocate it**

- ◊ A memory leak occurs when you allocate some memory dynamically and you do not retain the reference to it, so you are unable to release the memory:

- it often occurs within a loop when you're allocating memory inside it, use it and then jump out of the loop and you no longer release it.

- because you do not release the memory when it is no longer required, your program consumes more and more of the available memory on each loop iteration and eventually may occupy it all.

- ◊ **To free memory that you have allocated dynamically, you must still have access to the address that references the block of memory.**

- ◊ To release the memory for a block of dynamically allocated memory whose address you have stored in a pointer:

- all you have to do is call the **free()** function:

- code example:

```
→ free(pNumber);
 pNumber = NULL;
```

- ⇒ the **free()** function is also in the **stdlib.h** header.

- ⇒ you just pass in the pointer to the free function.

- ⇒ and after you pass in that pointer you wanna set it to **NULL**:

- because that's saying that you have no memory allocated for it.

- ◊ The **free()** function has a formal parameter of type **void\***

- you can pass a pointer of any type as the argument.

- you don't have to cast it or anything like that.

- ◊ As long as **pNumber** contains the **address that was returned when the memory was allocated**, the **entire block of memory will be freed** for further used.

- this is very significant when you're on an embedded system.
- ◊ You should always set the pointer to **NULL** after the memory that it points to has been freed.

- **calloc()** :

- ◊ The **calloc()** function offers a couple of advantages over **malloc()** :
  - it allocates memory as a number of elements of a given size.
  - it initializes the memory that is allocated so that all bytes are zero.
    - remember:
    - **always initialize your data with something.**
- ◊ The **calloc()** function requires two arguments values:
  - number of data items for which space is required.
  - size of each data item.
- ◊ It's declared in the **stdlib.h** header file.
- ◊ Code example:
  - ```
int *pNumber = (int*) calloc(75, sizeof(int));
```

 - it's a little bit cleaner because you don't have to do multiplication.
 - in **malloc()** you'd have to multiply the **75*sizeof(int)**
- ◊ The return value will be **NULL** if it was not possible to allocate the memory requested (in this case you should abort):
 - it's very similar to using **malloc()**, **but the big plus is that you know the memory area will be initialized to 0.**

- **realloc()** :

- ◊ The **realloc()** function enables you to reuse or extend memory that you previously allocated using **malloc()** or **calloc()**.
- ◊ This is a way to dynamically change how much memory is allocated while you're running:
- ◊ So you could initially create so much memory and then you realize that that's not enough

memory:

- well all you can do there is you can just call **realloc()** and this will reallocate memory.

◊ It expects two argument values:

- a pointer containing an address that was previously returned by a call to **malloc()** or **calloc()**

- the size in bytes of the new memory that you want allocated.

◊ This will allocate the amount of memory you specify by the second argument:

- it transfers the contents of the previously allocated memory referenced by the pointer that you supply as the first argument to the newly allocated memory.

- it returns a **void*** pointer to the new memory or **NULL** if the operation fails for some reason.

- so you have to recast it when you call **realloc()**

▪ **The most important feature of this operation is that *realloc()* preserves the contents of the original memory area:**

- you're just extending that memory.

→ it may be fragmented because you may not have been able to do it in sequence, but you still have all that memory that you can assign to now.

◊ Code example:

```
int main(){
    char *str = NULL;

    // INITIAL MEMORY ALLOCATION
    str = (char *) malloc(15 * sizeof(char));
    strcpy(str, "Juls");
    printf("String = %s, Address = %u\n", str, str);

    // REALLOCATING MEMORY
    str = (char *) realloc(str, 25 * sizeof(char));
    strcat(str, ".com");
    printf("String = %s, Address = %u\n", str, str);

    // FREE THE MEMORY
    free(str);
    str = NULL;

    return(0);
}
```

- always call **free()** when you call **malloc()** or **calloc()**.

• Guidelines:

◊ Avoid allocating lots of small amounts of memory:

- it's not efficient
 - allocating memory on the heap carries some overhead with it:
 - allocating many small blocks of memory will carry much more overhead than allocating fewer blocks.
-
- ◊ Only hang on to memory as long as you need it:
 - as soon as you are finished with a block of memory on the heap, release the memory.
 - memory is a scarce resource and when it's on the heap is even more scarce.
 - ◊ Always ensure that you provide for releasing memory that you have allocated:
 - decide where in your code you will release the memory when you write the code that allocates it.
 - make sure you have access to that pointer.
 - ◊ Make sure you do not inadvertently overwrite the address of memory you have allocated on the heap before you have released it:
 - this will cause a memory leak.
 - be especially careful when allocating memory within a loop.

108. (Challenge) Using Dynamic Memory

- In this challenge we'll test our understanding of how dynamic memory allocation works.

- **Requirements:**

- Write a program that allows a user to input a text string.

- The program will print out the text that was inputted.

- The program will utilize dynamic memory allocation.

- The user can enter the limit of the string they are entering:

- you can use this limit when invoking **malloc()**

- You should only create a **char pointer**, no character arrays.

- remember when you use a character array the memory is automatically already allocated for you based on the size that you put in and we don't want to do this.

- Be sure to **free()** the memory that was allocated.

My Code

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 16 - DYNAMIC MEMORY ALLOCATION
DATE: MAY 3RD, 2022
*/

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

void main(void){
    // VARIABLE DECLARATION
    char *pString = NULL;
    int byteAmount = 0;

    printf("\nPlease enter the amount of bytes that you wish to allocate: ");
    scanf("%d", &byteAmount);

    printf("\nPlease enter the string: ");
    scanf("%p", pString);

    // DYNAMIC MEMORY ALLOCATION
    pString = (char*)calloc(byteAmount, sizeof(char));

    // VALIDATE IF IT CAN ALLOCATE MEMORY
    if(!pString)
        exit;

    // PRINT INPUT TEXT
    printf("\nThe text is: %s", pString);

    // FREE MEMORY
    free(pString);
    pString = NULL;

    return;
}
```

- Working code after fixing it:

```
/*
AUTHOR: JFITECH
PURPOSE: CHALLENGE 16 - DYNAMIC MEMORY ALLOCATION
DATE: MAY 3RD, 2022
*/

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

void main(void){
    // VARIABLE DECLARATION
```

```
char *pString = NULL;
int byteAmount = 0;

printf("\nPlease enter the amount of bytes that you wish to allocate: ");
scanf("%d", &byteAmount);

// DYNAMIC MEMORY ALLOCATION
pString = (char*)calloc(byteAmount, sizeof(char));

// VALIDATE IF IT CAN ALLOCATE MEMORY
if (pString != NULL) {
    printf("\nPlease enter the string: ");
    scanf("%s", pString);
    // PRINT INPUT TEXT
    printf("\nThe text is: %s", pString);
}

// FREE MEMORY
free(pString);
pString = NULL;

return;
}
```

109. (Demonstration) Using Dynamic Memory

- When using character pointers you don't have to dereference the value.

Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int size;
    char *text = NULL;

    printf("Enter limit of the text: \n");
    scanf("%d", &size);

    text = (char *) malloc(size * sizeof(char));

    if (text != NULL)
    {
        printf("Enter some text: \n");
        scanf(" ");
        gets(text);

        printf("Inputted text is: %s\n", text);
    }

    free(text);
    text = NULL;
    return 0;
}
```

Section 13: Structures

110. Creating and Using Structures

- This is a new concept in C that allows you to group items together:
 - ◊ similar to arrays, but more powerful.

- **Overview:**

- ◊ Structures in C provide another tool for grouping elements together:
 - a powerful concept that you will use in many C programs that you develop.
- ◊ It's kind of thought as an object (OOP).
 - but only with data members, no action behaviors, no methods.
- ◊ You can group different items together in a single name and those can essentially represent data attributes.
- ◊ Suppose you wan to **store** a **date** inside a program:
 - we could create variables for **month**, **day** and **year** to store the date.
 - code example:
→ `int month = 9, day = 25, year = 2015;`
- ◊ Suppose your program also needs to **store** the **date of purchase** of a **particular item**:
 - you must keep track of three separate variables for each date that you use in the program.
 - this could get very repetitive and you'd end up creating a lot of variables.
 - **these variables are logically related and should be grouped together.**
- ◊ It would be much better if you could somehow **group** these sets of three variables together with a particular name:
 - this is precisely what the structure in C allows you to do.
 - you can then create different variables of that name and access the elements in that name, the different data attributes:
 - they're called **members**.

- **Creating a Structure:**

- ◊ A structure declaration describes **how a structure is put together**:

- **what elements are inside the structure.**

- these elements are members.

- ◊ The **struct** keyword enables you to define a **collection of variables of different types** called a **structure** that you can treat **as a single unit**:

- code example (using the previous example):

```
struct date {  
    int month;  
    int day;  
    int year;  
};
```

- by using the **struct** keyword all you have to do is:

- ⇒ provide a **name**

- referred to as the **tag name**

- ⇒ the **struct** keyword tells the compiler that this is gonna have specific functionality that allow you to group elements.

- ⇒ inside the curly braces **{ }** you can define **any type of variables that you want**:

- you can even put pointers in there.

- **what's inside the curly braces are called the members of the struct.**

- ◊ The above statement defines what a date structure looks like to the C compiler:

- there is no memory allocation for this declaration.

- all you're saying is that this is a structure that exists.

- we haven't started using it yet, this is how it's gonna be defined.

- ◊ The variable names within the **date structure, month, day** and **year**, are called:

- **members**

- or

- **fields**

- **members** of the structure appear between the braces that follow the **struct tag name date**.

- ◊ It's different from an array because there's:

- no indices
- you're not specifying a size
- there's different data types inside it

- **Using a structure:**

- ◊ The definition of **date** defines a new type in the language:

- you can think of it as a new data type.

- variables can now be declared to be of type:

- **struct date**

- code example:

⇒ `struct date today;`

- the above statement declares a **variable** to be of type **struct date**:

- ◊ **memory is now allocated for the variable above.**

- memory is allocated because you declared the variable.
- memory is allocated for the structure.
- the reason why it's allocated is because there's no pointers in there:
 - if there were pointers in that **struct** you'd have to allocate memory for that pointer

- ◊ **memory is allocated for three integer values for each variable.**

- the **today** variable is able to **hold three values**.

- ◊ and if you need to declare more dates you can create more variables of that same type:

- so you no longer need to have three separate variables for a **purchase date**, three separate variables for the **current date**, three separate variables for **billing date**:
 - you can have one variable or two variables of different instances of that **struct**.

- ◊ Be certain that you understand the **difference between defining a structure and declaring variables of the particular structure type**.

- **Accessing member in a struct:**

- ◊ Now that you know how to define a structure and declare structure variables, you need to be able to refer to the members of a structure.

- ◊ A structure variable name is not a pointer:

- you need a special syntax to access the members.

- ◊ You refer to a member of a structure by writing:

- the **variable name followed by a period, followed by the member variable name**

:

- the period between the structure variable name and the member name is called the member selection operator:

- it's usually called the **dot operator**.

- there are no spaces permitted between the variable name, the period, and the member

name.

- ◊ To set the value of the **day** in the variable **today** to **25**, you write:

```
today.day = 25;  
today.year = 2015;
```

- ◊ To test the value of the **month** to see if it is equal to 12:

```
if (today.month == 12)  
    nextMonth = 1;
```

- ◊ Code example:

```
struct date{  
    int month;  
    int day;  
    int year;  
};           // AT THIS POINT NO MEMORY IS ACTUALLY ALLOCATED FOR THIS struct  
  
struct date today;  
  
today.month = 9;  
today.day = 25;  
today.year = 2015;  
  
printf("Today's date is %i/%i%.2i.\n", today.month, today.day, today.year %  
100);
```

• Structures in expressions:

- ◊ When it comes to the evaluation of expressions, structure members follow the same rule as ordinary variables do:
 - division of a integer structure member by another integer is performed as an integer division.

- code example:

```
→ century = today.year / 100 +1;
```

• Defining the structure and variable at the same time:

- ◊ You do have some flexibility in defining a structure:
 - it is valid to declare a variable to be of a particular structure type at the same time that the

structure is defined.

- include the variable name (or names) before the terminating semicolon of the structure definition.
- you can also assign initial (initialize) values to the variables in the normal fashion.

◊ You should always:

- define a tag **name**
 - and
- define the **structure** in one place of your code
 - and
- then declare variables of it in another.

◊ You can also define **anonymous (no name)** structures.

