Programming for Mechatronics - MECHTRON 2MP3

Pedram Pasandide (pasandip@mcmaster.ca)

GitHub

McMaster University.

Fall 2023

Contents

1 Introduction

1.1 A Short Overview of Programming Languages

Programming languages can be categorized based on their level of abstraction, which refers to how closely they resemble the underlying hardware operations of a computer. At the lowest level, we have **machine code**, which consists of binary instructions that are directly executed by the computer's hardware. Above machine code, we have **assembly language**, which uses **human-readable** characters to represent the low-level instructions. However, assembly language is specific to each CPU architecture, so it can differ between different types of processors.

Moving up the hierarchy, we encounter **C**, which was created in around 1970 as a by- product of UNIX based operating systems. **C** is considered a slightly higher-level language compared to **assembly language**. It provides a set of abstract statements and constructs that are closer to human-readable text. **C** code can be compiled using a program called **compiler**, which translates the **C** code into **machine code** that can be executed on any CPU. This portability is one of the reasons why C is often referred to as "**portable assembly**." **C** allows programmers to efficiently write code with a relatively low level of abstraction.

Above **C**, we find **C++**, which was created around 1985. **C++** is an extension of **C** and introduces **object-oriented programming** concepts. It includes the ability to define **classes**, which encapsulate data and behavior into reusable structures. However, for the purpose of this course, we won't delve into the specifics of object-oriented programming.

Moving further up the ladder, we have **Java** and **C#**, which are considered **mid-level** languages. These languages restrict certain low-level operations available in **C** and **C++**, for example, managing memory environments. Instead of allowing direct memory allocation, **Java** and **C#** handle memory management themselves, offering automatic memory allocation and garbage collection. This trade-off provides programmers with increased security and simplifies memory management, but it also limits some of the flexibility and control offered by lower-level languages.

At the **highest level**, we have interpreted languages like **Python** and **JavaScript**. These languages are considered highly abstracted and provide significant levels of convenience and ease of use for developers. Interpreted languages do not require a separate compilation step; instead, they use an **interpreter** to execute the source code directly. The interpreter reads the code **line-by-line**, executing each instruction as it encounters it. This line-by-line execution allows for more dynamic and interactive programming experiences but can result in slower performance compared to compiled languages.

1.1.1 C, Strengths and Weaknesses

Weaknesses

While C has many strengths, it also has a few weaknesses that developers should be aware of:

1. Error-prone: Due to its flexibility, C programs can be prone to errors that may not be easily detectable by the compiler. For example, missing a semicolon or adding an extra one can lead to unexpected behavior, such as infinite loops! It means you are waiting for hours to get the result, then you figure out it shouldn't take that much time! Don't worry there are some ways to avoid it!

2. Difficulty in Understanding: C can be more difficult to understand compared to higher-level languages. It requires a solid understanding of low-level concepts, such as memory management and pointers. The syntax and usage of certain features, like pointers and complex memory operations, may be unfamiliar to beginners or programmers coming from higher-level languages. This learning curve can make it more challenging for individuals new to programming to grasp and write code in C effectively.

3. Limited Modularity: C lacks built-in features for modular programming, making it harder to divide large programs into smaller, more manageable pieces. Other languages often provide mechanisms like namespaces, modules, or classes that facilitate the organization and separation of code into logical components. Without these features, developers must rely on manual techniques, such as using header files and carefully organizing code structure, to achieve modularity in C programs. This can lead to codebases that are harder to maintain and modify as the project grows in complexity.

Strength

C programming language possesses several strengths that have contributed to its enduring popularity and wide-ranging applicability:

1. Efficiency: C is renowned for its efficiency, allowing programs written in C to execute quickly and consume minimal memory resources. It provides low-level access to memory and hardware, enabling developers to optimize their code for performance-critical applications.

2. Power: C offers a rich set of data types and a flexible syntax, enabling developers to express complex algorithms and manipulate data efficiently. Its extensive standard library provides numerous functions for tasks such as file I/O, memory management, and string manipulation, allowing programmers to accomplish a lot with concise and readable code.

3. Flexibility: The versatility of C is evident in its widespread use across different domains and industries. C has been employed in diverse applications, including embedded systems, oper-

ating systems, game development, scientific research, commercial data processing, and more. Its flexibility makes it a suitable choice for a broad range of programming tasks.

4. Portability: C is highly portable, meaning that programs written in C can be compiled and run on a wide range of computer systems, from personal computers to supercomputers. The C language itself is platform-independent, and compilers are available for various operating systems and architectures.

5. UNIX Integration: C has deep integration with the UNIX operating system, which includes Linux. This integration allows C programs to interact seamlessly with the underlying system, making it a favored language for system-level programming and development on UNIX-based platforms.

1.2 Understanding the Anatomy of a Computer System

In the digital age, computers are ubiquitous and indispensable tools that power our modern world. From the smartphones in our pockets to the massive data centers that drive the internet, computers come in all shapes and sizes. Yet, despite their diversity, they all share a common foundation—the intricate interplay of components that make up the heart of every computing machine.

In this section, we embark on a journey to explore the inner workings of a computer. We will unravel the complex web of connections between the key components that allow a computer to process information, store data, and enable the myriad tasks we rely on every day. Whether you're a tech enthusiast, a curious learner, or simply someone seeking a deeper understanding of the devices that have become an integral part of our lives, this exploration will provide you with valuable insights.

From the Central Processing Unit (CPU) that serves as the computer's brain to the memory and storage that hold and retrieve data, we'll examine each component's role and importance in the grand orchestration of computing. We'll also delve into the role of cache memory, the bridge between high-speed CPU operations and the relatively slower main memory. And, of course, we'll touch upon the vital input and output devices that enable us to interact with these electronic marvels.

By the end of this section, you'll have a clearer picture of the intricate dance that occurs within the heart of your computer, demystifying the technology that surrounds us daily. So, let's embark on this journey together and peer into the fascinating world of computer components and connections.

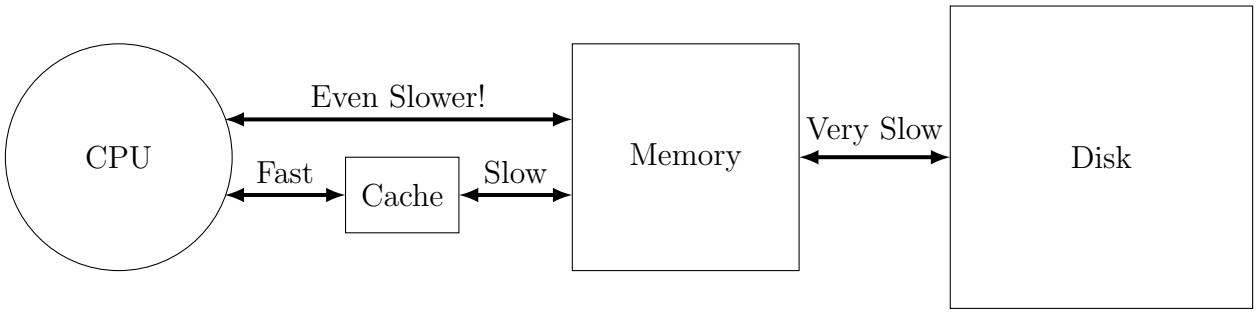
In a computer system, these components are interconnected as follows:

- CPU (Central Processing Unit): The CPU is the brain of the computer, responsible for

executing instructions. It communicates with other components through a system bus.

- Memory (RAM - Random Access Memory): RAM is a type of volatile memory that stores data and program code that the CPU is actively using. The CPU reads and writes data to/from RAM during its operations.
- Storage (Hard Drive, SSD, etc.): Storage devices like hard drives or solid-state drives (SSDs) are used for long-term data storage. The CPU can read data from storage devices and write data to them.
- Cache (CPU Cache): Cache is a smaller, faster memory located on the CPU itself or very close to it. It stores frequently used data and instructions to speed up CPU operations.
- Input Devices (Keyboard, Mouse, etc.): Input devices allow users to interact with the computer. Data from input devices is sent to the CPU for processing.
- Output Devices (Monitor, Speakers, etc.): Output devices display or produce the results of CPU operations. The CPU sends data to output devices for presentation to the user.

The following figure shows a simple way to describe a computer architecture:



The CPU acts as the central hub, orchestrating the flow of data between memory, storage, cache, input devices, and output devices as needed to perform tasks and run programs. Data and instructions move between these components through various buses and pathways within the computer system, but these connections are typically abstracted from the user and managed by the computer's hardware and operating system.

1.3 Shell Basics on Linux

To create a new folder (directory) in Linux using the terminal, you can use the `mkdir` command. Here's how you can do it:

Open your terminal application. This can vary depending on the Linux distribution you're using. You can typically find the terminal in Applications menu or by pressing `Ctrl+Alt+T`.

Navigate to the location where you want to create the new folder. You can use the `cd` command to change directories. For example, if you want to create the folder in your home directory, you can use `cd ~` to navigate there. Once you're in the desired location, use the `mkdir` command followed by the name you want to give to the new folder. For example, to create a folder called "MECHTRON2MP3," you would type `mkdir MECHTRON2MP3`. Press `Enter` to execute the command.

In the world of Linux commands hold the power to shape the digital landscape. Let's do start with some basic commands:

1. Current directory: To know your current directory, use the command `pwd`. It will display the path of the directory you are currently in. To view all the files and folders in the current directory, use the command `ls`. This will provide a list of all the files and folders. Look for a folder named `MECHTRON2MP3` in the list of files and folders (if you have did the previous step successfully!). Once you find it, you'll use the following steps to navigate and work within that folder. To open the `MECHTRON2MP3` folder in the file manager, use the command `open MECHTRON2MP3`.

2. Moving back and forth in directories: To change your current directory to the `MECHTRON2MP3` folder, use the command `cd MECHTRON2MP3` in the terminal. This command allows you to move into the specified folder (use `pwd` to double check!) If you want to go back to the initial directory, simply use the command `cd` without any arguments. This will take you back to the `/home/username` directory, where `username` is what you picked during installation of Linux. To go back one folder, use the command `cd ..`. This will navigate you up one level in the directory structure.