◊ Code example:

```
struct date{  
    int month;  
    int day;  
    int year;  
} today;
```

- we create a variable called **today**
- this is both creating the variable of type **struct date** and the definition of the **struct date**.

◊ In the above, an instance of the structure, called **today**, is declared at the same time that the structure is defined;

- **today** is variable of type **date**
- **we also refer to this as an instance of the structure.**

▪ you could also after you write the **today** initialize those variables by using the brackets and saying:

- **month = 5;**
- and so on...
- ⇒ just like you initialize any other integers.

• Unnamed Structures:

- ◊ You also do not have to give a structure a tag name:
 - if all of the variables of a particular structure type are defined when the structure is defined, the structure name can be omitted.
 - this means that you're only gonna use the structure just one time and one time only.

◊ Code example:

```

struct{
    int day;      // STRUCTURE DECLARATION AND...
    int year;
    int month;
} today;                      // STRUCTURE VARIABLE DECLARATION COMBINED

```

- notice that there's no **tag name** after the **struct** keyword.
- this is one struct that we're only gonna use one time.
→ so we create the variable **today** and then we're never gonna use it again.

- ◊ A disadvantage of the above is that you can no longer define further instances of the structure in another statement:
 - all the variables of this structure type that you want in your program must be defined in the one statement.

• Initializing structures:

- ◊ Initializing structures is similar to initializing arrays:
 - the elements are **listed inside a pair of braces**, with **each element separated by a comma**.
 - the initial values listed inside the curly braces must be constant expressions.

◊ Code example:

```
struct date today = {7, 2, 2015};
```

- ◊ Just like an array initialization, fewer values might be listed than are contained in the structure.

◊ Code example:

```
struct date date1 = {12, 10};
```

- the previous code sets **date1.month** to **12** and **date1.day** to **10** but gives no initial value to **date.year**

◊ You can also specify the member names in the initialization list:

- enables you to initialize the members in any order, or to only initialize specified members:
- code example:

```
.member = value; // SYNTAX
→ struct date date1 = {.month = 12, .day = 10};
```

- ◊ To set just the year member of the date structure variable **today** to **2015**:

```
▪ struct date today = { .year = 2015 };
```

- **Assignment with compound literals:**

- ◊ Compound literal assignment.
- ◊ You can assign one or more values to a structure in a single statement using what is known as compound literals:
 - this is only in **C11**.
- ◊ Code example:
 - `today = (struct date) { 9, 25, 2015 };`
- ◊ This statement can appear anywhere in the program:
 - it is not a declaration statement.
 - the type cast operator is used to tell the compiler the type of the expression.
 - the list of values follows the cast and are to be assigned to the members of the structure, in order.
 - listed in the same way as if you were initializing a structure variable.
 - the advantage of this is you don't assign each member variable individually on separate lines, you can do it all on one line.
- ◊ You can also specify values using the **.member** notation:
 - code example:
 - `today = (struct date) { .month = 9, .day = 25, .year = 2015 };`
 - the advantage of using this approach is that the arguments can appear in any order.
- ◊ The key and the difference of this assignments it's is not happening in the initialization:
 - you're doing it somewhere else in the program to assign new values to that type.

111. Structures and Arrays

- Let's talk about how structures are related to arrays.
- They're different from arrays because you can have different data types in them and you can group them that way.
- You can also have **arrays of structures**.

• Arrays of Structures:

- ◊ You have seen how useful a structure is in enabling you to logically group related elements together:
 - for example, it is only necessary to keep track of one variable, instead of three, for each **date** that is used by a program.
 - to handle 10 different dates in a program, you only have to keep track of **10** different variables instead of **30**.
- ◊ You can create multiple structures that are inside of an array.
- ◊ A better method for handling the 10 different dates involves the combination of two powerful features of the C programming languages:
 - structures and arrays,
 - it is perfectly valid to define an array of structures.
 - **the concept of an array of structures is a very powerful and important one in C.**
- ◊ Declaring an array of structures is like declaring any other kind of array:
 - code example:
 - `struct date myDates[10];`
 - the previous code defines an array called **myDates**, which **consists of 10 elements**
 - ⇒ each element inside the array is defined to be of type **struct date**.
 - **which contains three members.**
 - ◊ **you're essentially storing 30 data elements.**
 - ◊ To identify members of an **array of structures**, you apply the same rule used for individual structures:
 - follow the structure name with the **dot operator** and then with the member name.
 - ◊ Referencing a particular structure element inside the array is quite natural:

- to set the second date inside the **myDates** array to **August 8th, 1986**.

- code example:

```
myDates[1].day = 8;
myDates[1].month = 8;
→ myDates[1].year = 1986;
```

• Initializing an Array of Structures:

- ◊ Initialization of arrays containing structures is similar to initialization of multidimensional arrays:

- code example:

```
- struct date myDates[5] = { {12, 10, 1975}, {12, 30, 1980}, {11, 15, 2005} };
```

→ the previous code sets the first three dates in the arrays **myDates** to **12/10/1975**, **12/30/1980**, and **11/15/2005**

⇒ because we didn't supply the other two elements in the **myDates** array those are not gonna be initialized to anything.

- ◊ To initialize just the third element of the array to the specified value:

- code example:

```
- struct date myDates[5] = { [2] = {12, 10, 1975} };
```

→ the previous code sets the third element of the **myDates** array to **12, 10** and **1975**.

- ◊ To set just the **month** and **day** of the **second element** of the **myDates** array to **12** and **30** you'd do the following

- code example:

```
- struct date myDates[5] = { [1].month = 12, [1].day = 30 };
```

• Structures containing Arrays:

- ◊ It is also possible to define structures that contain arrays as members:

- **most common use is set up an array of characters inside a structure.**

- ◊ Suppose you want to define a structure called **month** that contains as its members the number of days in the month as well as a **three-character** abbreviation for the month name:

- code example:

```
struct month{
    int numberOfDays;
    char name[3];
}
```

- this is not allocating any memory, this is just defining the structure.
 - ⇒ these are the elements that are contained inside of it.

- ◊ The previous code sets up a **month** structure that contains an **integer member** called **numberOfDays** and a **character member** called **name**:
 - **member name** is actually an array of three characters.
 - You can now define a variable of type **struct month** and set the proper fields inside **name** for January.
 - the variable would create memory for the elements inside the **struct**.
 - code example:

```
struct month aMonth;
aMonth.numberOfDays = 31;
aMonth.name[0] = 'J';
aMonth.name[1] = 'a';
aMonth.name[2] = 'n';
→
```

⇒ **normally when you have a character array you would use the *strcpy()* function and assign everything with double quotes.**

- ◊ You can also initialize the previous variable to the same values by doing something similar when you declare the variable:

```
▪ struct month aMonth = {31, {'J', 'a', 'n'}};
```

- ◊ You can set up **12-month structures inside an array to represent each month of the year**:

```
▪ struct month months[12];
```

112. Nested Structures

- Structures inside structures.

- **Nested Structures:**

- ◊ C allows you to define a structure that itself contains other structures as **one or more** of its members.

- ◊ You have seen how it is possible to logically group the **month**, **day** and **year** into a structure called **date**:

- how about grouping the **hours**, **minutes** and **seconds** into a **structure** called **time**.
 - code example:

```
struct time {  
    int hours;  
    int minutes;  
    int seconds;  
};
```

- ◊ In some applications, you might have the need to **group** both a **date** and a **time** together:
 - you might need to set up a list of events that are to occur at a particular **date** and **time**.

- ◊ You want to have a convenient way to associate both the **date** and **time** together:
 - define a new structure, called, for example, **dateAndTime**, which contains as its members two elements:

- **date** and **time** (which are both structures).

- so now we have a nested structure:

- ⇒ code example:

```
struct dateAndTime{  
    struct date sdate;  
    struct time stime;  
}
```

- ◊ the syntax is very similar to creating other structures the only thing different here is:

- inside the actual structure as the **element names** you have:
 - the **struct** keyword
 - the **name of the struct**
 - and the **variable name**.

- ◊ In the previous example:

- the first member of this structure is of type:
 - **struct date**

→ and it's called:

- **sdate**

- ◊ Variables can now be defined to be of type **struct dateAndTime**:

- `struct dateAndTime event;`

- now we don't have to create different variables for **date** and **time**, we can create one and that **structure** is gonna store both the date and the time.

- **Accessing members in a nested structure:**

- ◊ To reference the **date** structure of the variable **event**, the syntax is the same as referencing any member:

- using the **dot operator**.

- ◊ Code example:

- `event.sdate;`

- ◊ To reference a particular member inside one of these structures, a period followed by the member name is tacked on the end:

- the below statement sets the month of the date structure contained within **event** to **October**, and adds one to the **seconds** contained within the time structure:

- code example:

- `event.sdate.month = 10;`
→ `+event.stime.seconds;`

- ◊ The **event** variable can be initialized just like normal:

- the following code sets the **date** in the variable **event** to **February 1st, 2015**, and sets the **time** to **3:30:00**:

- `struct dateAndTime event = {{2, 1, 2015}, {3, 30, 0}}; }`

- ◊ You can use members' names in the initialization:

- code example:

- `struct dateAndTime event = {{.month = 2, .day = 1, .year = 2015}, {.hour = 3, .minutes = 30, .seconds = 0}}; }`

- this would be beneficial because you wouldn't have to specify the order.

- **An array of nested structures:**

- ◊ It is also possible to set up an array of **dateAndTime** structures:

- code example:

- `struct dateAndTime events[100];`

- the array **events** is declared to contain **100 elements** of type **struct dateAndTime**

- ⇒ the fourth **dateAndTime** element contained within the array is referenced in the usual way as **events[3]**.

- ◊ Using the previous example, to set the **first time element in the array to noon**:

- `events[0].stime.hour = 12;`
`events[0].stime.minutes = 0;`
`events[0].stime.seconds = 0;`

- **Declaring a structure within a structure:**

- ◊ You can define the **Date** structure **within** the **Time** structure definition:

- code example:

- `struct Time{`
 `struct Date{`
 `int day;`
 `int month;`
 `int year;`
 `} dob;`

 `int hour;`
 `int minute;`
 `int second;`
 `}`

- you can notice that at the end of the **Date struct** we're creating a variable called **dob**

- the limitation of this is that you won't be able to use the **Date** structure anywhere unless you use an inside time:

- ⇒ you'd not be able to use **Date** by itself.

- ◊ In the previous code example:

- the declaration is **enclosed** within the scope of the **Time** structure definition:

- it does not exist outside it.

- it becomes impossible to declare a **Date** variable external to the **Time** structure.
- the only time you can ever use a **Date** structure variable is when you create the **Time** struct
 - this is somewhat limited, but you'd use it in a case where you'd only need the **Date** associated with the **Time**:
 - the **Date** should never be by itself.

113. Structures and Pointers

- We can have elements inside of a structure be a pointer and we can also make our structures themselves be pointers.
- C allows for pointers to structures:
 - ◊ when we create a variable of type **struct** we can create it as a pointer.
- Pointers to structures are easier to manipulate than structures themselves:
 - ◊ makes things a lot easier and more efficient.
- In some **older** implementations of the C language, a **structure cannot be passed as an argument to a function**:
 - ◊ but a **pointer to a structure can**:
 - this is another reason why you wanna have a structure be a pointer when you declare it:
 - because then you can pass it around in older versions of C.
 - Even if you can pass a structure as an argument, passing a pointer is more efficient:
 - ◊ in this case it's even more efficient because:
 - if you're passing in an entire structure, if it is large and it contains a lot of elements:
 - you're copying all the element in the structure to that parameter:
 - if you're passing in a pointer you're just copying the address.
 - Many data representations that use structures contain **pointers** to other structures.

• Declaring a **struct** as a pointer:

- ◊ You can define a **variable** to be a **pointer** to a **struct**:
 - code example:
 - `struct date *datePtr;`
 - and you're probably gonna want to initialize it to **NULL**:
 - ⇒ `struct date *datePtr = {NULL, NULL, NULL};`
 - it creates memory to store the actual pointer:
 - ◊ it's not creating memory for the data inside it.
 - ◊ The variable **datePtr** can be assigned just like other pointers:
 - **you can set it to point to todaysDate (if we already had a declaration of todaysDate) with the assignment statement.**
 - code example:

```
→ datePtr = &todaysDate;
```

⇒ this sets the **datePtr** variable to point to the address where **todaysDate** is stored at.

◊ You can then indirectly access **any of the members** of the **date** structure pointed to by **datePtr**:

- code example:

```
- (*datePtr).day = 21;
```

→ the above has the effect of setting the day of the **date** structure pointed to by **datePtr** to **21**:

⇒ **parentheses are required** because the **structure member operator . has higher precedence than the indirection operator ***.

• Using structs as pointers:

◊ If we wanted to test the value of **month** stored in the **date** structure pointed to by **datePtr**, we would do something like the following:

- code example:

```
if ((*datePtr).month == 12)  
    ...
```

◊ When you wanna access what the structure is pointing to you don't have to dereference it as previously stated.

◊ Pointers to structures are so often used in C that a special operator exists:

- the structure pointer operator:

- ->

→ which is the **dash** followed by the **greater than sign**

⇒ this allows you to not have to dereference the pointer.

- code example:

```
// INSTEAD OF DOING THE FOLLOWING:  
(*x).y;  
  
// YOU COULD MORE CLEARLY EXPRESS IT AS:  
◊ x->y;
```

▪ because **x** is a pointer and you don't have to dereference it and you wanna access the element inside that structure, whatever you're pointing to.

◊ The previously **if statement** can be conveniently written as:

- code example:

```
- if(datePtr->month == 12);
```

- if this were a nested structure you'd do the same as above but you'd have to add . after **month** to access the elements of the nested structure.

- ◊ Code example:

```
struct date{  
    int month;  
    int day;  
    int year;  
};  
  
struct date today, *datePtr;  
  
datePtr = &today;  
  
datePtr->month = 9;  
datePtr->day = 25;  
datePtr->year = 2015;  
  
printf("Today's date is %i/%i/.2i\n", datePtr->month, datePtr->day, datePtr->year % 100);
```

- ◊ Typically when you define **structs** you do it outside of a function:

- because then you can use them in any function.
 - it's kind of like a global.

- because when you define the structure you're not creating any memory, so you still create the variables as local ones or pointer:

- ⇒ but, the actual definition is outside the function, so if you wanna use it, you can use it in any function.

- **Structures containing pointers:**

- ◊ A pointer also can be a member of a structure:

- code example:

```
struct intPtrs{  
    int *p1;  
    int *p2;  
}
```

- ◊ We could allocate memory for a pointer if we didn't wanna assign an existing address of an existing pointer.

- ◊ In the previous example a structure called **intPtrs** is defined to contain two integer pointers:
 - the first one called **p1**.

- the second one called **p2**.
- ◊ You can define a variable of type **struct intPtrs**:
 - code example:
 - `struct intPtrs pointers;`
 - the variable **pointers** can now be used just like other **structs**:
 - ⇒ **pointers** itself is not a pointer, but a **structure variable** that **has two pointers as its members**.
 - so if we wanna access the member inside the **pointers** variable which is an **intPtrs** we don't use the `->` because **pointers** is not a pointer it's a **struct**.
 - ◊ we would use the **dot operator** to access **p1** and **p2**.

- ◊ Code example:

```
struct intPtrs{
    int *p1;
    int *p2;
};

struct intPtrs pointers;
int i1 = 100, i2;

pointers.p1 = &i1;
pointers.p2 = &i2;
*pointers.p2 = -97;

printf("i1 = %i, *pointers.p1 = %i\n", i1, *pointers.p1);
printf("i1 = %i, *pointers.p2 = %i\n", i2, *pointers.p2);
```

- if the **integer pointers** were not integer pointers but they were **character pointers**:
 - you'd have to allocate memory dynamically.

- ◊ In the previous code example we have a regular variable **pointer** but it **contains two elements that are pointers**:

- so if we wanna **assign to those two elements inside of the struct**:
 - we have to **dereference** them using the **indirection operator** `*`
 - if we wanna access the values that the pointers are pointing to, we also have to dereference them (last two print statements).

• Character Arrays or Character Pointers:

- ◊ Code example:

```
struct names{
    char first[20];
    char last[20];
};
```

- or

```
struct pnames{
    char * first;
    char * last;
};
```

- ◊ In the previous example you can do both, however, **you need to understand what is happening here.**

- ◊ Code example:

```
struct names veep = {"Talia", "Summers"};           // CONTAINS THE
CHARACTER ARRAYS
struct pnames treas = {"Brad", "Fallingjaw"}; // CONTAINS THE
POINTERS
printf("%s and %s\n", veep.first, treas.first);
```

- the **struct names** variable **veep**:

- strings are stored inside the structure.

- structure has allocated a total of **40 bytes** to hold the two names (they're character arrays).

- the **struct pnames** variable **treas**:

- strings are stored wherever the compiler stores strings constants.

- the structure holds the **two addresses**, which takes a total of **16 bytes** (for most systems).

- ⇒ all it's gonna do is:

- **allocate two pointers inside that structure.**

- **the struct pnames structure allocates no space to store strings.**

- ⇒ because it's only allocating space for the pointers themselves.

- ⇒ when you create those two pointer there's no memory allocated for those pointers, it allocates no space.

- it can be used only with strings that have had space allocated for them elsewhere:

- ⇒ such as **string constants** or **strings in arrays**.

- you have two pointers

- there's no memory allocated for that when you create a variable of that type:

- ⇒ it's just creating memory for the two variable pointers:

- they are **8 bytes** each.

- ◊ Whenever you have pointers inside of a structure you have to use **malloc()** or **calloc()**:

- otherwise they could be stored wherever string constants could be stored.

- ◊ The pointers in **pnames** structure **should be used only to manage strings** that **were**

created and allocated elsewhere in the program.

- ◊ To actually allocate memory for what ***first** and ***last** are pointing to you have to do a:
 - **malloc()**
- ◊ One instance in which it does make sense to use a pointer in a structure to handle a string is:
 - if you are dynamically allocating that memory:
 - use a pointer to store the address.
 - it has the advantage that you can ask **malloc()** to allocate just the amount of space that is needed for a string.
 - other primitive data types are gonna allocate memory automatically for you.
- ◊ You're gonna use **malloc()** on any pointers created inside of a **struct**:
 - code example:

```
struct nameCount{  
    char *fname;           // USING POINTERS INSTEAD OF ARRAYS  
    char *lname;  
    int letters;  
}
```

- understand that the **two strings are not stored in the structure**:
 - ⇒ we only create memory for the two pointers which contain addresses.
 - ⇒ they are stored in the chunk of memory managed by **malloc()**
 - ⇒ the addresses of the two strings are stored in the structure (not what they're pointing to).
 - ⇒ addresses are what string-handling functions typically work with:
 - this is why you wanna use pointers in there, because then you can just pass them around a lot easier
 - we wanna create memory for ***fname** and ***lname** for what the pointers are pointing to:
 - ⇒ we wanna store that in a chunk managed by **malloc()**
- ◊ Code example:

```
struct nameCount{  
    char *fname;           // USING POINTERS INSTEAD OF ARRAYS  
    char *lname;  
    int letters;  
}  
  
void getInfo(struct nameCount *pst){  
    char temp[SLEN];  
    printf("Please enter your first name:\n");  
    s_gets(temp, SLEN);
```

```

// ALLOCATE MEMORY TO HOLD NAME
pst->fname = (char *) malloc(strlen(temp) + 1);

// COPY NAME TO ALLOCATED MEMORY
strcpy(pst->fname, temp);
printf("Please enter your last name:\n");
s_gets(temp, SLEN);
pst->lname = (char *) malloc(strlen(temp) + 1);
strcpy(pst->lname, temp);
}

```

- the **getInfo** function takes a pointer and we can pass around that pointer very easily.
- the **s_gets()** function is basically the same as **scanf()** function.
- on line **14** we allocate memory for **fname** inside that pointer (**pst**):
 - and we do it just like we allocate memory for any character:
 - ⇒ we call **malloc()**, we pass in the number of bytes, in this case:
 - malloc(strlen(temp) + 1)
 - ◊ the **+ 1** is for the **null terminator**.
 - ⇒ **malloc()** returns a **void*** so we cast it to **(char *)**.
- we can then do our copy very easily:
 - we can copy **temp** to **fname** of our **structure pst** that was passed in as a parameter.

- ◊ The **malloc()** call in the previous example is **critical for any structure that contain pointers**:
 - we passed the pointer in and it never had memory allocated for it:
 - so we're allocating memory inside here.
- ◊ What's shown in the previous example is what you're gonna need to do when have **pointers to character** inside your **structures**:
 - you have to do the **malloc()** :
 - because those pointers only store the address.
 - they allocated enough memory to store an address.
 - not enough memory to store a **string**.

114. Structures and Functions

- How structures are related to functions.
- How you can pass structures as an argument to a parameter in a function:
 - ◊ and how this is more efficient as well
- **Structures as arguments to functions:**
 - ◊ After declaring a structure named **Family**, how do we pass this structure as an argument to a function?
 - ◊ Code example:

```
struct Family{
    char name[20];
    int age;
    char father;
    char mother;
}

bool siblings(struct Family member1, struct Family member2) {
    if(strcmp(member1.mother, member2.mother) == 0){
        return true;
    } else {
        return false;
    }
}
```
 - we pass this **structure** to a function called **siblings**:
 - you'd set the type (**struct**) as the parameter name.
 - the function is taking two **struct** parameters
 - ⇒ on more advanced C you could do a **typedef** on the **struct**, this way you could name it something else and shorten it:
 - this would probably be better.
 - then we pass the variable name.
 - **member1.mother** and **member2.mother** are character arrays (character arrays are pointers essentially):
 - so we can pass to the **strcmp()** function
 - ⇒ **strcmp()** is gonna return **equals** if they're the same.
 - ◊ In the previous code the function has **two parameters**, each of which is a **structure of type Family**.

- **Pointers to Structures as function arguments:**

- ◊ You should use a pointer to a structure as an argument:
 - **it can take quite a bit of time to copy large structures as arguments, as well as requiring whatever amount of memory to store the copy of the structure.**
 - pointers to structures avoid memory consumption and the copying time (only a copy of the pointer argument is made).
- ◊ This is better to do because we're doing pass by value in C:
 - when you do **pass by value**, you copy.
 - if you have a pointer all you have to do is copy the address:
 - which is much quicker.
- ◊ If you have a small structure, no big deal:
 - but you definitely wanna use pointers in structures that are big.

- ◊ Code example:

```
bool siblings(struct Family *pMember1, struct Family *pMember2) {
    if(strcmp(pMember1->mother, pMember2->mother) == 0)
        return true;
    else
        return false;
}
```

- you just define the name of the type of the structure and you put the * before the variable name.

- and if you didn't want any of the data inside of the **structure** to change you'd:

- **const**

- ⇒ before the **structure** type.

- if you didn't want the **address to change**:

- you'd put the constant after the **struct Family** pointer.

- ◊ You can also use the **const** modifier to **not allow any modification of the member of the struct** (what the struct is pointing to):

- code example:

```
bool siblings(Family const *pMember1, Family const *pMember2) {
    if(strcmp(pMember1->mother, pMember2->mother) == 0)
        return true;
    else
        return false;
}
```

⇒ you could also put the **const** before the **Family** as well.

⇒ here **Family** has a **typedef** so we omit the **struct** keyword when declaring the function's parameters.

- otherwise it would be something like:

```
bool siblings(struct Family const *pMember1, struct Family const *pMember2)
{
    if(strcmp(pMember1->mother, pMember2->mother) == 0)
        return true;
    else
        return false;
}
```

◊ You can also use the **const** modifier to **not allow any modification of the pointer's address**:

- any attempt to change those structures will cause an error message during compilation.
 - so you'd do the following:

```
bool siblings(struct Family *const pMember1, struct Family *const pMember2)
{
    if(strcmp(pMember1->mother, pMember2->mother) == 0)
        return true;
    else
        return false;
}
```

⇒ what this is saying is:

- the value of the pointer (which is an address).

⇒ this is not very useful because when you **pass by value**, you're actually **passing in a copy of the address**.

- so this is redundant, you don't need to do this, but you can.

◊ The indirection operator in each parameter definition is now in front of the **const** keyword:

- not in front of the parameter name.
- you cannot modify the addresses stored in the pointers.
- it's the pointers that are protected here, not the structures to which they point.

• Returning a structure from a function (that's a pointer):

◊ The function prototype has to indicate the return value in the normal way.

◊ Code example:

```
// FUNCTION PROTOTYPES
struct Date my_fun(void);
```

- this is a prototype for a function taking no arguments that returns a structure of type

Date.