3. Making a file: Now, go back to `MECHTRON2MP3` directory. If you want to create a text file named "AnneMarie", use the command `nano AnneMarie.txt`. This will open the file editor where you can write and edit text. To open the `AnneMarie.txt` file in the Nano text editor, use the command `nano AnneMarie.txt` again. This allows you to make changes to the file. If you wish to save your changes, press `Ctrl+O` and then press `Enter`. This will save the file. Alternatively, if you want to save and exit the Nano text editor, press `Ctrl+X` and then press `Enter`. This will save the changes and return you to the terminal.

To reopen the `AnneMarie.txt` file in the terminal using the Nano text editor, use the command `nano AnneMarie.txt`. If you prefer to open the `AnneMarie.txt` file in the Windows environment, use the command open `AnneMarie.txt` after saving the file. This will open the file using the default application associated with `.txt` files.

4. Copy and Paste: To copy the `AnneMarie.txt` file and paste it into another directory, you can use the `cp` command, like:

```
cp AnneMarie.txt /path/to/destination/directory/
```

Replace `/path/to/destination/directory/`. with the actual path of the directory where you want to paste the file.

5. View a file: The `cat` command is used to display the contents of a file in the terminal. It can be helpful when you want to quickly view the contents of files. Here's an `.txt` file example: `cat AnneMarie.txt`. Running this command will print the contents of `AnneMarie.txt` in the terminal window. Easier way just double click on the file!

6. Finding a folder of file: To find a file or folder using the `locate` command or similar commands, you need to have the appropriate indexing database set up on your system. The `locate` command searches this database for file and folder names. For example, `locate AnneMarie.txt`, will search the indexing database for any files or folders with the name `AnneMarie.txt` and display their paths if found. In this case, nothing is shown in terminal. Since we just made this file, the database including directories is not updated to include this file. To update the directory database in all memory, we have to use the command `updatedb`, and you will probably get the following message:

`/var/lib/plocate/: Permission denied`, indicating that you do not have the necessary permissions to access the directory. To solve the issue and update the directories index successfully, use the `sudo updatedb` which prompts you to a password which you picked. Enter the password, it might not be shown when you are entering the password, then press `Enter`. By this command, you are running the `updatedb` with superuser (root) privileges. `sudo` stands for "Super User Do" and is a command that allows regular users to execute commands with the security privileges of the superuser. Now you can use command `locate`, but this time it shows the directory that the file is saved.

Alternatively, you can use:

```
find /path/to/search/directory -name AnneMarie.txt
```

Replace `/path/to/search/directory` with the directory where you want to start the search.

Tips! Debugging is an essential aspect of programming. No programmer possesses complete knowledge or can remember every command, especially when working with multiple programming languages. However, it is crucial to know where to find the answers. Here are two approaches to tackle a specific issue:

1. Use internet. For instance, search for "permission denied on Linux." Take a quick look at the top search results, paying particular attention to reputable sources such as Stack Overflow. Spend a few minutes scrolling through the search results.

2. Engage with ChatGPT by asking a specific question related to the problem you encountered. For example, you could ask, "I tried `updatedb` on Linux terminal, and it gives me Permission denied. How can I solve it?"

7. Remove: The `rm` command is used to remove (delete) files and directories. It's important to exercise caution when using this command, as deleted files cannot be easily recovered. Here's an example to remove the `AnneMarie.txt` file: `rm AnneMarie.txt`. This command will permanently delete the `AnneMarie.txt` file. Be sure to double-check the file name and verify that you want to delete it.

The `rmdir` command is used to remove (delete) empty directories. It cannot remove directories that have any files or subdirectories within them. Here's an example: `rmdir empty_directory`. Replace `empty_directory` with the name of the directory you want to remove. This command will only work if the directory is empty. If there are any files or subdirectories inside it, you'll need to delete them first or use the `rm` command with appropriate options to remove them recursively.

8. Let's checkout some OS characteristics.

Linux distribution: The `lsb_release -a` command provides comprehensive information about your Linux distribution, including the release version, codename, distributor ID, and other relevant details. The `-a` option is a command-line option or flag that stands for "all." When used with the `lsb_release` command, it instructs the command to display all available information about the Linux distribution. It is a convenient way to quickly check the specifics of your Linux distribution from the command line. My Linux distribution is:

```
Distributor ID: Ubuntu
Description: Ubuntu 22.04.2 LTS
Release: 22.04
Codename: jammy
```

CPU: The `lscpu` command is used to gather and display information about the CPU (Central Processing Unit) and its architecture on a Linux system. It provides detailed information about

the processor, including its model, architecture, number of cores, clock speed, cache sizes, and other relevant details. **Some** of details for my system:

```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  8
  On-line CPU(s) list:   0-7
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
  CPU family:            6
  Model:                 158
  Thread(s) per core:    2
```

Disk space available: The `df -h` command is used to display information about the disk space usage on your Linux system. In this command, `df` stands for **disk free**, and `-h` is a command-line option or flag that stands for **human-readable**. The

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda2       228G  113G  104G  53% /
```

`/dev/sda2` is a partition identifier that represents the second partition on the first SATA (or SCSI) disk (sda). It typically contains the main file system of your Linux system. The actual partition identifiers may differ depending on your system's configuration and the number of disks or partitions present.

Information about the RAM: In the command `free -h`, `free` represents the command used to display memory usage statistics, and `-h` is a command-line option that stands for **human-readable**.

```
               total        used        free      shared  buff/cache   available
Mem:            15Gi       4.4Gi       3.3Gi       637Mi       7.8Gi        10Gi
Swap:          2.0Gi          0B       2.0Gi
```

The **Swap** space is a portion of the hard drive that is used as virtual memory by the operating system. It acts as an extension to the physical memory (RAM) and allows the system to temporarily store data that doesn't fit into the RAM.

9. Short keys in terminal: `Ctrl+L` to remove the history of terminal. Pressing Up Arrow Key will show you the previous commands. To copy from terminal you have to press `Cntrl + Shift + C` and to paste a command press `Ctrl + shift + V`.

2 Fundamentals of C Language

This section is designed to provide an introduction to fundamental concepts in C, including data types, variables, and control flow statements such as if and loops. Some of students with prior knowledge of programming might already know these concepts, but we will delve into each topic extensively and provide detailed explanations. Our goal is to ensure that everyone, regardless of their programming background, can grasp these essential concepts effectively. We will illustrate each concept with practical examples and problem-solving exercises to enhance understanding, as these concepts form the backbone of C programming.

2.1 Writing a Simple Program, `Hello, McMaster!`

2.1.1 A General Form of a Program

Let's make a new file using `nano Hello.c` where the **.c** extension represent a C file. Before closing the the opened file in terminal copy and paste the following code in the file and save it. To paste the the copied section you have to use `Ctrl + shift + V`. Press `Ctrl + X` then `y` and press Enter to save before exit.

```c
// this code is written by Pedram
#include <stdio.h>

// the main function
int main(void) {
 /* calling "printf" function
 the "defination" is included in "stdio" library */
 printf("Hello, McMaster!\n");
}
```

Open the file by `open Hello.c`. This is a more user friendly environment to edit the code. Maybe the main difference is I can click different parts of the code and edit the code. During you lab lectures this week, you TAs will discuss Visual Studio Code IDE (Integrated Development Environment) which support many programming languages, including C, as well as many useful tools such as automatic code formatting. I strongly suggest to install it and set it up for C format for the next lectures. If I want to modify the code in terms of extra spaces in the code using `open Hello.c` it will take me some time. But if you have Visual Studio Code installed, in the terminal enter `code Hello.c` to open the same code in Visual Studio Code (VSC).Right click on the code and choose **Format Document** which automatically modifies the code in C format.

Tips! Get your hands dirty! I promise you that by only reading these notes you will NOT be able to learn programming. You have to write, modify, and change every code in these notes. Play with them like you play video games!

Anything starts with two slashes, `//`, until the end of the line is a comment and it will not be executed. To make multiple lines as comments, you can place them between `/*` and `*/`. The following lines in the code are all comments:

```
// this code is written by Pedram

// the main function

/* calling "printf" function
the "defination" is included in "stdio" library */
```

Based on the IDE that you are using the colour of the comments might be different. I have said earlier that understanding C codes especially written by someone else might be difficult. Comments are supposed to be helpful to remember or tell others what is the purpose of each part of the code or even how it works.

So the first line which is really executed is `include <stdio.h>`. Usually at the beginning of a code, we include libraries used in the code. `include <stdio.h>` tells C to read **header file** named `stdio` with extension `.h` which stands for **header file**. In this library the definition of function `printf` is mentioned. Any time you forget what library a function is in, you can ask your best friend **Google** or **ChatGPT** by simply asking "what library in C printf is in."

The second line that will be executed is `int main(void)` as a function. In C, defining a function starts with the type of value the function returns, which in this case it is `int`. The `int` means **integer** and the function `main` will automatically return the value **0** if there is no error in the code.

The `main` is the name of function. This function must be included in all C programs, and you have to write your code inside this function.

Between the parentheses, `void` what inputs your program needs and requires the user to give the input(s) every time the code is going to be executed. In this case, we don't have any input so it is `void`, or we can define `int main()` instead of `int main(void)`. Both means the program requires no input(s).

Anything between `{`, right after `int main(void)`, and `}`, at the end of the code, is called **compound statement** or **code block**, which belongs to `main` function. So the following state-

ment is the general format that we have to include in all C programs:

```
Inclduing Library

int main(){

your statements

   }
```

In this code, `printf("Hello, McMaster!\n")` is the only statement we have and `printf` function is used to print a text. Between `(` and `)` the arguments for the function `printf` is given, which is a string or text to be printed!

A string is sequence of characters enclosed within quotation marks. `"Hello, McMaster!\n"` is the string that will be printed. The backslash `\` before `n` tells C to go to (**n**)ext line after printing `Hello, McMaster!`. Try to compile and run the code two times with `\n` and two times without `\n` and see the difference in the terminal! At the end of this line there is `;` indicating the command for this line has ended.

A reminder, the closing bracket `}` at the end, means the end of function `main`!

2.1.2 Compiling and Executing the Program

Now we have to convert the program, the file `Hello.c`, to a format that machine can understand and execute. It usually involves, **Pre-processing**, **Compiling**, and **Linking**. During **Pre-processing** some modification will be done automatically to the code, before making the executable **object** code in **Compiling** step. These two steps are done at the same time and automatically, so won't need to be worried about it! In the Linking step, the library included will be linked to the executable file. Since this code is using a standard library, `<stdio.h>`, it happens automatically. Again, you don't need to be worried!!!

To compile the program in UNIX based OS, usually `cc` is used. Open a terminal in the directory where `Hello.c` file is located (please refer to the section Shell Basics on Linux). Type `cc Hello.c`. Check out the directory. A new executable **object** file named `a.out` by default will be created.

If you see the following error after executing `cc Hello.c`, it means you are not in the same directory where `Hello.c` is located. A reminder, use `pwd` to check your current directory.