→ now we can return more than one element or one data item from a function:

- ⇒ if you're returning a **struct**, you can return 10 values, 10 different data items.
- ⇒ the **struct** allows you to return more than one from a function.

◊ It is often more convenient to return a pointer to a structure:

▪ when returning a pointer to a structure, it should be created on the heap (because all pointers are created on the heap).

- this means it's gonna be around for longer.

→ and this means that you can delete and manage it at arbitrary time.

▪ code example:

```
// FUNCTION PROTOYPE
double sum(struct funds moolah);

struct funds{
    char bank[FUNDLEN];
    double bankFund;
    char save[FUNDLEN];
    double saveFund;
};

int main(void){
    struct funds stan = {
        "Garlic-Melon Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
    };

    printf("Stan has a total of %.2f.\n", sum(stan));

    return 0;
}

double sum(struct funds moolah){
    return (moolah.bankFund + moolah.saveFund)
}
```

→ just assume that **FUNDLEN = 50**;

→ in this example **sum** is a double, we're not returning a pointer, we're just taking a pointer as a parameter.

• Reminder:

◊ The instructor mentioned earlier that you should always use pointers when passing structures to a function.

- the benefits of it are:
 - it works on older as well as newer C implementations.
 - because older versions of C don't allow structures to be passed, but they do allow pointers.
 - it is quick.

- ◊ However, you have less protection for your data:

- some operations in the called function could inadvertently affect data in the original structure.

- using the **const** qualifier solves that problem.

- use the **const** before the actual type, not after the asterisk.

- ◊ **Advantages of passing structures as arguments:**

- the function works with copies of the original data, which is safer than working with the original data.

- this is the only time you wanna pass a structure as an argument that's not a pointer.

- the programming style tends to be clearer.

- ◊ **Main disadvantages to passing structures as arguments:**

- older implementations might not handle the code.

- wastes time and space.

- especially wasteful to pass large structures to a function that uses only one or two members of the structure:

- we have to a lot of copying for each member.

- ◊ Programmers use structure pointers as function arguments for reasons of efficiency and use **const** when necessary.

- this is the combination that you wanna do.

- ◊ Passing structures by value is most often done for structures that are small.

115. (Challenge) Declaring and Initializing a structure

- How to declare and initialize structures.

- Requirements:

- Write a program that declares a **structure** and **prints out it's content**:
 - create an **employee structure** with **3 members**:
 - name (character array)**
 - hireDate (int)**
 - salary (float)**
- Declare** and **initialize** an **instance** of an **employee** type.
- Print** out the **contents** of this **employee**.
- Read in a **second employee from the console** and store it in a **structure of type employee**.
- Then **print** out the **contents** of this **employee**.

My Code

```
/*
AUTHOR: JFITECH
PURPOSE: STRUCTS AND POINTERS
DATE: MAY 12TH, 2022
*/



#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// STRUCT DEFINITION
struct employee{
    char name[20];
    int hireDate;
    float salary;
};

// MAIN FUNCTION
void main(void){
    // DECLARE VARIABLES
    struct employee Joe;
    struct employee employee2;

    // ASSIGN DATA TO THE FIRST EMPLOYEE STRUCT
    // SETS THE SIZE OF THE DATA INSIDE Joe.name TO NULL TERMINATORS
    memset(Joe.name, '\0', sizeof(Joe.name));
    // COPIES THE STRING "Joe" TO THE Joe.name ARRAY - ONLY COPIES THE FIRST 4
    // CHARACTERS (THIRD ARGUMENT)
    strncpy(Joe.name, "Joe", 4);
    Joe.hireDate = 10102015;
    Joe.salary = 78000.00;

    // PRINT FIRST EMPLOYEE'S INFO
    printf("\nEmployee's name: %s", Joe.name) ;
    printf("\nEmployee's hire date: %d", Joe.hireDate);
    printf("\nEmployee's salary: %.2f", Joe.salary);

    // READ INPUT FROM THE USER
    // NAME
    memset(employee2.name, '\0', sizeof(employee2.name));
    printf("\n\nPlease enter the second employee's name: ");
    scanf("%s", employee2.name);
    strncpy(employee2.name, employee2.name, 20);
    // HIRE DATE
    printf("\nPlease enter the second employee's hire date: ");
    scanf("%d", &employee2.hireDate);
    // SALARY
    printf("\nPlease enter the second employee's salary: ");
    scanf("%f", &employee2.salary);
```

```
// PRINT SECOND EMPLOYEE'S INFO
printf("\nSecond employee's name: %s", employee2.name) ;
printf("\nSecond employee's hire date: %d", employee2.hireDate);
printf("\nSecond employees salary: %.2f", employee2.salary);

return;
}
```

116. (Demonstration) Declaring and Initializing a structure

- Another way to **initialize** our **struct** is:

```
◊ struct employee emp = {"Mike", 1120, 76909.00f};
```

Instructor's Solution

```
#include <stdio.h>
#include <stdlib.h>

struct employee {
char name[30];
char date[15];
float salary;
};

int main()
{
/*declare and initialization of structure variable*/
struct employee emp={"Mike","7/16/15",76909.00f};

printf("\n Name: %s          ,emp.name);
printf("\n Hire Date: %s          ,emp.date);
printf("\n Salary: %.2f\n",emp.salary);

printf("\nEnter employee information: \n");

printf("Name: ");
scanf("%s", emp.name);

printf("\nHire Date: ");
scanf("%s", emp.date);

printf("\nSalary: ");
scanf("%f", &emp.salary);

printf("\n Name: %s          ,emp.name);
printf("\n Hire Date: %s          ,emp.date);
printf("\n Salary: %.2f\n",emp.salary);

return 0;
}
```

117. (Challenge) Structure Pointers and Functions

- This challenge is gonna get you familiar with elements in a structure that contain a pointer.

- Requirements:

- Write a C program that creates a **structure pointer** and passes it to a function:
 - create a structure named **item** with the following members:
 - itemName - char pointer**
 - quantity - int**
 - price - float**
 - amount - float (stores quantity * price)**
 - Create a **function** named **readItem** that takes a **structure pointer** of **type item** as a parameter:
 - this function should read in (from the user) a product **name**, **price** and **quantity**.
 - the contents read in should be stored in the passed in structure to the function.**
 - Create a **function** named **printItem** that takes as a parameter a **structure pointer of type item**:
 - the function should print out the contents of the parameter.
 - The **main function** should **declare an item and a pointer to the item**:
 - you will need to **allocate memory** for the **itemName** pointer.
 - remember to use **free()**
 - the **item pointer** should be **passed into both** the **read** and **print item functions**.

My Code

```
/*
AUTHOR: JFITECH
PURPOSE: STRUCTS AND POINTERS
DATE: MAY 12TH, 2022
*/



#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SLEN 30

// STRUCT DEFINITION
struct item{
    char *itemName;
    int quantity;
    float price;
    float amount; // STORES quantity * price
};

// FUNCTION PROTOTYPES
void readItem(struct item *pItem);
void printItem(struct item *pItem);

// MAIN FUNCTION
void main(void){
    // DECLARE VARIABLES
    struct item *itemPtr = {NULL};

    // FUNCTION CALLS
    readItem(itemPtr);
    printItem(itemPtr);

    return;
}

// READ ITEM FUNCTION
void readItem(struct item *pItem){
    // VARIABLE DECLARATION
    char tempName[SLEN];

    // READ IN PRODUCT'S NAME
    printf("\nPlease enter the product's name: ");
    scanf("%s", tempName);

    // ALLOCATE MEMORY TO STORE THE NAME
    pItem->itemName = (char *) calloc(SLEN, sizeof(char));
    // pItem->itemName = (char *) malloc(strlen(tempName));
    // COPY THE STRING INTO THE itemName STRUCT MEMBER
    strncpy(pItem->itemName, tempName, SLEN);

    // READ IN PRODUCT'S PRICE
}
```

```

printf("\nPlease enter the product's price: ");
scanf("%d", pItem->price);
// READ IN PRODUCT'S QUANTITY
printf("\nPlease enter the product's quantity: ");
scanf("%f", pItem->quantity);

pItem->amount = (float)(pItem->quantity) * (pItem->price);

// FREE THE PREVIOUSLY CALLOC'D MEMORY
free(pItem);
}

// PRINT ITEM FUNCTION
void printItem(struct item *pItem){
    // PRINT OUT THE PRODUCT'S NAME
    printf("\nThe product's name is: %s", pItem->itemName);
    // PRINT OUT THE PRODUCT'S QUANTITY
    printf("\nThe product's quantity is: %i", pItem->quantity);
    // PRINT OUT THE PRODUCT'S PRICE
    printf("\nThe product's price is: %f", pItem->price);
    // PRINT OUT THE AMOUNT
    printf("\nThe amount due is: %f", pItem->amount);
}

```

- Working code after fixing it:

```

/*
    AUTHOR: JULS
    PURPOSE: STRUCTS AND POINTERS
    DATE: MAY 12TH, 2022
*/


#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SLEN 30

// STRUCT DEFINITION
struct item{
    char *itemName;
    int quantity;
    float price;
    float amount; // STORES quantity * price
};

// FUNCTION PROTOTYPES
void readItem(struct item *item);
void printItem(struct item *pItem);

// MAIN FUNCTION
void main(void){
    // DECLARE VARIABLES

```

```

struct item itm;
struct item *itemPtr = {NULL};

itemPtr = &itm;

// ALLOCATE MEMORY
itemPtr->itemName = (char *) calloc(SLEN, sizeof(char));
// CHECK IF MEMORY WAS ALLOCATED CORRECTLY
if(itemPtr == NULL)
    exit(-1);

// FUNCTION CALLS
readItem(itemPtr);
printItem(itemPtr);

// FREE THE PREVIOUSLY CALLOC'D MEMORY
// WE JUST WANNA FREE WHAT WE CALLOC'D (itemName)
free(itemPtr->itemName);

return;
}

// READ ITEM FUNCTION
void readItem(struct item *pItem) {
    // READ IN PRODUCT'S NAME
    printf("\nPlease enter the product's name: ");
    scanf("%s", pItem->itemName);
    // READ IN PRODUCT'S QUANTITY
    printf("\nPlease enter the product's quantity: ");
    scanf("%d", &pItem->quantity);
    // READ IN PRODUCT'S PRICE
    printf("\nPlease enter the product's price: $");
    scanf("%.2f", &pItem->price);
    // CALCULATE THE PRODUCT'S AMOUNT
    pItem->amount = ((float)pItem->quantity) * (pItem->price);
}

// PRINT ITEM FUNCTION
void printItem(struct item *pItem) {
    // PRINT OUT THE PRODUCT'S NAME
    printf("\n\nThe product's name is: %s", pItem->itemName);
    // PRINT OUT THE PRODUCT'S QUANTITY
    printf("\nThe product's quantity is: %i", pItem->quantity);
    // PRINT OUT THE PRODUCT'S PRICE
    printf("\nThe product's price is: $%.2f", pItem->price);
    // PRINT OUT THE AMOUNT
    printf("\nThe amount due is: $%.2f", pItem->amount);
}

```

118. (Demonstration) Structure Pointers and Functions

- **malloc()** goes in the **main function**.
- After you allocate memory you wanna check for **NULL** (if the program couldn't allocate memory):

- ◊ code example:

```
// ... MEMORY ALLOCATION HERE  
  
if (itemPtr == NULL)  
    exit(-1)  
  
...
```

- Whatever is **not a string** needs to be accessed inside the struct with a pointer like this:

- ◊ code example:

```
// READ IN PRODUCT'S PRICE  
printf("\nPlease enter the product's price: ");  
scanf("%d", &pItem->price);
```

- We only wanna **free()** what we **malloc'd()** or **calloc'd()** because the entire structure will be deleted on the stack when the function returns:
 - ◊ we only **malloc'd()** **itemName** so we wanna free that.

Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>

struct item
{
    char *itemName;
    int qty;
    float price;
    float amount;
};

void readItem(struct item *i);
void printItem(struct item *i);

int main()
{
    struct item itm;
    struct item *pItem;

    pItem = &itm;

    pItem->itemName = (char *) malloc(30 * sizeof(char));

    if(pItem == NULL)
        exit(-1);

    // read item
    readItem(pItem);

    // print item
    printItem(pItem);

    free(pItem->itemName);

    return 0;
}

void readItem(struct item *i)
{
    printf("Enter product name: ");
    scanf("%s", i->itemName);

    printf("\nEnter price: ");
    scanf("%f", &i->price);

    printf("\nEnter quantity: ");
    scanf("%d", &i->qty);

    i->amount = (float)i->qty * i->price;
}

void printItem(struct item *i)
{
    /*print item details*/
    printf("\nName: %s", i->itemName);
    printf("\nPrice: %.2f", i->price);
    printf("\nQuantity: %d", i->qty);
    printf("\nTotal Amount: %.2f", i->amount);
}
```

}

Section 14: File Input and Output

119. Overview

- Writing and reading from files.
- We're gonna focus on text files, but we'll talk about binary files as well.

- **Overview:**

- ◊ Up until this point, all data that our program accesses is via memory.
 - Scope and variety of applications you can create is limited.
- ◊ The reason it's stored in memory is because it's really fast.
- ◊ All serious business applications **require more data than would fit into main memory:**
 - also depend on the ability to process data that is persistent and stored on an external device such as disk drive.
- ◊ On embedded devices you don't have as much RAM.
- ◊ C provides many functions in the header file **stdio.h** for **writing to** and **reading from** external devices:
 - the external device you would use for storing and retrieving data is typically a disk drive.
 - however, the library will work with virtually any external storage device:
 - like a **USB** drive, etc...
- ◊ With all the examples up to now, **any data that the user enters is lost once the program ends:**
 - **if the user wants to run the program with the same data, he or she must enter it again each time.**
 - very inconvenient and limits programming.
 - referred to as volatile memory.

- **Files:**

- ◊ Programs need to store data on permanent storage:
 - referred to as **non-volatile** memory because it stays around.
 - continues to be maintained after your computer is turned off.
- ◊ A **file** can store non-volatile data and is usually stored on a disk or a solid-state device:

- it's a named section of storage.
 - **stdio.h** is a file containing useful information.
- ◊ C views a file as a continuous sequence of bytes:
- each byte can be read individually.
 - corresponds to the file structure in the Unix environment.
- ◊ What a file looks like:
-
- The diagram illustrates a file structure as a sequence of bytes. It shows four bytes labeled byte0, byte1, byte2, and byte3. An arrow labeled "Current-2" points to byte2. Below the first byte is an arrow labeled "Beginning of File". An arrow labeled "End of File" points to the byte after byte3. A dashed line extends from byte3 to the right, indicating more bytes. An arrow labeled "Current Position = Beginning + 4" points to the byte immediately after byte3, which is the fifth byte in the sequence.
- a file has **beginning**, an **end** and a **current position** (defined as so many bytes from the beginning).
 - the current position is where any file action (read/write) will take place:
 - you can move the current position to any point in the file (even the end).
 - the **end of file** is usually marked with a special marker called:
 - **EOF**

• Text and Binary Files:

- ◊ There are two ways of writing data to a stream that represents a file:
- **text**
 - **binary**
- ◊ A **text file** is **data** written as a sequence of characters organized as lines (each line ends with a newline).
- ◊ The text data can vary from system to system depending how characters are interpreted.
- ◊ A **binary file** is **data** written as a series of bytes exactly as they appear in memory:
- files like:
 - **image data**
 - **music encoding**

- binary files are not readable because they are just bytes.
- ◊ You can write any data you like to a file:
 - once a file has been written, it just consists of a series of bytes (even if it's represented as text).
- ◊ You have to understand the format of the file in order to read it:
 - a **sequence of 12 bytes** in a binary file **could be 12 characters, 12 8-bit signed integers, 12 8-bit unsigned integers, etc.**
 - in binary mode, each and every byte of the file is accessible.
- ◊ In order to read something from a file you have to understand what's in there as far as data:
 - also how it's stored:
 - this is gonna depend on the system that you're running on
 - and
 - also whether it's text or binary.

• Streams:

- ◊ C programs automatically open three files on your behalf:
 - **standard input**: the normal input device for your system, usually your keyboard.
 - **standard output**: usually your display screen.
 - **standard error**: usually your display screen.
- ◊ A stream can represent a file, but this is a more generic term that can represent:
 - any kind of input.
 - it could represent:
 - **keyboard**
 - **console**
 - **file**
 - **socket** (networking).
- ◊ **Standard input** is the file that is read by **getchar()** and **scanf()** .
- ◊ **Standard output** is used by **putchar()**, **puts()**, and **printf()** .
- ◊ You can also redirect files on the system to be recognized as a **standard input / output**:
 - this is redirection, it's an operating system concept.
 - redirect data, for example:
 - send the output to a file

→ get input from a file as opposed to keyboard.

- ◊ The purpose of the **standard error** output file is to provide a logically distinct place to send error messages.
- ◊ A **stream** is an abstract representation of any external source or destination for data:
 - the **keyboard**, the **command line** on your display, and **files on a disk** are all examples of things you can work with the streams.
 - the C library **provides functions for reading** and **writing to or from data streams**:
 - you use the same **input/output functions for reading** and **writing** any external device that is mapped to a stream.
 - it all depends on where the **stream** is pointing to.
- ◊ Streams are gonna be much more important when you start studying C++.

120. Accessing Files

• Accessing Files:

- ◊ Files on a disk have a name and the rules for naming files are determined by your operating system.
 - you may have to adjust the names depending on what OS your program is running on.
- ◊ A program references a file through a file pointer (or stream pointer, since it works on more than one file):
 - you associate a file pointer with a file programmatically when the program is run.
 - so what you end up doing is creating a **pointer** of type **file** in your program:
 - and it's gonna represent the **file** when you read and write to it.
 - pointers can be reused to point to different files on different occasion.
 - ◊ A **file pointer** points to a **struct** of type **FILE** that represents a stream:
 - it contains information about the file:
 - whether you want to **read**, **write** or **update** the file.
 - the address of the buffer in memory to be used for data.
 - a pointer to the current position in the file for the next operation.
 - the above is all set via **input/output** file operations.
 - ◊ If you want to use several files simultaneously in a program, **you need a separate file pointer for each file**:
 - there is a limit to the number of files you can have open at one time:
 - this is defined as **FOPEN_MAX** in **stdio.h**
 - this depends on the OS you're running on.

• Opening a File:

- ◊ You associate a specific **external file name** with an **internal file pointer** variable through a process referred to as **opening a file**:
 - via the **fopen()** function:
 - it **returns** the **file pointer** for a **specific external file**.
 - you can then use this pointer for reading and writing.
 - ◊ The **fopen()** function is defined in **stdio.h**:
 - ```
FILE *fopen(const char * restrict name, const char * restrict mode);
```

      - all caps **FILE** structure.
      - the **first argument** to the function is a **pointer** to a **string** that is the **name of the external file you want to process**:

→ you can **specify the name explicitly** or **use a char pointer** that **contains the address of the character string** that defines the file name.

→ you can **obtain** the file name **through the command line**, as **input from the user** or **defined as a constant** in your program.

⇒ don't worry about this **restrict** keyword, it's more **C11** stuff.

- the **second argument** to the **fopen()** function is a **character string that represents the file mode**:

→ it specifies what you want to do with the file.

→ a file mode specification is a character string between double quotes.

◊ Assuming the call to **fopen()** is successful, the function returns a pointer of type **FILE\*** that you can use to **reference** the file in **further input/output operations** using other functions in the library.

◊ If the file cannot be opened for some reason, **fopen()** returns **NULL**:

▪ before you start doing any reading or writing make sure that the pointer is not equal to **NULL**.

- another **if** check.

◊ File modes (only applies to text files):

| Mode | Description                                                                                                                                        |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| "w"  | Open a text file for write operations. If the file exists, its current contents are discarded.                                                     |
| "a"  | Open a text file for append operations. All writes are to the end of the file.                                                                     |
| "r"  | Open a text file for read operations.                                                                                                              |
| "w+" | Open a text file for update (reading and writing), first truncating the file to zero length if it exists or creating the file if it does not exist |
| "a+" | Open a text file for update (reading and writing) appending to the end of the existing file, or creating the file if it does not yet exist         |
| "r+" | Open a text file for update (for both reading and writing)                                                                                         |

#### • Write Mode:

◊ If you want to write to an existing text file with the name **myfile.txt**:

- code example:

```
FILE *pFile = NULL;
char *fileName = "myfile.txt";

pFile = fopen(fileName, "w"); // OPEN mymyfile.txt TO WRITE IT

if(pFile == NULL)
 printf("Failed to open %s.\n", fileName);
```

→ the previous code **opens** the **file** and **associates** the file with the name **myFile.txt** with your file pointer **pFile**:

- ⇒ the mode as "**w**" means you can only write to the file.
- ⇒ you cannot read it.

→ if a file with the name **myFile.txt does not exist**, the call to **fopen()** will create a new file with this name.

◊ If you only provide the file name without any path specification, the file is assumed to be in the current directory:

- you can also specify a string that is the full path and name for the file.
  - **relative path**: when you provide no path (the file's in the current directory).
    - this is a better option because it's not dependent in case the file is moved elsewhere.
  - **absolute path**: the file is in another directory.

◊ On opening a file for writing, the file length is **truncated to zero** and the position will be at the beginning of any existing data for the first operation:

- any data that was previously written to the file will be lost and overwritten by any write operations.

- **Append Mode:**

◊ If you want to add to an existing text file rather than overwrite it:

- specify mode "**a**".
- the append mode of operation.

◊ This positions the file at the end of any previously written data:

- if the file does not exist, a new file will be created.

◊ Code example:

```
pFile = fopen("myfile.txt", "a"); // OPEN myFile.txt TO ADD TO IT
```

◊ Don't forget that you should test the return value for **NULL** each time.

```
if(pFile == NULL)
 printf("Failed to open %s.\n", fileName);
```

- ◊ When you open a file in append mode:
  - all write operations will be at the end of the data in the file on each write operation.
  - all write operations append data to the file and you cannot update existing contents in this mode.

- **Read Mode:**

- ◊ If you want to read file:
  - open it with mode argument as "r".
  - you can not write to this file.
- ◊ Code example:

```
pFile = fopen("myFile.txt", "r");
```
- ◊ In the previous code example this positions the file to the beginning of the data.
- ◊ If you are going to just read the file:
  - it must already exist.
    - you're not gonna read an empty file.
- ◊ If you try to open a file for reading that does not exist, **fopen()** will return a file pointer of **NULL**.
- ◊ You always want to check the value returned from **fopen()**.

- **Renaming a file:**

- ◊ Renaming a file is very easy:
  - use the **rename()** function.
- ◊ Function arguments:

```
int rename(const char *oldname, const char *newname);
```
- the integer returned from the function will be:

→ **0** if the name change was successful.

→ **nonzero** otherwise.

◊ The file must not be open when you call **rename()**, otherwise the operation will fail.

- so you could do the following:

```
if(rename("C:\\temp\\myfile.txt", "C:\\temp\\myfile_copy.txt"))
 printf("Failed to rename file.");
else
 printf("File renamed successfully.");
```

→ the backslash is there twice because in C it's a special character, so you have to escape it.

◊ If the file path is incorrect or the file does not exist, the renaming operation will fail.

## • Closing a file:

◊ When you have finished with a file, you need to tell the operating system so that it can free up the file:

- you can do this by calling the **fclose()** function.

◊ **fclose()** accepts a file pointer as an argument:

- it returns **EOF** (int) if an error occurs.
  - **EOF** is a special character called the end-of-file character.
  - defined in **stdio.h** as a negative integer that is usually equivalent to the value **-1**.
- returns **0** if it's successful.

◊ Code example:

```
fclose(pFile); // CLOSE THE FILE ASSOCIATED WITH pFILE
pFile = NULL;
```

◊ The result of calling **fclose()** is that the **connection between the pointer, pFile**, and the **physical file is broken**:

- **pFile** can no longer be used to access the file.

◊ If the file was being written to, the current buffer of the output buffer are written to the file to ensure that data is not lost:

- you don't have any problems with data integrity.

◊ It is good programming practice to close a file as soon as you have finished with it:

- protects against output data loss.

◊ You must also close a file before attempting to:

- **rename** it.

- or

- **remove** it.

- **Deleting a file:**

- ◊ You can delete a file by invoking the **remove()** function:

- declared in **stdio.h**

- ◊ Code example:

- ```
remove ("myfile.txt");
```

- here you don't have to use the file pointer because the file is not open.

- ◊ This will delete the file that has the name **myfile.txt** from the current directory.

- ◊ The file cannot be open when you try to delete it.

- ◊ You should always double check with operations that delete files:

- you could wreck your system if you do not!

121. Reading from a File

- **Reading characters from a text file:**

- The **fgetc()** function reads a character from a text file that has been opened for reading.
- This function **takes a file pointer** as its only argument and **returns the character read** as **type int**.

- Function structure:

```
int mchar = fgetc(pFile);           // READ A CHARACTER INTO mchar WITH pFile A  
FILE POINTER
```

- **mchar** is of type **int** because **EOF** is returned if the end of file has been reached.
 - ⇒ so we can always check that integer to see if it's **EOF**, and if it we don't wanna read anymore.

- The function **getc()**, which is equivalent to **fgetc()**, is also available:
 - it requires an argument of type **FILE*** and returns the character read as type **int**.
 - virtually identical to **fgetc()**.
 - only difference between them is that **getc()** may be implemented as a macro, whereas **fgetc()** is a function.
 - **getc()** has more protection in it.

- **You can read the contents of a file again when necessary:**
 - the **rewind()** function **positions the file** that is specified by the **file pointer argument at the beginning**:

- code example:

```
→ rewind(pFile);
```

- Code example:

```
#include <stdio.h>

int main(){
    FILE *fp = NULL;
    int c;

    fp = fopen("file.txt", "r");

    if(fp == NULL){
        perror("Error in opening file");
        return(-1);
    }

    // READ A SINGLE CHAR
    while((c = fgetc(fp)) != EOF)
        printf("%c", c);
```

```
fclose(fp);
fp = NULL;

return(0);
}
```

- when you usually read characters you're gonna have them inside of a loop.

- **Reading strings from a text file:**

- ◊ You can use the **fgets()** function to read from any file or stream.

- ◊ Function structure:

- ```
char *fgets(char *str, int nchars, FILE *stream);
```

- ◊ The function reads a **string** into the memory area pointed to by **str**, from the file specified by stream:

- characters are read until either a '\n' is read or **nchars - 1** characters have been read from the stream, whichever occurs first.

- if a newline character is read, it's retained in the string:
    - a '\0' character will be appended to the end of the string.
  - if there is no error, **fgets()** returns the pointer, **str**.
  - if there is an error, **NULL** is returned.
  - reading **EOF** causes **NULL** to be returned.

- ◊ Code example:

```
#include <stdio.h>

int main(void){
 FILE *fp;
 char str[60];

 // OPENING FILE FOR READING
 fp = fopen("file.txt", "r");

 if(fp == NULL){
 perror("Error opening file.");
 return(-1);
 }
```

```

if(fgets (str, 60, fp) != NULL) {
 // WRITING CONTENT TO stdout
 printf("%s", str);
}

fclose(fp);
fp = NULL;

return(0);
}

```

- **Reading formatted input from a file:**

- ◊ You can get formatted input from a file by using the standard **fscanf()** function.
- ◊ Function structure:
 

```
int fscanf(FILE *stream, const char *format, ... a variable
number of arguments (variables) goes here);
```

  - the **format** is just like **format specifiers**.
- ◊ Formatted input is:
  - data that has a delimiter or has specific format associated with it.
- ◊ The **first argument** to this function is the **pointer** to a **FILE** object that **identifies the stream**.
- ◊ The **second argument** to this function is the **format**:
  - **a C string that contains one or more of the following items:**
    - **whitespace character**.
    - **non-whitespace character**.
    - **format specifiers**.
  - usage is similar to **scanf()**, but, from a file.
- ◊ The function **returns the number of input items successfully matched and assigned**.
- ◊ Code example:
 

```
// AUTHOR: JFITECH
// PURPOSE: READING FROM A FILE - FORMATTED STRINGS
// DATE: MAY 17TH, 2022

#include <stdio.h>
#include <stdlib.h>
```

```
#include <stddef.h>

int main(void) {
 char str1[10], str2[10], str3[10];
 int year;
 FILE *fp = NULL;

 fp = fopen("file.txt", "w+");
 if(fp != NULL)
 fputs("Hello, how are you?", fp);

 rewind(fp);

 fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);

 printf("Read string1 |%s|\n", str1);
 printf("Read string2 |%s|\n", str2);
 printf("Read string3 |%s|\n", str3);
 printf("Read integer |%d|\n", year);

 fclose(fp);
 fp = NULL;

 return 0;
}
```

- we're opening it up as **reading** and **writing**, but we're actually not gonna write anything to it.

## 122. Writing to a File

- **Writing characters to a text file:**

- ◊ The simplest way write operation is provided by the function **fputc()**
  - writes a single character to a text file.
- ◊ Function structure:
  - ```
int fputc(int ch, FILE *pfile);
```
- ◊ The function writes the character specified by the first argument to the file identified by the second argument (file pointer):
 - it returns the character that was written if successful.
 - returns **EOF** if there was a failure.
- ◊ In practice, characters are not written to a file one by one:
 - this is extremely inefficient.
- ◊ The **putc()** function is equivalent to **fputc()**:
 - it requires the same arguments and the return type is the same.
 - the difference between them is that **putc()** may be implemented in the standard library as a macro, whereas **fputc()** is a function.
- ◊ Code example:
 - ```
#include <stdio.h>

int main(void) {
 FILE *fp = NULL;
 int c;

 // FIRST THING YOU HAVE TO DO IS OPEN UP THE FILE
 fp = fopen("file.txt", "w+");

 for(ch = 33; ch <= 100; ch++)
 fputc(ch, fp)

 fclose(fp);
 fp = NULL;

 return(0);
}
```

- **Writing a string to a text file:**

- ◊ You can use the **fputs()** function to write to any file or stream.

◊ Function structure:

```
int fputs(const char *str, FILE *pfile);
```

- the **first argument** is a **pointer to the character string** that is **to be written to the file**.
- the **second argument** is the **file pointer**.

◊ This function will write characters from a string until it reaches a '\0' character:

- it does not write the **null terminator** character to the file:
  - so you have to write the **null terminator** explicitly using the **fputsc()** function.
  - it can complicate reading back variable-length strings from a file that have been written by **fputs()**
  - expecting to write a line of text that has a newline character at the end.

◊ Code example:

```
#include <stdio.h>

int main(void){
 FILE *filePointer = NULL;

 filePointer = fopen("file.txt", "w+");

 fputs("This is a C course.", filePointer);
 fputs("I am happy to be here.", filePointer);

 fclose(filePointer);
 filePointer = NULL;

 return(0);
}
```

• Writing formatted output to a file:

◊ The standard function for formatted output to a stream is **fprintf()**

◊ Function structure:

```
int fprintf(FILE *stream, const char *format, ... a variable
 number of arguments (variables) goes here);
```

- the **first argument** to the function is the **pointer** to a **FILE** object that **identifies the stream**.

- the **second argument** to the function is the **format**:

- a C string that contains one or more of the following items:
  - ⇒ **whitespace character**.
  - ⇒ **non-whitespace character**.

⇒ **format specifiers.**

→ usage is similar to **printf()**, but, to a file.

- ◊ If successful, the total number of character written is returned otherwise, a negative number is returned.

◊ Code example:

```
#include <stdio.h>
#include <stlib.h>

int main(void) {
 FILE *fp = NULL;

 fp = fopen("file.txt", "w+");
 if(fp != NULL)
 fprintf(fp, "%s %s %s %s %d", "Hello", "my", "number", "is", 555);

 fclose(fp);
 fp = NULL;

 return(0);
}
```

# **123. Finding your position in a File**

- **File Positioning:**

- ◊ Some functions that will allow you to change the position of a file.
- ◊ For many applications, you need to be able to access data in a file other than sequential order.
  - sequential order is starting at the beginning and going byte by byte towards the end.
- ◊ There are various functions that you can use to access data in random sequence.
- ◊ **There are two aspects to file positioning:**
  - **finding out where you are in a file.**
  - **moving to a given point in a file.**
- ◊ You can access a file at a random position regardless of whether you opened the file.

- **Finding out where you are:**

- ◊ You have two functions to tell you where you are in a file:
  - **f.tell()**
  - **f.getpos()**

- ◊ **f.tell()**

- `long ftell(FILE *pfile)`

- this function accepts a **file pointer as an argument** and **returns a long integer value** that **specifies the current position in the file**.

- you can use it like this:

- `long fpos = ftell(pfile);`

- the **fpos** variable now **holds the current position in the file** and you can use this to return to this position at any subsequent time:

- the value is the offset in bytes from the beginning of the file.

- code example:

```
// AUTHOR: JFITECH
// PURPOSE: FINDING POSITION IN A FILE - ftell()
// DATE: MAY 17TH, 2022
```

```

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

int main(void) {
 FILE *fp = NULL;
 int len = 0;

 fp = fopen("file.txt", "r");
 if(fp == NULL) {
 perror("Error opening file.");
 return(-1);
 }

 // THIS GOES TO THE END OF THE FILE
 fseek(fp, 0, SEEK_END);

 len = ftell(fp);
 fclose(fp);

 printf("\nTotal size of file.txt = %d bytes\n", len);

 return(0);
}

```

## ◊ **fgetpos()**

- the second function providing information on the current file position is a little more complicated.

- function structure:

- `int fgetpos(FILE *pfile, fpos_t *position);`

- the **first parameter** is a **file pointer**.

- the **second parameter** is a **pointer to a type that is defined in stdio.h**

- ⇒ **fpos\_t** : a type that is able to record every position within in a file.

- the **fgetpos()** function is designed **to be used with** the positioning function **fsetpost()**

- the **fgetpos()** function **stores the current position** and **file state information** for the file in position and **returns 0** if the **operation is successful**:

- returns a nonzero integer value for failure.

- code example:

- `fpos_t here;`  
`fgetpos(pfile, &here);`

- this code records the current file position in the variable **here**.

- you must **declare a variable** of type **fpos\_t**.

- ⇒ cannot declare a pointer of type **fpos\_t\*** because there will not be any memory

allocated to store the position data.

- code example:

```
FILE *fp = NULL;
fpos_t position;

fp = fopen("file.txt", "w+");
fgetpos(fp, &position);
fputs("Hello, World!", fp);

fclose(fp)
fp = NULL;
```

- **Setting a position in a file:**

- ◊ As a complement to **fseek()**, you have the **fseek()** function.

- ◊ Function structure:

- `int fseek(FILE *pfile, long offset, int origin);`

- the **first argument** is a **pointer to the file you are repositioning**.
      - the **second and third parameters** define where you want to go in the file:
        - **second parameter** is an offset from a reference point specified by the third parameter.
        - **reference point** can be one of three values that are specified by the predefined names:

- ⇒ **SEEK\_SET**: defines the beginning of the file.
      - ⇒ **SEEK\_CUR**: defines the current position in the file.
      - ⇒ **SEEK\_END**: defines the end of the file.

- ◊ For a text mode file, the **second argument must be a value returned by **fseek()****

- ◊ The third argument for text mode files must be **SEEK\_SET**:

- for **text files**, all operations with **fseek()** are performed **with reference to the beginning of the file**.

- for binary files, the offset argument is simply a relative byte count:
        - can therefore supply positive or negative values for the offset when the reference point is specified as **SEEK\_CUR** .

- ◊ Code example:

- `#include <stdio.h>`

```

int main(void) {
 FILE *fp = NULL;

 fp = fopen("file.txt", "w+");
 fputs("This is Juls", fp);

 fseek(fp, 7, SEEK_SET);
 fputs("Hello, how are you", fp);

 fclose(fp);
 fp = NULL;

 return(0);
}

```

- ◊ You have the **fsetpos()** function to go with **fgetpos()**
- ◊ Function structure:
  - `int fsetpos(FILE *pfile, const fpos_t *position);`
    - the **first argument** is a **pointer to the open file**.
    - the **second argument** is a pointer of the **fpos\_t** type:
      - the position that is stored at the address was obtained by calling **fgetpos()**
- ◊ Code example:
  - `fsetpos(pfile, &here);`
    - the variable here was previously set by a call to **fgetpos()**
- ◊ The **fsetpos()** returns a nonzero value on error or 0 when it succeeds.
- ◊ This function is designed to work with a value that is returned by **fgetpos()**:
  - you can only use it to get to a place in a file that you have been before.
  - **fseek()** allows you to go to any position just by specifying the appropriate offset.

- ◊ Code example:

```

#include <stdio.h>

int main()
{
 FILE *fp = NULL;
 fpos_t position;

 fp = fopen("file.txt", "w+");

 fgetpos(fp, &position);
 fputs("Hello, World!", fp);
}

```

```
fsetpos(fp, &position);
fputs("This is going to override previous content", fp);

fclose(fp);
fp = NULL;

return(0);
}
```

## **124. (Challenge) Find the number of lines in a file**

- Requirements:

- Write a program to find the **total number of lines in a text file**.
- Create a **text file** that contains some lines of text.
- Open** your test file.
  - fopen()** call
    - remember to close your file.
- Use the **fgetcf()** function to parse characters in a file until you get the **EOF**:
  - if **EOF** → '\n' increment counter.
  - create a local variable for the counter.
- Display as output the total number of lines in the file.

# My Code

```
// AUTHOR: JFITECH
// PURPOSE: CHALLENGE 19TH - FIND THE AMOUNT OF LINES IN A FILE
// DATE: MAY 17TH, 2022

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

void main(void){
 FILE *pFile = NULL;
 int counter = 0, character = 0;

 // OPEN FILE
 pFile = fopen("myFile.txt", "r");

 // CHECK IF THE POINTER IS NULL
 if(pFile == NULL){
 perror("Error in opening file.");
 exit(-1);
 }

 // READ EVERY CHAR UNTIL IT HITS THE LINE FEED CHARACTER
 while((character = fgetc(pFile)) == '\n')
 counter++;

 // CLOSE THE FILE
 fclose(pFile);
 pFile = NULL;

 printf("\nThe amount of lines in the file is: %d", counter);

 return;
}
```

- Working code after fixing it:

```
// AUTHOR: JFITECH
// PURPOSE: CHALLENGE 19TH - FIND THE AMOUNT OF LINES IN A FILE
// DATE: MAY 17TH, 2022

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

void main(void){
 FILE *pFile = NULL;
 int counter = 0;
 char character;

 // OPEN FILE
```

```
pFile = fopen("myFile.txt", "r");

// CHECK IF THE POINTER IS NULL
if(pFile == NULL){
 perror("Error in opening file.");
 exit(-1);
}

// READ EVERY CHAR UNTIL IT HITS THE LINE FEED CHARACTER
while((character = fgetc(pFile)) != EOF) {
 if(character == '\n')
 counter++;
}

// CLOSE THE FILE
fclose(pFile);
pFile = NULL;

printf("\nThe amount of lines in the file is: %d\n", ++counter);

return;
}
```

## **125. (*Demonstration*) Find the number of lines in a file**

•

# Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>

#define FILENAME "Test.txt"

int main()
{
 FILE *fp = NULL;
 char ch;
 int linesCount = 0;

 fp = fopen(FILENAME, "r");

 if(fp == NULL)
 {
 printf("File does not exist\n");
 return -1;
 }

 while((ch=fgetc(fp)) != EOF)
 {
 if(ch=='\n')
 linesCount++;
 }

 fclose(fp);
 fp = NULL;

 printf("Total number of lines are: %d\n", ++linesCount);

 return 0;
}
```

## **126. (Challenge) Convert characters in a file to uppercase**

- Requirements:

Write a program that **converts all characters of a file to uppercase** and **write the results out to a temporary file**.

then **rename the temporary file** to the **original filename** and **remove the temporary filename**.

You need to use the following functions:

- fgetc()**
- fputc()**
- rename()**
- remove()**
- islower()**

can convert to uppercase by subtracting 32 from it.

character = character - 32;

then do an **fputc()** after converting.

Remember to close your files.

Display the contents of the original file to standard output in UPPERCASE.

read the file.

Read  
 Write  
 Read

# My Code

```
// AUTHOR: JFITECH
// PURPOSE: CHALLENGE 20TH - CONVERT CHARACTER TO ALL UPPERCASE
// DATE: MAY 17TH, 2022

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

void main(void){
 FILE *pFile = NULL, *pTempFile = NULL;
 char c; // EACH CHARACTER

 // OPEN, READ AND WRITE FILES
 pFile = fopen("myFile.txt", "r");
 pTempFile = fopen("tempFile.txt", "w+");

 // CHECK IF THE EITHER POINTER IS NULL
 if(pFile == NULL || pTempFile == NULL) {
 perror("Error in opening file.");
 exit(-1);
 }

 // READ EVERY CHAR IN THE ORIGINAL FILE
 while((c = fgetc(pFile)) != EOF) {
 if(islower(c = fgetc(pFile))) {
 // CONVERT EACH CHARACTER TO UPPERCASE
 c = c - 32;
 // COPY EVERY CHARACTER TO THE TEMP FILE
 fputc(c, pTempFile);
 }
 }

 // RENAME THE TEMPORARY FILE
 rename("myFile.txt", "tempFile.txt");

 // READ THE CONTENTS OF THE FILE
 printf("The contents of the file are: %c", pTempFile);

 // CLOSE THE FILES
 fclose(pFile);
 fclose(pTempFile);
 pFile = NULL;
 pTempFile = NULL;

 return;
}
```

- (Working) code after fixing it:



```

// AUTHOR: JFITECH
// PURPOSE: CHALLENGE 20TH - CONVERT CHARACTER TO ALL UPPERCASE
// DATE: MAY 17TH, 2022

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

void main(void){
 FILE *pFile = NULL, *pTempFile = NULL;
 char c; // EACH CHARACTER

 // OPEN, READ AND WRITE FILES
 pFile = fopen("myFile.txt", "r");
 pTempFile = fopen("tempFile.txt", "w");

 // CHECK IF EITHER POINTER IS NULL
 if(pFile == NULL || pTempFile == NULL) {
 perror("Error in opening file.");
 exit(-1);
 }

 // READ EVERY CHAR IN THE ORIGINAL FILE
 while((c = fgetc(pFile)) != EOF) {
 if(islower(c)) {
 // CONVERT EACH CHARACTER TO UPPERCASE
 c = c - 32;
 }
 // COPY EVERY CHARACTER TO THE TEMP FILE
 fputc(0, pTempFile);
 }

 // CLOSE THE FILES
 fclose(pFile);
 fclose(pTempFile);

 // RENAME THE TEMPORARY FILE
 rename("tempFile.txt", "myFile.txt");

 // REMOVE THE TEMPORARY FILE
 remove("tempFile.txt");

 // OPEN THE RENAMED FILE TO READ
 pFile = fopen("myFile.txt", "r");

 // CHECK IF THE POINTER IS NULL
 if(pFile == NULL) {
 perror("Error in opening file.");
 exit(-1);
 }

 // ITERATE THROUGH EACH CHARACTER AND PRINT IT
 while((c = fgetc(pFile)) != EOF)
 // READ THE CONTENTS OF THE FILE
 printf("%c", c);

 // CLOSE THE LAST OPENED FILE
 fclose(pFile);
}

```

```
pFile = NULL;
pTempFile = NULL;

return;
}
```

## **127. (*Demonstration*) Convert characters in a file to uppercase**

•

# Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>

#define FILENAME "Test.txt"

int main()
{
 FILE *fp = NULL;
 FILE *fp1 = NULL;

 char ch = ' ';

 fp = fopen(FILENAME, "r");

 if (fp == NULL)
 return -1;

 // create a temp file
 fp1 = fopen("temp.txt", "w");

 if (fp1 == NULL)
 return -1;

 while ((ch = fgetc(fp)) != EOF)
 {
 if (islower(ch))
 {
 ch = ch-32;
 }

 fputc(ch, fp1);
 }

 fclose(fp);
 fclose(fp1);

 remove(FILENAME);

 // rename temp file to Test.txt file
 rename("temp.txt", FILENAME);

 // remove the temp file
 remove("temp.txt");

 fp = fopen(FILENAME, "r");

 if (fp == NULL)
 return -1;

 while ((ch = fgetc(fp)) != EOF)
 printf("%c", ch);

 fclose(fp);
 fp = NULL;
 fp1 = NULL;
```

```
 return 0;
```

```
}
```

## **128. (Challenge) Print the contents of a file in reverse order**

- In this challenge we'll become familiar with changing the position of a file.

- **Requirements:**

Use the **fseek()** and **ftell()** to **change the position** so you can **point to the middle of the file**.

Write a program that **prints the contents of a file in reverse order**.

Use the **fseek()** function to **seek the end of the file**

Use the **ftell()** function to **get the position of the file pointer**.

Display as **output** the **file in reverse order**.

Create the file beforehand.

Open the file as read only.

Before you start reading the file, seek to the end of it using **fseek()**

once you use **fseek()**, use **ftell()** to get actual position

you can then use the position to start printing out all of the characters using

**fgetc()**

# My Code

```
// AUTHOR: JFITECH
// PURPOSE: CHALLENGE 21st - PRINT CONTENTS OF FILE IN REVERSE ORDER
// DATE: MAY 18TH, 2022

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

void main(void){
 FILE *pFile = NULL;
 int c = 0;

 // OPEN THE FILE
 pFile = fopen("myFile.txt", "r");

 // CHECK IF THE FILE POINTER IS NULL
 if(pFile == NULL){
 perror("Error opening file.");
 exit(-1);
 }

 // MOVE THE POSITION TO THE END OF THE FILE
 fseek(pFile, 0, SEEK_END);

 // TO KNOW WHERE WE ARE IN THE FILE
 long fPos = ftell(pFile);

 // READ THE FILE'S CHARACTERS
 while((c = fgetc(pFile)) != EOF)
 c--;
 printf("%c", c);

 // CLOSE THE FILE
 fclose(pFile);
 pFile = NULL;

 return;
}
```

- Working code after fixing it:

```
// AUTHOR: JFITECH
// PURPOSE: CHALLENGE 21st - PRINT CONTENTS OF FILE IN REVERSE ORDER
// DATE: MAY 18TH, 2022

#include <ctype.h>
#include <stdio.h>
```

```

#include <stdlib.h>
#include <stddef.h>

void main(void) {
 FILE *pFile = NULL;
 int c = 0, i = 0;

 // OPEN THE FILE
 pFile = fopen("myFile.txt", "r");

 // CHECK IF THE FILE POINTER IS NULL
 if(pFile == NULL) {
 perror("Error opening file.");
 exit(-1);
 }

 // MOVE THE POSITION TO THE END OF THE FILE
 fseek(pFile, 0, SEEK_END);

 // WILL TELL US THE AMOUNT OF BYTES IN THE FILE
 long fPos = ftell(pFile);

 // READ THE FILE'S CHARACTERS
 // while((c = fgetc(pFile)) != EOF)
 // c--;
 // printf("%c", c);

 //
 //while(i < fPos) {
 // i++;
 // fseek(pFile, -i, SEEK_END); // SEEK SO MANY BYTES FROM THE
END, WHICH -i FROM THE END
 printf("%c", fgetc(pFile)); // READ IN THE CHARACTER AND
PRINT IT OUT
 }

 printf("\n");

 // CLOSE THE FILE
 fclose(pFile);
 pFile = NULL;

 return;
}

```

## **129. (*Demonstration*) Print the contents of a file in reverse order**

- We're reading the file from the end to the beginning using our position functions.

# Instructor's Code

```
#include <stdio.h>
#include <stdlib.h>

#define FILENAME "Test.txt"

int main()
{
 FILE *fp = NULL;

 int cnt = 0;
 int i = 0;

 fp = fopen(FILENAME, "r");

 if (fp == NULL)
 return -1;

 // moves the file pointer to the end of the file
 fseek(fp, 0, SEEK_END);