```
cc1: fatal error: Hello.c: No such file or directory
compilation terminated.
```

Visual Studio Code! You may open the code in Visual Studio Code (VSCode) by running `code Hello.c` in the terminal. Again make sure current directory is where `Hello.c` is located, otherwise you will see an error indicating the is no such a file in this directory. At the top of VSCode opened, press **View** and click **Terminal**. A window will be opened at the bottom, named **TERMINAL**. There is no difference between this Terminal and terminal you open using `Ctrl + Alt + T`. Make sure the directory of this Terminal in VScode is the where `Hello.c` is located to avoid the error I have mention before running `cc Hello.c` in the terminal.
After `cc Hello.c` if there is no problem, you should be able to see a new executable **object** file named `a.out` by default in the directory.

At this stage the code is compiled, the **object** `a.out` readable by machine is created. This means the program is translated to a language that machine can understand and it saved in the **object** file `a.out`. Now it is time to tell the machine to run the translated code and show us the result. Run `./a.out` in the terminal. Again, make sure you are in the directory where `a.out` is located. It is done! you should be able to see the result!

Hello, McMaster!

I have mentioned the name `a.out` is given to the **object** file by default. I can define any name I want. Lets remove the previous object file by `rm a.out` and make a new one with the name "**Brad**" by using `cc -o Brad Hello.c`. The compiler `cc` has many options, and `-o` means translate the program `Hello.c` to the machine language with an (**o**)bject file named `Brad`. Now you must be able to see an **object** file named `Brad` in the same directory. If you run the command line `./Brad` in the same directory, you should be able to see the output.

GCC compiler Another popular compiler in C is GCC (GNU Compiler Collection) compiler supported by Unix OS. It is known for its robustness, efficiency, and compatibility with multiple platforms. GCC supports various optimizations and provides comprehensive error checking, making it a reliable choice for compiling C code.
One notable feature of GCC is its similarity to the "cc" compiler command. This similarity stems from the fact that on many Unix-like systems, the "cc" command is often a symbolic link or an alias for GCC. Therefore, using "cc" to compile your code essentially invokes the GCC compiler with its default settings. GCC offers numerous options to control various aspects of the compilation process, such as optimization levels, debugging symbols, and specific target architectures.
You can compile the same program using GCC instead of `cc` by executing `gcc -o Brad Hello.c` in the terminal.

The **object** file `./Brad` is executable in any Unix based OS. It is software of application you developed!

2.2 Integer Data Type

In contemporary C compilers, there is support for a range of integer sizes spanning from 8 to 64 bits. However, it is worth noting that the specific names assigned to each size of integer may differ among different compilers, leading to potential confusion among developers. First we need to know how data is stored in computers.

2.2.1 Binary Representation

In the world of computers, information is stored and processed as sequences of bits, representing either a 0 or a 1. This fundamental concept forms the basis of how data is handled by computer systems.

Consider the scenario where we have a fixed number of bits, denoted by N, to represent our numbers. Let's take **N=4** to save an `int` number where `int` in C stands for **integer** number. In this case, we are allocating nine bits to express our numerical values. If the integer is `unsigned` then:

- $0 = 0000$
- $1 = 0001$
- $2 = 0010$
- $3 = 0011$
- $4 = 0100$
- $5 = 0101$
- $6 = 0110$
- $7 = 0111$
- $8 = 1000$
- $9 = 1001$
- $10 = 1010$
- $11 = 1011$
- $12 = 1100$
- $13 = 1101$
- $14 = 1110$
- $15 = 1111 = 2^N - 1$

More example?! If N = 3 then for `unsigned` integer we would have:

- 0 = 000
- 1 = 001
- 2 = 010
- 3 = 011
- 4 = 100
- 5 = 101
- 6 = 110
- 7 = 111 = $2^N - 1$

which means the range of numbers can be saved as `unsigned` integer in C is from 0 to $2^N - 1$. For `signed` integers, the negative numbers can be obtained by inverting the bits then adding one to the result. This method is called the two's complement operation.

- **-8 = 1000** = -2^{N-1}
- -7 = (inverting 0111, we have 1000, +1 is:)1001
- -6 = (inverting 0110, we have 1001, +1 is:)1010
- -5 = (inverting 0101, we have 1010, +1 is:)1011
- -4 = (inverting 0100, we have 1011, +1 is:)1100
- -3 = (inverting 0011, we have 1100, +1 is:)1101
- -2 = (inverting 0010, we have 1101, +1 is:)1110
- -1 = (inverting 0001, we have 1110, +1 is:)1111
- 0 = 0000
- 1 = 0001
- 2 = 0010
- 3 = 0011
- 4 = 0100
- 5 = 0101
- 6 = 0110
- 7 = 0111 = $2^{N-1} - 1$

If you have noticed, you can see the 1000 is signed to -8. In total with 4 bits, machine can have a combination of 2^4 zero and ones. It can be seen that all positive numbers starts with 0, and negative ones start with one. Therefore, it is accepted to sign 1000 bit sequence to the lowest number -8.

More example?! If N = 3 then for `unsigned` integer we would have:

- **-4 = 100** $= -2^{N-1}$
- -3 = (inverting 011, we have 100, +1 is:)101
- -2 = (inverting 010, we have 101, +1 is:)110
- -1 = (inverting 001, we have 110, +1 is:)111
- 0 = 000
- 1 = 001
- 2 = 010
- 3 = 011 $= 2^{N-1} - 1$

which means the range of numbers can be saved as `signed` integer in C is from -2^{N-1} to $2^{N-1} - 1$. Exceeding this range can cause errors, probably unseen, which it is called **integer overflow**.

2.2.2 Using `printf` and `limits.h` to Get the Limits of Integers

Let's see some of these limits using `limits.h` library. Make a new file using `nano limit.c` which `limit` is the name of program given by you, and `.c` is the extension which stands for C files. Copy and past the following code. Press `Ctrl + X` then `y` and Enter to save and close the file. Open the code in VScode, if you have it as IDE, by entering `code limit.c` in your terminal. Make sure you are in the same directory where you made this file.

```c
// This code is written by ChatGPT
#include <stdio.h>
#include <limits.h>

int main() {

 printf("Size of char: %zu bits\n", 8 * sizeof(char));
 printf("Signed char range: %d to %d\n", SCHAR_MIN, SCHAR_MAX);
 printf("Unsigned char range: %u to %u\n", 0, UCHAR_MAX);
 printf("\n");

 printf("Size of int: %zu bits\n", 8 * sizeof(int));
 printf("Signed int range: %d to %d\n", INT_MIN, INT_MAX);
 printf("Unsigned int range: %u to %u\n", 0, UINT_MAX);
 printf("\n");
```

```
    printf("Size of short: %zu bits\n", 8 * sizeof(short));
    printf("Signed short range: %d to %d\n", SHRT_MIN, SHRT_MAX);
    printf("Unsigned short range: %u to %u\n", 0, USHRT_MAX);
    printf("\n");

    printf("Size of long: %zu bits\n", 8 * sizeof(long));
    printf("Signed long range: %ld to %ld\n", LONG_MIN, LONG_MAX);
    printf("Unsigned long range: %u to %lu\n", 0, ULONG_MAX); //
       change %u to lu
    printf("\n");

    printf("Size of long long: %zu bits\n", 8 * sizeof(long long));
    printf("Signed long long range: %lld to %lld\n", LLONG_MIN,
       LLONG_MAX);
    printf("Unsigned long long range: %u to %llu\n", 0, ULLONG_MAX);
        // change %u to llu
}
```

Let's check the code line-by-line.

1. `#include <limits.h>`: This is a preprocessor directive that includes the header file `limits.h` in the C program. The `limits.h` header provides constants and limits for various data types in the C language, such as the minimum and maximum values that can be represented by different types.

2. Placeholders: is a special character or sequence of characters that is used within a formatted string to represent a value that will be substituted during runtime. Placeholders are typically used in functions like `printf` or `sprintf` to dynamically insert values into a formatted output. When the program runs, the placeholders are replaced with the actual values passed as arguments to the formatting function. The values are appropriately converted to match the format specifier specified by the corresponding placeholders.

Placeholders provide a flexible way to format output by allowing dynamic insertion of values. They help in producing formatted and readable output based on the specified format specifiers and the provided values.

- `%zu`: This is a placeholder used with `printf` to print the value of an `unsigned` **integer**.
- `%d`: This is a placeholder used to `printf` the value of a `signed integer`.
- `%u`: This is a placeholder used to `printf` the value of an `unsigned` **integer**.

- `%ld`: This is a placeholder used to `printf` the value of a `signed long` **integer**.
- `%lu`: This is a placeholder used to `printf` the value of an `unsigned long` **integer**.
- `%lld`: This is a placeholder used to `printf` the value of a `signed long long` **integer**.
- `%llu`: This is a placeholder used to `printf` the value of an `unsigned long long` **integer**.

3. `sizeof()`: This is an operator in C that returns the size of a variable or a data type in bytes. In the given code, `sizeof()` is used to determine the size of different data types (e.g., char, int, long, long long). The result of `sizeof()` is then multiplied by 8 to obtain the size in bits.

4. `printf("\n")`: This line of code is using `printf` to print a newline character `\n`. It adds a line break, resulting in a new line being displayed in the console output.

5. `NAME_MIN` and `NAME_MAX`: The code refers to variables like `SCHAR_MIN`, `SCHAR_MAX`, `INT_MIN`, `INT_MAX`, etc. These variables are predefined in the C library, specifically in the limits.h header file. They represent the minimum and maximum values that can be stored in the respective data types (e.g., `char`, `int`, `short`, `long`, `long long`).

6. `char`, `int`, `short`, `long`, and l`long long`: These are data types in the C language. They represent different ranges of integer values that can be stored. The code provided displays the size and range of each of these data types, both signed and unsigned.

This time we compiler the code with more options `gcc -Wall -W -std=c99 -o limit limit.c`, where:

`-Wall`: This flag enables a set of warning options, known as "**all warnings**." It instructs the compiler to enable a comprehensive set of warning messages during the compilation process. These warnings help identify potential issues in the code, such as uninitialized variables, unused variables, type mismatches, and other common programming mistakes. By enabling `-Wall`, you can ensure that a wide range of warnings is reported, assisting in the production of cleaner and more reliable code.

`-W`: This flag is used to enable additional warning options beyond those covered by `-Wall`. It allows you to specify specific warning options individually. Without any specific options following `-W`, it enables a set of commonly used warnings similar to `-Wall`. By using `-W`, you have more control over the warning messages generated by the compiler.

`-std=c99`: This flag sets the C language standard that the compiler should adhere to. In this case, c99 indicates the **C99 standard**. The **C99 standard** refers to the ISO/IEC 9899:1999 standard for the C programming language. It introduces several new features and improvements compared to earlier versions of the C standard, such as support for variable declarations anywhere

in a block, support for `//` single-line comments, and new data types like `long long`. By specifying `-std=c99`, you ensure that the compiler follows the **C99 standard** while compiling your code.

`-o limit`: This flag is used to specify the output file name. In this case, it sets the output file name as "limit". The compiled binary or executable will be named "limit" as a result.

`limit.c`: This is the source file that contains the C code to be compiled.

After compiling the code, I can run the object file `limit` in the directory by `./limit`. This is the result I get in **my computer**!

```
Size of char: 8 bits
Signed char range: -128 to 127
Unsigned char range: 0 to 255

Size of int: 32 bits
Signed int range: -2147483648 to 2147483647
Unsigned int range: 0 to 4294967295

Size of short: 16 bits
Signed short range: -32768 to 32767
Unsigned short range: 0 to 65535

Size of long: 64 bits
Signed long range: -9223372036854775808 to 9223372036854775807
Unsigned long range: 0 to 18446744073709551615

Size of long long: 64 bits
Signed long long range: -9223372036854775808 to 9223372036854775807
Unsigned long long range: 0 to 18446744073709551615
```

We will find out about the importance of knowing limits in section A Simple Example for Integer Overflows.

2.2.3 Declaring and Naming with and without Initializing

To declare a variable you can simply:

```
int Pedram;
```

where `int` is the type of variable and `Pedram` is the name of variable. You can after this line

of code calculate the value of `Pedram`. You may also declare the value for this variable when it is initialized:

```
int Pedram = 10;
```

Tips! There are some reserved names that you cannot use for your variables, including: `auto`, `break`, `case`, `char`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `inline`, `int`, `long`, `register`, `restrict`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile`, `while`, `_Bool`, `_Complex`, `_Imaginary`, `#define`, `#include`, `#undef`, `#ifdef`, `#endif`, `#ifndef`, `#if`, `#else`, `#elif`, `#pragma`, and more! Don't worry if you use these you will see some errors and warnings when compiling the the code!

More Tips! About naming style, if you use only characters like `a,b,c, ..., z`, no one can follow your code or what is the purpose of this variable. So:

- Use meaningful and descriptive names that convey the purpose or nature of the variable. for example, `rectangle_height` and `triangle_width`
- Avoid excessively long names, but provide enough clarity to understand the purpose of the variable.
- Follow a consistent naming convention, such as `camelCase` or `snake_case`.
- use comments if necessary to explain the purpose or usage of a variable.

If you are compiling the code you can use `-Wextra` flag for uninitialized variables. Let's try the following code with and without.

```
#include <stdio.h>

int main() {
 int x;
 int y = x + 5;  // Using uninitialized variable x
 printf("%d\n", y);
}
```

Compiling the code with `gcc -o Pedram Pedram.c`, then executing the program with `./Pedram`. I get the following result:

```
-1240674203
```

If I execute the code one more time, `./Pedram`, without even compiling the code, I get:

```
233918565
```

What is going on???? When you run this code, you may get different output each time because the value of x is unspecified and can contain any arbitrary value. The variable x could be storing whatever value was previously in that memory location, and performing calculations with such a value can lead to unexpected results.

Let's compile the code but this time with `gcc -Wextra -o Pedram Pedram.c`. In my terminal it tells me `Using uninitialized variable x` and mentioning the line of code this issue is happening. At this point, the source code `Pedram.c` is compiled and the Pedram object is available. It means I can execute the program, but I am aware of the fact that this will result a wrong and an unexpected result. There are some warning you have to take serious even more than error! I could see the same warning by compiling the code using `gcc -Wall -W -std=c99 -o Pedram Pedram.c`

2.2.4 Constant Variables

Constant variables are declared using the const keyword, indicating that their value cannot be modified once assigned. Here's an example:

```
#include <stdio.h>

int main() {
 const int MAX_VALUE = 100;

 printf("Max value: %d\n", MAX_VALUE);;
}
```

Start developing this habit to mention the constant values during programming which make you code to be more understandable. After using `const` for variable `MAX_VALUE`, I cannot change the value for this variable. You can try to do it and you will see errors like:

`expression must be a modifiable lvalue`

or

`assignment of read-only variable 'MAX_VALUE'`

2.2.5 Arithmetic Operations on Integers

Let's play with some of these integer numbers using arithmetic operations.

```c
#include <stdio.h>

int main() {
 int a = 5;
 int b = 3;
 int sum = a + b;
 int difference = a - b;
 int product = a * b;
 int quotient = a / b;
 int remainder = a % b;

 printf("Sum: %d\n", sum);
 printf("Difference: %d\n", difference);
 printf("Product: %d\n", product);
 printf("Quotient: %d\n", quotient);
 printf("Remainder: %d\n", remainder);

 return 0;
}
```

after compiling the code and executing the object file, you should get the following results:

```
Sum: 8
Difference: 2
Product: 15
Quotient: 1
Remainder: 2
```

2.2.6 A Simple Example for Integer Overflows

Why we tried to understand the limits of integer variables? If you define an `int` value equal to 2,147,483,647 + 1, you will encounter a phenomenon known as **integer overflow**. In C, when an arithmetic operation results in a value that exceeds the maximum representable value for a given integer type, the behavior is undefined.

In most cases, when an integer overflow occurs, the value will "wrap around" and behave as if it has rolled over to the minimum representable value for that integer type. In the case of a 32-bit int, which has a maximum value of 2,147,483,647, adding 1 to it will result in an **integer overflow**.

The exact behavior after the overflow is undefined, meaning it's not guaranteed what will happen. However, it is common for the value to wrap around to the minimum value for the int data type, which is typically -2,147,483,648 for a 32-bit signed integer. Let's try an example:

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

int main() {
 int value = INT_MAX + 1;
 printf("Value: %d\n", value);

 return 0;
}
```

Compile the code using `gcc -o Anna Anna.c` where `Anna.c` is the source code, and `Anna` is the object file. Run the code with `./Anna`. The result I get in **my machine** is:

```
Value: -2147483648
```

while I was expecting to see 2,147,483,648.

Let's see another example. Make a new file using terminal by `nano Pedram.c`. Copy and paste the following code using `Ctrl + Shift + V`. Press `Ctrl + X`, then press `y` and Enter to save the changes made to new file named `Pedram` with extension `.c`. In the following code the limits for each type is given in comments, which these limits are based on **my machine** and it might be different in yours. We found these limits by running the code in section Using `printf` and `limits.h` to Get the Limits of Integers.

```
#include <stdio.h>

int main() {

 // ------------------------- char -------------------------
 char Ped_RealChar = 'P';
 // char range: -128 to 127
 char Ped_NumChar = 80;
 // unsigned char range: 0 to 255
 unsigned char Ped_NumChar_unsigned = 252;
 // ------------------------- short -------------------------
 // short range: -32768 to 32767
 short Ped_sh = -1234;
 // unsigned short range: 0 to 65535
```

```
    unsigned short Ped_sh_unsigned = 56789;
    //-------------------------- int --------------------------
    // int range: -2147483648 to 2147483647
    int Ped_int = -42;
    // unsigned int range: 0 to 4294967295
    unsigned int Ped_int_unsigned = 123456;
    // ------------------------- long -------------------------
    // long range: -9223372036854775808 to 9223372036854775807
    long Ped_long = -9876543210;
    // unsigned long range: 0 to 18446744073709551615
    unsigned long Ped_long_unsigned = 9876543210;
    // #----------------------- long long ----------------------
    // long long range: -9223372036854775808 to
       9223372036854775807
    long long Ped_longlong = -123456789012345;
    // unsigned long long range: 0 to 18446744073709551615
    unsigned long long Ped_longlong_unsigned = 123456789012345;

    // ----------------------- printing -----------------------
    printf("char used for saving character: %c\n", Ped_RealChar);
    printf("char used saving integer BUT the character is printed
       : %c\n", Ped_NumChar);
    printf("char used saving integer: %d\n", Ped_NumChar);
    printf("unsigned char saving integer: %u\n",
       Ped_NumChar_unsigned);
    printf("short: %hd\n", Ped_sh);
    printf("unsigned short: %hu\n", Ped_sh_unsigned);
    printf("int: %d\n", Ped_int);
    printf("unsigned int: %u\n", Ped_int_unsigned);
    printf("long: %ld\n", Ped_long);
    printf("unsigned long: %lu\n", Ped_long_unsigned);
    printf("long long: %lld\n", Ped_longlong);
    printf("unsigned long long: %llu\n", Ped_longlong_unsigned);
}
```

Now you have the source code `Pedram.c`, open it in VScode by executing `code Pedram.c` in the terminal. You should be able to see the code in the opened window. Now we need another terminal inside the VScode to compile and run the code. To open a terminal in VScode, go to the View menu at the top of the window. From the View menu, select "Terminal". You can see

which directory this terminal is in by executing `pwd` in the terminal. Change your directory to the one where source code `Pedram.c` is. We need to do this so when we are compiling the code, the compiler can find the source code and translate it to the machine's language!

Let's checkout the code. A `char` is a type used to save a single character, like `a` or `G` or anything else. But you can assign a numeric value to a `char` variable in C. In fact, a `char` variable is internally represented as a small integer. So, you can assign a number within the range of -128 to 127 a `char` variable.