 // get the position of the file pointer
 cnt = ftell(fp);

 while (i < cnt)
 {
 i++;
 fseek(fp, -i, SEEK_END);
 printf("%c", fgetc(fp));
 }

 printf("\n");
 fclose(fp);
 fp = NULL;

 return 0;
}
```

## ***Section 15: The Standard C Library***

# 130. Standard Header Files

- We've been using a lot of the functions in the C standard library.
- The C standard library is basically all of the functionality that comes with the C programming language.
- Code that you can reuse.

- **<stddef.h>**

- ◊ Contains some standard definitions:

| Define                                           | Meaning                                                                                                                                    |
|--------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| NULL                                             | A null pointer constant                                                                                                                    |
| offsetof ( <i>structure</i> ,<br><i>member</i> ) | The offset in bytes of the member <i>member</i> from the start of the structure <i>structure</i> ; the type of the result is <i>size_t</i> |
| <i>ptrdiff_t</i>                                 | The type of integer produced by subtracting two pointers                                                                                   |
| <i>size_t</i>                                    | The type of integer produced by the <i>sizeof</i> operator                                                                                 |
| <i>wchar_t</i>                                   | The type of the integer required to hold a wide character (see Appendix A, "C Language Summary")                                           |

- ◊

- ◊ It has some constants and functions.

- **<limits.h>**

- ◊ Contains various implementation-defined limits for character and integer data types.

- ◊

| Define     | Meaning                                                                                       |
|------------|-----------------------------------------------------------------------------------------------|
| CHAR_BIT   | Number of bits in a char (8)                                                                  |
| CHAR_MAX   | Maximum value for object of type char (127 if sign extension is done on chars, 255 otherwise) |
| CHAR_MIN   | Minimum value for object of type char (-127 if sign extension is done on chars, 0 otherwise)  |
| SCHAR_MAX  | Maximum value for object of type signed char (127)                                            |
| SCHAR_MIN  | Minimum value for object of type signed char (-127)                                           |
| UCHAR_MAX  | Maximum value for object of type unsigned char (255)                                          |
| SHRT_MAX   | Maximum value for object of type short int (32767)                                            |
| SHRT_MIN   | Minimum value for object of type short int (-32767)                                           |
| USHRT_MAX  | Maximum value for object of type unsigned short int (65535)                                   |
| INT_MAX    | Maximum value for object of type int (32767)                                                  |
| INT_MIN    | Minimum value for object of type int (-32767)                                                 |
| UINT_MAX   | Maximum value for object of type unsigned int (65535)                                         |
| LONG_MAX   | Maximum value for object of type long int (2147483647)                                        |
| LONG_MIN   | Minimum value for object of type long int (-2147483647)                                       |
| ULONG_MAX  | Maximum value for object of type unsigned long int (4294967295)                               |
| LLONG_MAX  | Maximum value for object of type long long int (9223372036854775807)                          |
| LLONG_MIN  | Minimum value for object of type long long int (-9223372036854775807)                         |
| ULLONG_MAX | Maximum value for object of type unsigned long long int (18446744073709551615)                |

- ◊ These are useful for when you're running on different systems.

- **<stdbool.h>**

- ◊ Contains definitions for working with Boolean variables (type **\_Bool**)

| Define | Meaning                                                    |
|--------|------------------------------------------------------------|
| bool   | Substitute name for the basic <code>_Bool</code> data type |
| true   | Defined as 1                                               |
| false  | Defined as 0                                               |

# 131. Various Functions

- **String functions:**

- ◊ To use these string functions, you must include the file:
  - **<string.h>**
- ◊ You can initialize a character array (a string constant) inside double quotes:
  - but if you later wanna change it you'd have to use **strncpy()** function.

- **size\_t strlen (s)**
  - returns the number of characters in s, excluding the null character

- ◊
  - returns **size\_t** which is the **integer** representing the length.

- **char \*strrchr (s, c)**
  - searches the string s for the last occurrence of the character c. If found, a pointer to the character in s is returned; otherwise, the null pointer is returned

- **char \*strtok (s1, s2)**
  - breaks the string s1 into tokens based on delimiter characters in s2

- **Character functions:**

- ◊ To use these character functions, you must include the file:
  - **<ctype.h>**

- ◊
  - **isalnum, isalpha, isblank, iscntrl, isdigit, isgraph, islower, isspace, ispunct, isupper, isxdigit**

- **I/O functions:**

- ◊
  - included in this file are declarations for the I/O functions and definitions for the names EOF, NULL, stdin, stdout, stderr (all constant values), and FILE

- **stdin:** standard input
- **stdout:** standard output
- **stderr:** standard error

- int fclose (filePtr)
  - closes the file identified by filePtr and returns zero if the close is successful, or returns EOF if an error occurs

- int feof (filePtr)
  - returns nonzero if the identified file has reached the end of the file and returns zero otherwise

- int feof (filePtr)
  - returns nonzero if the identified file has reached the end of the file and returns zero otherwise

- you're getting rid of anything inside of a buffer.

- long ftell (filePtr)
  - returns the relative offset in bytes of the current position in the file identified by filePtr, or -1L on error

- int remove (fileName)
  - removes the specified file. A nonzero value is returned on failure

- int rename (fileName1, fileName2)
  - renames the file fileName1 to fileName2, returning a nonzero result on failure.

- **Conversion functions:**

- To use these functions that convert character strings to numbers, you must include the file:
  - <stdlib.h>

- double atof (s)
  - converts the string pointed to by s into a floating-point number, returning the result

- int atoi (s)
  - converts the string pointed to by s into an int, returning the result

- int atol (s)
  - converts the string pointed to by s into a long int, returning the result

- **Dynamic Memory functions:**

- ◊ To use these functions that allocate and free memory dynamically, you must include the file:
  - **<stdlib.h>**

- ◊
  - void \*calloc (n, size)
    - allocates contiguous space for n items of data, where each item is size bytes in length. The allocated space is initially set to all zeroes. On success, a pointer to the allocated space is returned; on failure, the null pointer is returned

- ◊
  - void \*malloc (size)
    - allocates contiguous space of size bytes, returning a pointer to the beginning of the allocated block if successful, and the null pointer otherwise

- ◊
  - void \*realloc (pointer, size)
    - changes the size of a previously allocated block to size bytes, returning a pointer to the new block (which might have moved), or a null pointer if an error occurs

## 132. Math Functions

- **Math functions:**

- ◊ To use common math functions you must include the **math.h** header file and link to the math library.
  - the linking usually happens for you when using an IDE like Code::Blocks.

- ◊
  - **double acosh (x)**

- ◊
  - returns the hyperbolic arccosine of x,  $x \geq 1$



- ◊
  - **double asin (x)**

- ◊
  - returns the arcsine of x as an angle expressed in radians in the range  $[-\pi/2, \pi/2]$ . x is in the range  $[-1, 1]$



- ◊
  - **double atan (x)**

- ◊
  - returns the arctangent of x as an angle expressed in radians in the range  $[-\pi/2, \pi/2]$



- ◊
  - **double ceil (x)**

- ◊
  - returns the smallest integer value greater than or equal to x. Note that the value is returned as a double



- ◊
  - **double ceil (x)**

- ◊
  - returns the smallest integer value greater than or equal to x. Note that the value is returned as a double



- ◊
  - **double floor (x)**

- ◊
  - returns the largest integer value less than or equal to x. Note that the value is returned as a double



- ◊
  - **double log (x)**

- ◊
  - returns the natural logarithm of x,  $x \geq 0$



- ◊
  - **double nan (s)**

- ◊
  - returns a NaN, if possible, according to the content specified by the string pointed to by s



- ◊
  - **NaN: Not A Number**



- double pow (x, y)
  - returns  $xy$ . If x is less than zero, y must be an integer. If x is equal to zero, y must be greater than zero

◊

- double remainder (x, y)
  - returns the remainder of x divided by y

- ◊ □ you could use this as opposed to the **mod** operator.

◊

- double round (x)
  - returns the value of x rounded to the nearest integer in floating-point format. Halfway values are always rounded away from zero (so 0.5 always rounds to 1.0)

◊

- double sin (r)
  - returns the sine of r

◊

- double sqrt (x)
  - returns the square root of x,  $x \geq 0$

◊

- double tan (r)
  - returns the tangent of r

◊ and so many more...

- there's actually a complex arithmetic library that you can use for even more complex arithmetic.

# 133. Utility Functions

- **Utility functions:**

- ◊ Some of the utility functions that are provided by the C standard library.
- ◊ To use these include the `<stdlib.h>` file.

- **int abs (n)**
- returns the absolute value of its int argument n



- `void exit (n)`
- terminates program execution, closing any open files and returning the exit status specified by its int argument n
- `EXIT_SUCCESS` and `EXIT_FAILURE`, defined in `<stdlib.h>`
- other related routines in the library that you might want to refer to are `abort` and `atexit`



- a **negative number** means there's an error.
- a **positive number** or 0 means no error.
- you can use the constants stated above if you don't wanna use numbers.



- `char *getenv (s)`
- returns a pointer to the value of the environment variable pointed to by s, or a null pointer if the variable doesn't exist
- used to get environment variables



- it's like a global variable that you can use anywhere on the OS.



- `void qsort (arr, n, size, comp_fn)`
- sorts the data array pointed to by the void pointer arr
- there are n elements in the array, each size bytes in length. n and size are of type `size_t`
- the fourth argument is of type "pointer to function that returns int and that takes two void pointers as arguments."
- `qsort` calls this function whenever it needs to compare two elements in the array, passing it pointers to the elements to compare



- `int rand (void)`
- returns a random number in the range [0, `RAND_MAX`], where `RAND_MAX` is defined in `<stdlib.h>` and has a minimum value of 32767
- `void srand (seed)`
- seeds the random number generator to the unsigned int value seed



- `int system (s)`
- gives the command contained in the character array pointed to by s to the system for execution, returning a system-defined value
- `system ("mkdir /usr/tmp/data");`

- **Assert library:**

- ◊ The assert library, supported by the **assert.h** header file, is a small one designed to **help with debugging programs**.
- ◊ it consists of a macro named **assert()**:
  - it takes as its argument an integer expression.
  - if the **expression evaluates as false (nonzero)**, the **assert()** macro writes an error message to the standard error stream (**stderr**) and calls the **abort()** function, which terminates the program.

```
z = x * x - y * y; /* should be + */
assert(z >= 0); // asserts that z is greater than or equal to 0
```

- if the previous code returns **false** your program will exit, otherwise it'll execute the next line of code.
  - ⇒ it's like making true assumptions throughout your code.

- **Other useful header files:**

- ◊
  - **time.h**
    - defines macros and functions supporting operations with dates and times
  - returns a **time\_t struct**
- ◊
  - **errno.h**
    - **defines macros for the reporting of errors**
  - often used for better debugging and reporting specific errors.

- ◊
  - **locale.h**
    - defines functions and macros to assist with formatting data such as monetary units for different countries

- ◊
  - **signal.h**
    - defines facilities for dealing with conditions that arise during program execution, including error conditions
  - useful for program termination, generating different signals.
  - this could be useful when calling abort, to specifying specific signals when the program crashed or terminated.
  - if you understand signals you can use them in your core file as well.

- stdarg.h

- defines facilities that enable a variable number of arguments to be passed to a function

- **var args**

## ***Section 16: Conclusion***

# **134. Further Topics of Study**

- We're wrapping up the class.
- So we're gonna take a look at further topics of study.
- There are a lot of advanced topics that you can further study to become awesome at C programming.

- **Further topics of study:**

- ◊ More on **data types**:

- defining your own data types (**typedef**).

- ◊ More on the **preprocessor**:

- string concatenation.
    - the **preprocessor** runs before the compiler.
    - you can use it to optimize a lot of your code.

- ◊ More on **void\***'s:

- ◊ Static libraries and shared objects:

- you can create your own libraries.

- ◊ Macros:

- this gets run by the preprocessor to mainly do substitution:
        - string substitution.

- ◊ Unions:

- these go along with structs.
        - it's a way of having multiple types of data returned.

- ◊ Function pointers:

- you can have pointers that point to various functions.

- ◊ Advanced pointers.

- ◊ Variable arguments to functions (variadic functions).

- ◊ Dynamic linking (**dlopen**):

- when you open up a library at runtime.
        - this increases performance.
          - makes executable programs smaller.

- ◊ Signals, forking and inter-process communication:
  - **inter-process communication** is when you have two programs that wanna communicate.
- ◊ Threading and concurrency:
  - allows multiple execution points inside your program.
- ◊ Sockets.
- ◊ **setjmp** and **longjmp** for restoring state.
- ◊ More on memory management and fragmentation:
  - how to manage your memory and put in the **heap** or the **stack**.
    - to make your program a little more efficient.
      - **fragmentation**:
        - ⇒ when you have memory in different locations, it's not sequential.
  - ◊ More on making your program portable.
  - ◊ Interfacing with kernel modules (**drivers** and **ioctls**):
    - the kernel works with all sorts of calls to lower level.
    - this is for more low-level embedded stuff.
    - we could create drivers to talk with hardware directly.
  - ◊ More on compiler and linker flags.
  - ◊ Advanced uses of gdb.
  - ◊ Profiling and tracing tools (**gprof**, **dtrace**, **strace**):
    - static analyzers: tools that can analyze your code before you run your program to see if there's any dead code or if there's any bad programming practices.
  - ◊ Memory debugging tools such as **valgrind**:
    - this helps when you have like race conditions or you have memory corruption problems.
    - a lot of times when you're managing memory you have memory leaks or buffer overflows, it's very hard to find those.