In this example, the decimal value 80 is assigned to the `char` variable `Ped_RealChar`, which I have picked this name to save this value. The `%c` format specifier in the `printf` statement is used to print the character representation of `Ped_RealChar`. In this case, it will print the character 'P', as the ASCII value 80 corresponds to the character 'P'.

So, while a `char` variable is primarily used to represent characters, it can also store numeric values within its valid range. We will learn more about characters and strings in section Characters and strings. This is what the output should be:

```
char used for saving character: P
char used saving integer BUT the character is printed: P
char used saving integer: 80
unsigned char saving integer: 252
short: -1234
unsigned short: 56789
int: -42
unsigned int: 123456
long: -9876543210
unsigned long: 9876543210
long long: -123456789012345
unsigned long long: 123456789012345
```

Run this code after the lecture, by exceeding the limits and see the warnings **OR** errors **OR wrong** results. Let's say, change the value `Ped_NumChar_unsigned` to 256 which is higher than then maximum allowed for this type. The other thing you can do, is applying arithmetic operations on different types of variables that we have learned in section Arithmetic Operations on Integers.

What is the problem with this code? Variable names are too long. I can just search for a variable in VScode to see the type of variable when it is initialized!

2.2.7 Fixed-width Integer types

In the section Using `printf` and `limits.h` to Get the Limits of Integers we talked about the limits of integer that might be different in different platforms. **How we can write a code that is portable to any OS?**

The C99 standard introduced fixed-width integer types in order to provide a consistent and portable way of specifying integer sizes across different platforms. Prior to C99, the sizes of integer types like `int` and `long` were implementation-dependent, which could lead to issues when writing code that relied on specific bit widths.

By adding fixed-width integer types such as `int8_t`, `int16_t`, `int32_t`, and `int64_t`, the C99 standard ensured that programmers had precise control over the sizes of their integer variables. These types have guaranteed widths in bits, making them useful in situations where exact bit-level manipulation with low-level systems is required.

To use the fixed-width integer types, the header `<stdint.h>` needs to be included. This header provides the type definitions for these fixed-width types, ensuring consistency across different platforms. By including `<stdint.h>`, programmers can use these types with confidence, knowing the exact size and range of the integers they are working with.

In addition to `<stdint.h>`, the header ¡inttypes.h¿ is included to access the placeholders associated with the fixed-width integer types. These format specifiers, such as `PRId8`, `PRIu16`, and so on, enable proper printing and scanning of these types using the `printf` function. Make a new source code by `nano FixedInteger.c`, and paste the following code inside the file and save this program. Open the code in VScode using `code FixedInteger.c` and change the directory where the source code `FixedInteger.c` is in. Compile and execute the code by:

`gcc -o FixedInteger FixedInteger.c`

and

`./FixedInteger`.

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

int main() {

 printf("Size of int8_t: %zu bits\n", 8 * sizeof(int8_t));
 printf("Signed int8_t range: %" PRId8 " to %" PRId8 "\n",
    INT8_MIN, INT8_MAX);
```

```
    printf("\n");

    printf("Size of uint8_t: %zu bits\n", 8 * sizeof(uint8_t));
    printf("uint8_t range: %d to %" PRIu8 "\n", 0, UINT8_MAX);
    printf("\n");

    printf("Size of int16_t: %zu bits\n", 8 * sizeof(int16_t));
    printf("Signed int16_t range: %" PRId16 " to %" PRId16 "\n",
       INT16_MIN, INT16_MAX);
    printf("\n");

    printf("Size of uint16_t: %zu bits\n", 8 * sizeof(uint16_t));
    printf("uint16_t range: %d to %" PRIu16 "\n", 0, UINT16_MAX);
    printf("\n");

    printf("Size of int32_t: %zu bits\n", 8 * sizeof(int32_t));
    printf("Signed int32_t range: %" PRId32 " to %" PRId32 "\n",
       INT32_MIN, INT32_MAX);
    printf("\n");

    printf("Size of uint32_t: %zu bits\n", 8 * sizeof(uint32_t));
    printf("uint32_t range: %d to %" PRIu32 "\n", 0, UINT32_MAX);
    printf("\n");

    printf("Size of int64_t: %zu bits\n", 8 * sizeof(int64_t));
    printf("Signed int64_t range: %" PRId64 " to %" PRId64 "\n",
       INT64_MIN, INT64_MAX);
    printf("\n");

    printf("Size of uint64_t: %zu bits\n", 8 * sizeof(uint64_t));
    printf("uint64_t range: %d to %" PRIu64 "\n", 0, UINT64_MAX);
}
```

The result not only in my machine, but also in any platform must be the same.

```
    Size of int8_t: 8 bits
    Signed int8_t range: -128 to 127

    Size of uint8_t: 8 bits
```

```
uint8_t range: 0 to 255

Size of int16_t: 16 bits
Signed int16_t range: -32768 to 32767

Size of uint16_t: 16 bits
uint16_t range: 0 to 65535

Size of int32_t: 32 bits
Signed int32_t range: -2147483648 to 2147483647

Size of uint32_t: 32 bits
uint32_t range: 0 to 4294967295

Size of int64_t: 64 bits
Signed int64_t range: -9223372036854775808 to 9223372036854775807

Size of uint64_t: 64 bits
uint64_t range: 0 to 18446744073709551615
```

2.3 Characters and strings

In C, a string is defined as a sequence of characters. Individual characters are enclosed in single quotes `' '`, while strings are enclosed in double quotes `" "`.

To print a character, we use the placeholder `%c` in the printf function. For example, if `char c = 'P'`, then `printf("%c", c)`; will print the value of the character variable `c`.

To print a string, we use the placeholder `%s` in `printf`. For example, if `char s[] = "Pedram"`, then `printf("%s", s);` will print the contents of the string variable `s`.

The size of a string can be determined using the `sizeof` operator (the same as finding the size of integer values), which returns the number of bytes occupied by the string. To print the size, we can use `%zu` as the placeholder in `printf`.

Strings in C are null-terminated, meaning they end with a **null character** (represented by 0). When accessing elements of a string, the index starts from 0, and the last character is always the null character. For example, in the `string s[] = "Pedram"`, `s[6]` refers to the null character. Don't for get the indexing in C starts from 0! Look at the following example:

```c
#include <stdio.h>
// Compile and run the code with and without string.h
#include <string.h>

int main() {
 char c = 'P';
 char s[] = "Pedram";
 char s2[] = "Pasandide";

 printf("Character: %c\n", c);
 printf("String: %s\n", s);
 printf("s[0]: %c\n", s[0]);
 printf("s[5]: %c\n", s[5]);
 printf("Size of s: %zu\n", sizeof(s));
 printf("s[6] (null character): %d\n", s[6]);


 char s3[20]; // Make sure s3 has enough space to hold the
    concatenated string

 strcpy(s3, s); // Copy the content of s to s3
 strcat(s3, s2); // Concatenate s2 to s3

 printf("s3: %s\n", s3);

 return 0;
}
```

Open a terminal, checkout the directory you are in, using `pwd`. Make sure you are still in `/home/username/MECHTRON2MP3`.

Make a new source code with `nano PedramString.c`, and copy-paste the code mentioned above. Open it in VScode with `code PedramString.c`. Change the directory to where you saved the source code. Compile the program with `gcc` and execute the code with `./PedramString`.

To concatenate two strings `s` and `s2` and store the result in `s3`, you can use the `strcpy` function from the `<string.h>` header.

In this code, `s` and `s2` are two strings that you want to concatenate. The variable `s3` is declared as an array of characters, with enough space to hold the concatenated string.

First, the `strcpy` function is used to copy the contents of `s` to `s3`, ensuring that `s3` initially holds the value of `s`. Then, the `strcat` function is used to concatenate `s2` to `s3`, effectively appending the contents of `s2` to `s3`.

Tips! There are many more functions dealing with string, and this one was just an example. Depending on you problem, you can search on Google and find how you can tackle your specific problem. Otherwise, remembering all these functions would be ALMOST impossible.

The result must be like:

```
Character: P
String: Pedram
s[0]: P
s[5]: m
Size of s: 7
s[6] (null character): 0
s3: PedramPasandide
```

2.4 Floating-point Numbers

In scientific programming, integers are often insufficient for several reasons:

- **1. Precision**: Integers have a finite range, and they cannot represent numbers with fractional parts. Many scientific computations involve non-integer values, such as real numbers, measurements, and physical quantities. Floating-point arithmetic allows for more precise representation and manipulation of these non-integer numbers.
- **2. Range**: While `long long int` can store larger integer values compared to regular `int`, it still has a limit. Scientific calculations often involve extremely large or small numbers, such as astronomical distances or subatomic particles. Floating-point numbers provide a wider range of values, accommodating these large and small magnitudes.

Floating-point numbers are represented and manipulated using floating-point arithmetic in CPUs. The encoding of floating-point numbers is based on the IEEE 754 standard, which defines formats for single-precision (32 bits) and double-precision (64 bits) floating-point numbers.

The basic structure of a floating-point number includes three components: the **sign** (positive or negative which can 0 or 1), the base (also known as the significant or **mantissa**), and the

exponent. The base represents the significant digits of the number, and the exponent indicates the scale or magnitude of the number. Any floating-point number is represented in machine by:

$$(-1)^{\text{sign}} \times \text{mantissa} \times 2^{\text{exponent}}$$

For example, let's consider the number 85.3. In binary, it can be represented as approximately 1010101.01001100110011... In the IEEE 754 format, this number would be encoded as per the specifications of single-precision or double-precision floating-point representation. You can use online converters to get this number or you can read more how to do it. Right now in this course you don't need to necessary learn how to do it, and I am mentioning this to illustrate everything clearly!

In the case of decimal fraction 0.3, its binary representation is non-terminating and recurring (0.0100110011...), meaning the binary fraction repeats infinitely. However, due to the finite representation of floating-point numbers in IEEE 754 format, the repeating binary fraction is rounded or truncated to fit the available number of bits. As a result, the exact decimal value of 0.3 cannot be represented accurately in binary using a finite number of bits.

So, when converting 0.3 to binary in the context of IEEE 754 floating-point representation, it will be approximated to the closest binary fraction that can be represented with the available number of bits. The accuracy of the approximation depends on the precision (number of bits) of the floating-point format being used.

The floating-point types used in C are `float`, `double` and `long double`. All these types are always signed meaning that they can represent both positive and negative value.

- `float` **or single-precision** has a precision of approximately 7 decimal digits. With smallest positive value of $1.17549435 \times 10^{-38}$ and largest positive value of $3.40282347 \times 10^{38}$
- `double` **or double-precision** has a precision of approximately 15 decimal digits. With smallest positive value of $2.2250738585072014 \times 10^{-308}$ and the largest positive value of $1.7976931348623157 \times 10^{308}$.
- `long double` **or extended-precision** format can vary in size depending on the platform. In x86 systems, it commonly uses 80 bits, but the specific number of bits for long double can differ across different architectures and compilers. In this course we won't use it!

Get the same results in **your machine** using the following code:

```c
#include <stdio.h>
#include <float.h>

int main() {
```

```
    printf("Precision:\n");
    printf("Float: %d digits\n", FLT_DIG);
    printf("Double: %d digits\n", DBL_DIG);
    printf("Long Double: %d digits\n\n", LDBL_DIG);

    printf("Minimum and Maximum Values:\n");
    printf("Float: Minimum: %e, Maximum: %e\n", FLT_MIN, FLT_MAX);
    printf("Double: Minimum: %e, Maximum: %e\n", DBL_MIN, DBL_MAX);
    printf("Long Double: Minimum: %Le, Maximum: %Le\n", LDBL_MIN,
        LDBL_MAX);
}
```

The results for float and double in any computer should be the same showing the portability of these two types. I suggest you to use only these two at least in this course. In **my machine**, the results are:

```
Precision:
Float: 6 digits
Double: 15 digits
Long Double: 18 digits

Minimum and Maximum Values:
Float: Minimum: 1.175494e-38, Maximum: 3.402823e+38
Double: Minimum: 2.225074e-308, Maximum: 1.797693e+308
Long Double: Minimum: 3.362103e-4932, Maximum: 1.189731e+4932
```

Let's initialize a `double` value and print it using `printf`. This example defines a constant `double Ped` as 1.23456789 and demonstrates different printing options using the `%a.bf` placeholder, where `a` represents the minimum width and `b` specifies the number of digits after the decimal point:

```
#include <stdio.h>

int main() {
    const double Ped = 1.23456789;

    // Printing options with different placeholders
    // Minimum width = 0, 2 digits after decimal
    printf("Printing options for Ped = %.2f:\n", Ped);
```

```
    // Minimum width = 10, 4 digits after decimal
    printf("Printing options for Ped = %10.4f:\n", Ped);

    // Minimum width = 6, 8 digits after decimal
    printf("Printing options for Ped = %6.8f:\n", Ped);
}
```

In the first `printf` statement, `%2.2f` is used, where 2 represents the minimum width (minimum number of characters to be printed) and .2 specifies 2 digits after the decimal point. This will print 1.23 as the output.

In the second printf statement, `%10.4f` is used. Here, 10 represents the minimum width, specifying that the output should be at least 10 characters wide, and .4 indicates 4 digits after the decimal point. This will print 1.2346 as the output, with 4 digits after the decimal and padded with leading spaces to reach a width of 10 characters.

In the third printf statement, `%6.8f` is used. The 6 represents the minimum width, and .8 specifies 8 digits after the decimal point. This will print 1.23456789 as the output, with all 8 digits after the decimal point.

By running this code, you can see the different printing options for the Ped value with varying widths and decimal precision.

```
Printing options for Ped = 1.23:
Printing options for Ped =     1.2346:
Printing options for Ped = 1.23456789:
```

2.4.1 Rounding Error in Floating-point Numbers

Rounding errors occur in floating-point arithmetic due to the finite number of bits allocated for representing the fractional part of a number. The rounding error becomes more prominent as we require higher precision or perform multiple arithmetic operations. The magnitude of the rounding error is typically on the order of the smallest representable number, which is commonly referred to as machine epsilon. **Run the following code!**

```
#include <stdio.h>

int main() {
    const float F = 1.23456789f;
    const double D = 1.23456789;
```

```
    const long double L = 1.23456789L;

    printf("Original values:\n");
    printf("Float: %.8f\n", F);
    printf("Double: %.8lf\n", D);
    printf("Long Double: %.8Lf\n\n", L);


    printf("Rounded values:\n");
    printf("Float: %.20f\n", F);
    printf("Double: %.20lf\n", D);
    printf("Long Double: %.20Lf\n", L);
}
```

In the provided code, the original values of `F`, `D`, and `L` are set to 1.23456789f, 1.23456789, and 1.23456789L, respectively. These values are printed with 8 digits of precision using printf statements.

When we examine the output, we can observe some differences between the original values and the rounded values. These differences arise due to the limitations of representing real numbers in the computer's finite memory using floating-point arithmetic.

In the original values section:

The original float value F is printed as 1.23456788, which differs from the original value due to the limited precision of the float data type. The original double value D is printed as 1.23456789, and in this case, there is no visible difference since the double data type provides sufficient precision to represent the value accurately. The original long double value L is also printed as 1.23456789, indicating that the long double data type preserves the precision without any visible loss in this case.

In the rounded values section:

The float value `F` is printed with increased precision using `%.20f`. The rounded value is 1.23456788063049316406, which introduces rounding error due to the limited number of bits available for representing the fractional part of the number. The double value `D` is printed with increased precision using `%.20lf`. Here, we can see a slight difference between the original value and the rounded value, with the rounded value being 1.23456788999999989009. This difference is attributed to the rounding error that occurs in the 16th digit after the decimal point. The long double value `L` is printed with increased precision using `%.20Lf`. The rounded value is 1.23456789000000000003, demonstrating that even with the long double data type, there can still

be a small rounding error.

In the case of `double` precision, the rounding error is approximately on the order of 10^{-16}, meaning that the least significant digit after the 16th decimal place can be subject to rounding error.

It's important to be aware of these limitations and potential rounding errors when performing calculations with floating-point numbers, especially in scientific and numerical computing, where high precision is often required.

```
Original values:
Float: 1.23456788
Double: 1.23456789
Long Double: 1.23456789

Rounded values:
Float: 1.23456788063049316406
Double: 1.23456788999999989009
Long Double: 1.23456789000000000003
```

2.4.2 Type Conversion

In C, type conversion refers to the process of converting a value from one data type to another. There are two types of type conversion: explicit type conversion (also known as type casting) and implicit type conversion (also known as type coercion).

1. Implicit Type Conversion (Type Coercion): Implicit type conversion occurs automatically by the C compiler when performing operations between different data types. The conversion is done to ensure compatibility between operands. Here's an example:

```c
#include <stdio.h>

int main() {
 int num = 10;
 double result = num / 3;  // Implicitly convert int to double
 printf("Result: %f\n", result);

 int num2 = 10.6;
 printf("num2: %d\n", num2); // Implicitly convert double to int
}
```

In the above code, `num` is an integer with a value of `10`. When dividing it by `3`, the division operation requires a common data type for both operands. In this case, the compiler implicitly converts `num` to a double before performing the division. The result is stored in the result variable, which is of type double. When printing `result`, `%f` is used as the format specifier for floating-point numbers. In the other example, `num2` is also defined to be equal to `10.6`, but since the type is `int` anything after decimal will be neglected. Here is the output you should get:

```
Result: 3.000000
num2: 10
```

Implicit type conversion is performed based on a set of rules defined by the C language. It ensures that the operands are of compatible types to perform the desired operation.

2. Explicit Type Conversion (Type Casting): Explicit type conversion involves manually specifying the desired data type for the conversion. It is performed using type casting operators.

```c
#include <stdio.h>

int main() {
 const int num1 = 10;
 const int num2 = 3;

 // 1. Print the division using int placeholder, ignoring
    anything after the decimal
 int resultInt = num1 / num2;
 printf("Division without casting using int placeholder: %d\n",
    resultInt);

 // 2. Print the division using double placeholder without
    casting (warning expected)
 int resultDoubleNoCast = num1 / num2;
 printf("Division without casting floating-point placeholder %f\n
    ", resultDoubleNoCast);

 // 3. Print the division by casting one of the operands
 printf("Division (double with cast): %f\n", num1 / (double)num2)
    ;

 // 4. Print the division by casting both operands
 double resultDoubleBothCasted = (double)num1 / (double)num2;
 printf("Division (double with both cast): %f\n",
```

```
    resultDoubleBothCasted);
}
```

Let's go through the code.

The division of `num1` by `num2` is stored in the `resultInt` variable, which is of type int. When using `%d` as the format specifier, the output will be an integer, ignoring anything after the decimal point.

The division without casting `(num1  num2)` is assigned to `resultDoubleNoCast`, which is of type double. However, this can lead to unexpected results due to integer division. The warning suggests that an implicit conversion is happening from int to double, which may not yield the desired precision.

To ensure proper division with decimal points, we cast one of the operands (`num2`) to double explicitly. This casting allows for a more accurate calculation. In some compiler you might see a warning here! To avoid that:

To perform the division correctly, both `num1` and `num2` are explicitly cast to double. This ensures that the division is carried out using floating-point arithmetic and produces the desired result. The `%f` format specifier is used to print the double value.

```
Division without casting using int placeholder: 3
Division without casting floating-point placeholder 0.000000
Division (double with cast): 3.333333
Division (double with both cast): 3.333333
```

2.4.3 Arithmetic Operations on Floating-point Numbers

.

The same as applying arithmetic operations on integer values, we can do the same on floating-point numbers. Here's a comprehensive example demonstrating arithmetic operations on floating-point numbers, including different data types, const and non-const values, and various arithmetic operators:

```
#include <stdio.h>

int main() {
 const float num1 = 10.5;
 float num2 = 5.2;
 double num3 = 7.8;
```

```c
const long double num4 = 3.14;
long double num5 = 2.71;

// Addition
float sumFloat = num1 + num2;
double sumDouble = num1 + num3;
long double sumLongDouble = num4 + num5;

printf("Addition:\n");
printf("%.2f + %.2f = %.2f\n", num1, num2, sumFloat);
printf("%.2f + %.2f = %.2f\n", num1, num3, sumDouble);
printf("%.2Lf + %.2Lf = %.2Lf\n", num4, num5, sumLongDouble);
printf("\n");

// Subtraction
float diffFloat = num1 - num2;
double diffDouble = num1 - num3;
long double diffLongDouble = num4 - num5;

printf("Subtraction:\n");
printf("%.2f - %.2f = %.2f\n", num1, num2, diffFloat);
printf("%.2f - %.2f = %.2f\n", num1, num3, diffDouble);
printf("%.2Lf - %.2Lf = %.2Lf\n", num4, num5, diffLongDouble);
printf("\n");

// Multiplication
float productFloat = num1 * num2;
double productDouble = num1 * num3;
long double productLongDouble = num4 * num5;

printf("Multiplication:\n");
printf("%.2f * %.2f = %.2f\n", num1, num2, productFloat);
printf("%.2f * %.2f = %.2f\n", num1, num3, productDouble);
printf("%.2Lf * %.2Lf = %.2Lf\n", num4, num5, productLongDouble)
   ;
printf("\n");

// Division
float quotientFloat = num1 / num2;
```

```
    double quotientDouble = num1 / num3;
    long double quotientLongDouble = num4 / num5;

    printf("Division:\n");
    printf("%.2f / %.2f = %.2f\n", num1, num2, quotientFloat);
    printf("%.2f / %.2f = %.2f\n", num1, num3, quotientDouble);
    printf("%.2Lf / %.2Lf = %.2Lf\n", num4, num5, quotientLongDouble
       );
    printf("\n");

    // Compound Assignment Operators
    num2 += num1;
    num3 -= num4;
    num5 *= num4;
    float P1 = num1; // since I cannot change the value of num1
    P1 /= num2;

    printf("Compound Assignment Operators:\n");
    printf("num2 += num1: %.2f\n", num2);
    printf("num3 -= num4: %.2f\n", num3);
    printf("num5 *= num4: %.2Lf\n", num5);
    printf("num1 /= num2: %.2f\n", P1);
}
```

There are some compound assignment operators in the code including `+=`, `-=`, `*=`, and `/=`. How they work?

`+=` (Addition Assignment): It adds the value on the right-hand side to the variable on the left-hand side and assigns the result back to the variable. Example: a += b; is equivalent to a = a + b;

`-=` (Subtraction Assignment): It subtracts the value on the right-hand side from the variable on the left-hand side and assigns the result back to the variable. Example: a -= b; is equivalent to a = a - b;

`*=` (Multiplication Assignment): It multiplies the variable on the left-hand side by the value on the right-hand side and assigns the result back to the variable. Example: a *= b; is equivalent to a = a * b;

`/=` (Division Assignment): It divides the variable on the left-hand side by the value on the right-hand side and assigns the result back to the variable. Example: a /= b; is equivalent to a =

a / b;

Note that we can use these compounds on inter values! The result you should get is:

```
Addition:
10.50 + 5.20 = 15.70
10.50 + 7.80 = 18.30
3.14 + 2.71 = 5.85

Subtraction:
10.50 - 5.20 = 5.30
10.50 - 7.80 = 2.70
3.14 - 2.71 = 0.43

Multiplication:
10.50 * 5.20 = 54.60
10.50 * 7.80 = 81.90
3.14 * 2.71 = 8.51

Division:
10.50 / 5.20 = 2.02
10.50 / 7.80 = 1.35
3.14 / 2.71 = 1.16

Compound Assignment Operators:
num2 += num1: 15.70
num3 -= num4: 4.66
num5 *= num4: 8.51
num1 /= num2: 0.67
```

2.5 Mathematical Functions

Mathematical functions in C are a set of functions provided by the standard library to perform common mathematical operations. These functions are declared in the `<math.h>` header file. They allow you to perform calculations involving numbers, trigonometry, logarithms, exponentiation, rounding, and more.

Some of the important and commonly used mathematical functions in C include:

- `sqrt(x)`: Calculates the square root of a number x.

- `pow(x, y)`: Raises x to the power of y.
- `fabs(x)`: Computes the absolute value of x.
- `sin(x)`, `cos(x)`, `tan(x)`: Computes the sine, cosine, and tangent of an angle x, respectively.
- `log(x)`: Computes the natural logarithm of x.
- `exp(x)`: Calculates the exponential value of x.
- `floor(x)`, ceil(x), round(x): Perform different types of rounding operations on x.
- `fmod(x, y)`: Calculates the remainder of dividing x by y.

Here is an example using these functions. Compile the code using `gcc -o pedram pedram.c`.

```c
#include <stdio.h>
#include <math.h>

int main() {
 const double num1 = 4.0;
 const double num2 = 2.5;

 double sqrtResult = sqrt(num1);
 double powResult = pow(num1, num2);
 double sinResult = sin(num1);
 double logResult = log(num1);
 double ceilResult = ceil(num2);
 double fmodResult = fmod(num1, num2);

 printf("Square root of %.2f: %.2f\n", num1, sqrtResult);
 printf("%.2f raised to the power of %.2f: %.2f\n", num1, num2,
    powResult);
 printf("Sine of %.2f: %.2f\n", num1, sinResult);
 printf("Natural logarithm of %.2f: %.2f\n", num1, logResult);
 printf("Ceiling value of %.2f: %.2f\n", num2, ceilResult);
 printf("Remainder of %.2f divided by %.2f: %.2f\n", num1, num2,
    fmodResult);
}
```

Probably, you see the following error:

```
/usr/bin/ld: /tmp/ccaR3sn3.o: in function ‘main’:
pedram.c:(.text+0x30): undefined reference to ‘sqrt’
/usr/bin/ld: pedram.c:(.text+0x50): undefined reference to ‘pow’
/usr/bin/ld: pedram.c:(.text+0x67): undefined reference to ‘sin’
/usr/bin/ld: pedram.c:(.text+0x7e): undefined reference to ‘log’
/usr/bin/ld: pedram.c:(.text+0x95): undefined reference to ‘ceil’
/usr/bin/ld: pedram.c:(.text+0xb5): undefined reference to ‘fmod’
collect2: error: ld returned 1 exit status
```

Why? By default, the GCC compiler includes standard C libraries, but it does not automatically include all other libraries such as the `math` library. Therefore, when you use math functions like `sqrt`, `pow`, or `log`, the linker needs to know where to find the implementation of these functions.

Including the `<math.h>` header file in your code is necessary to provide the function prototypes and declarations for the math functions. It allows the compiler to understand the function names, parameter types, and return types when you use these functions in your code. However, including `<math.h>` alone is not sufficient to resolve references to the math functions during the linking phase (run the code without including `<math.h>` and check the errors!). The math library (`libm`) that contains **the actual implementation** of the math functions needs to be linked explicitly.

To solve this issue, the `-lm` flag explicitly tells the compiler to link with the math library (`libm`), allowing it to resolve the references to the math functions used in your code. You don't need to remember all these flags you can always use Google. Now compiler the code using `gcc -o pedram pedram.c -lm`. Execute the object, and the result must be:

```
Square root of 4.00: 2.00
4.00 raised to the power of 2.50: 32.00
Sine of 4.00: -0.76
Natural logarithm of 4.00: 1.39
Ceiling value of 2.50: 3.00
Remainder of 4.00 divided by 2.50: 1.50
```

2.6 Statements

2.6.1 Comparison: `if` and `switch`

1. `if` **statement**:

The `if` statement in C is a conditional statement that allows you to perform different actions based on the evaluation of a condition. It follows a general format:

```
if (condition) {
 // Code to execute if the condition is true
}
```

What does it mean? The `condition` is an expression that evaluates to either **true** or **false**. If the condition is **true**, the code inside the if block will be **executed**. If the condition is **false**, the code inside the if block will be **skipped**. Let's say if **a** greater that **b** (`a>b`), execute the code inside the `if` block. Note that `condition` is replaced by `a>b`, and it is called **Comparison operators**. Comparison operators used in `if` statements:

- `==`: Checks if two values are equal.
- `!=`: Checks if two values are not equal.
- `<`: Checks if the left operand is less than the right operand.
- `>`: Checks if the left operand is greater than the right operand.
- `<=`: Checks if the left operand is less than or equal to the right operand.
- `>=`: Checks if the left operand is greater than or equal to the right operand.

There are also some **Logical operators** for combining conditions. Let's say **A** is the first `condition` and B is the second `condition`:

- `!A`: Logical **NOT** operator. It means if the condition **A** is not true, execute the statement in the `if` block. For example. in `!(a>b)`, execute the code if **a** is **NOT** greater than **b**.
- `A || B`: Logical **OR** operator. It evaluates to true if **either** operand **A or B** is true. So, one of these conditions at least must be true to execute the `if` block.
- `A && B`: Logical **AND** operator. It evaluates to true only if both operands **A and B** are true. It means both **A and B** conditions **MUST** be **true**.

Another way to include more complex `if` statement with multiple conditions is to use the general format using `if`, `else if`, and `else`:

```
if (condition1) {
 // Code to execute if condition1 is true
} else if (condition2) {
```

```
 // Code to execute if condition2 is true
} else {
 // Code to execute if all previous conditions are false
}
```

In this format, each condition is checked sequentially. If the first condition is true, the corresponding code block is executed. If it's false, the next condition is checked. If none of the conditions are true, the code block inside the `else` block is executed. Here is an example:

```
#include <stdio.h>

int main() {
 int a = 5;
 int b = 10;

 if (a == b) {
  printf("a is equal to b\n");
 } else if (a != b) {
  printf("a is not equal to b\n");
 } else if (a < b) {
  printf("a is less than b\n");
 } else if (a > b) {
  printf("a is greater than b\n");
 } else if (a <= b) {
  printf("a is less than or equal to b\n");
 } else if (a >= b) {
  printf("a is greater than or equal to b\n");
 } else {
  printf("None of the conditions are true\n");
 }

 int A = 1;
 int B = 0;

 if (!A) {
  printf("A is false\n");
 }

 if (A || B) {
  printf("At least one of A or B is true\n");
```

```
  }

  if (A && B) {
   printf("Both A and B are true\n");
  }
 }
```

You should see the following results:

```
a is not equal to b
At least one of A or B is true
```

A and B were supposed to be conditions why they are equal to 0 and 1? In C, the value **0** is considered **false**, and any **non-zero** value is considered **true**. Take a look at the following example:

```
#include <stdio.h>

int main() {
 int condition1 = 0;
 int condition2 = 1;

 if (condition1) {
  printf("Condition 1 is true\n");  //statement 1.1
 } else {
  printf("Condition 1 is false\n"); //statement 1.2
 }

 if (condition2) {
  printf("Condition 2 is true\n");  //statement 2.1
 } else {
  printf("Condition 2 is false\n"); //statement 2.2
 }
}
```

The result is given here. based on the results we can say **statement 1.2** and **statement 2.1** are executed and the rest is skipped.

```
Condition 1 is false
Condition 2 is true
```

In C, the `stdbool.h` header provides a set of definitions for Boolean data types and values. It introduces the `bool` type, which is specifically designed to represent **Boolean values**. The `bool` type can have two possible values: `true` and `false`. This header also defines the constants true and false as macro **constants**.

Using `stdbool.h` and the `bool` type can improve code readability and express the intent more clearly when dealing with **Boolean values**. It enhances code portability, as it ensures consistent Boolean semantics across different platforms and compilers. Here's an example that demonstrates the usage of `stdbool.h` and the `bool` type:

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

int main() {
 bool condition1 = true;
 bool condition2 = false;

 if (condition1) {
  printf("Condition 1 is true\n");  // statement 1.1
 } else {
  printf("Condition 1 is false\n"); // statement 1.2
 }

 if (condition2) {
  printf("Condition 2 is true\n");  // statement 2.1
 } else {
  printf("Condition 2 is false\n"); // statement 2.2
 }
}
```

The result is given here. based on the results we can say **statement 1.1** and **statement 2.2** are executed and the rest is skipped. In this example, we include the `stdbool.h` header and declare two `bool` variables `condition1` and `condition2`. We assign `true` to `condition1` and `false` to `condition2`. We then use these variables in if statements to check their values and print corresponding messages.

```
Condition 1 is true
Condition 2 is false
```

By using `stdbool.h` and the `bool` type, the code becomes more self-explanatory, as it explicitly shows the usage of Boolean values. The constants true and false provide clarity in expressing

the intent of conditions, making the code more readable and maintainable.

2. `switch` **statement**:

The switch statement in C is a control flow statement that allows you to select one of several execution paths based on the value of an expression. It provides an alternative to using multiple `if` statements when you have a series of conditions to check against a single variable.

The general format of the switch statement in C is as follows:

```
switch (expression) {
 case value1:
 // code to be executed if expression matches value1
 break;
 case value2:
 // code to be executed if expression matches value2
 break;
 case value3:
 // code to be executed if expression matches value3
 break;
 // more cases...
 default:
 // code to be executed if expression doesn't match any case
}
```

Warning! The `break` statement is used to exit the `switch` statement after executing the corresponding code block. Without the break statement, the execution would **fall through** to the next case, resulting in unintended behavior.

Here's an example that demonstrates the usage of the `switch` statement:

```
#include <stdio.h>

int main() {
 int choice;

 printf("Enter a number between 1 and 3: ");
 scanf("%d", &choice);

 switch (choice) {
  case 1:
```

```
    printf("You chose option 1.\n");
    break;
    case 2:
    printf("You chose option 2.\n");
    break;
    case 3:
    printf("You chose option 3.\n");
    break;
    default:
    printf("Invalid choice.\n");
   }
  }
```

Here, we have used `scanf` function. The `scanf` function in C is used to read input from the standard input (usually the keyboard) and assign it to variables based on specified format specifiers. It allows you to accept user input during program execution, making your program more interactive. The general format of the `scanf` function is `scanf(format, argument_list);`, where the `format` parameter specifies the format of the expected input, while the `argument_list` contains the addresses of variables where the input will be stored. Here we have `scanf("%d", &choice)`, which means the given number will be saved in `choice` with format of `int` since we used placeholder `%d`.

In this example, the user is prompted to enter a number between 1 and 3. The input is stored in the variable `choice`. The `switch` statement is then used to check the value of choice and execute the corresponding code block. The output will be different based on the value you enter every time you execute the object file by `./pedram`.

If the user enters 1, the code inside the case 1 block is executed, printing "You chose option 1." If the user enters 2, the code inside the case 2 block is executed, printing "You chose option 2." If the user enters 3, the code inside the case 3 block is executed, printing "You chose option 3."

If the user enters any other value, the code inside the default block is executed, printing "Invalid choice."

More example of `scanf()`! The `scanf` function scans the input stream and tries to match the input with the specified format. It skips whitespace characters by default and stops scanning when it encounters a character that doesn't match the format specifier. Here's an example to illustrate the usage of `scanf`:

```
#include <stdio.h>

int main() {
 int age;
 float height;

 printf("Enter your age: ");
 scanf("%d", &age);

 printf("Enter your height in meters: ");
 scanf("%f", &height);

 printf("You are %d years old and %.2f meters tall.\n", age,
     height);
}
```

In this example, the `scanf` function is used to read user input for age and height. The `%d` format specifier is used to read an integer value, and the `%f` format specifier is used to read a floating-point value. The `&` operator is used to obtain the address of the variables age and height for `scanf` to store the input values.
After the input is read, the program prints the values of `age` and `height` using `printf`.
It's important to note that `scanf` requires correct format specifiers to match the input data type. Failure to use the correct format specifier can lead to unexpected behavior or errors in your program. Additionally, input validation and error handling are crucial when using `scanf` to ensure that the input is valid and the expected values are successfully read.

2.6.2 Loops and Iterations: `while`

The `while` statement in C is a control flow statement that allows you to repeatedly execute a block of code as long as a specified condition is true. It provides a way to create loops in your program.

The general format of the while statement in C is as follows:

```
while (condition) {
 // code to be executed while the condition is true
}
```

The `condition` is a **Boolean** expression that determines whether the loop should continue or terminate. **As long as the condition evaluates to true**, the code inside the loop will be executed repeatedly. If the condition becomes false, the loop will be exited, and the program will continue with the next statement after the loop. Here's an example that demonstrates the usage of the `while` statement:

```
#include <stdio.h>

int main() {
 int count = 1;

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

In this example, the while loop is used to print the value of the count variable as long as it is less than or equal to 5. The count variable is **incremented** by 1 in each iteration using the `count++` statement.

The while loop continues to execute as long as the condition count ¡= 5 is true. Once the count value becomes 6, the condition becomes **false**, and the loop is terminated. Here is the output in terminal.

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

Something that I forgot to tell you! **Incrementing** and **decrementing** are unary **operators** in C that are used to increase or decrease the value of a variable by a specific amount.

The **increment** operator ++ is used to **increment** the value of a variable by 1. It can be applied as a prefix (`++x`) or a postfix (`x++`) operator. When used as a prefix, the increment operation is performed before the value is used in an expression. When used as a postfix, the increment operation is performed after the value is used in an expression. Similarly, the decrement operator – is used to decrease the value of a variable by 1. It follows the same prefix and postfix notation as the increment operator. Here is an example:

```c
#include <stdio.h>

int main() {
 int x = 5;
 printf("Original value: %d\n", x);

 printf("After x++: %d\n", x++);  // Postfix increment
 printf("Print x after x++ is applied: %d\n", x);
 printf("Print x again: %d\n", x);
 printf("After ++x: %d\n", ++x);  // Prefix increment

 printf("After x--: %d\n", x--);  // Postfix decrement
 printf("Print x after x-- is applied: %d\n", x);
 printf("Print x again: %d\n", x);
 printf("After --x: %d\n", --x);  // Prefix decrement
}
```

and the result is:

```
Original value: 5
After x++: 5
Print x after x++ is applied: 6
Print x again: 6
After ++x: 7
After x--: 7
Print x after x-- is applied: 6
Print x again: 6
After --x: 5
```

2.6.3 Loops and Iterations: `do`

The `do` statement is another type of loop in C that is similar to the `while` loop. The main difference is that the `do` loop executes the code block first and then checks the condition. This guarantees that the code inside the loop will be executed **at least once**, even if the condition is initially false.

The general format of the do statement in C is as follows:

```
do {
 // code to be executed
} while (condition);
```

Here's an example to illustrate the usage of the `do` statement:

```
#include <stdio.h>

int main() {
 int count = 1;

 do {
  printf("Count: %d\n", count);
  count++;
 } while (count <= 5);
}
```

In this example, the `do` loop is used to print the value of the `count` variable and increment it by 1. The loop continues to execute as long as the condition `count <= 5` is true. Since the initial value of count is 1, the loop body will be executed once, and then the condition is checked. If the condition is true, the loop will repeat, and if the condition is false, the loop will be exited. Here is the result:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

2.6.4 Loops and Iterations: `for`

The `for` statement in C is a looping construct that allows you to execute a block of code repeatedly based on a specific condition. It is typically used when you know the exact number of iterations or when you need to perform a specific action for a fixed range of values. The general format of the for statement is as follows:

```
for (initialization; condition; increment/decrement) {
 // Code to be executed in each iteration
}
```

The `initialization` step is used to initialize the loop control variable before the loop starts. It is typically used to set an initial value.

The `condition` is evaluated before each iteration. If the condition is true, the loop body is executed; otherwise, the loop terminates.

The **increment** or **decrement** step is performed after each iteration and updates the loop control variable. It is used to control the termination condition of the loop.

Here's an example that demonstrates the usage of the `for` loop to print numbers from 5 to 0 using **decrement**:

```
#include <stdio.h>

int main() {
 for (int i = 5; i >= 0; i--) {
  printf("%d \n", i);
 }
}
```

In this example, the loop is initialized with `int i = 5`, the condition is `i >= 0`, and `i` is decremented by `i--` after each iteration. The loop iterates as long as the condition `i >= 0` is **true**. In each iteration, the value of `i` is printed using `printf`.

```
5
4
3
2
1
0
```

2.6.5 Nested loops

Nested loops in C are loops that are placed inside another loop. They allow you to perform repetitive tasks in a structured and organized manner when dealing with multiple dimensions or when you need to iterate over a combination of rows and columns. The general structure of nested loops is as follows:

```
for (outer initialization; outer condition; outer increment/
   decrement) {
 // Code before the inner loop

 for (inner initialization; inner condition; inner increment/
    decrement) {
  // Code inside the inner loop
 }

 // Code after the inner loop
}
```

Here's an example that demonstrates nested loops by printing out a matrix-like pattern based on rows and columns:

```
#include <stdio.h>

int main() {
 int rows = 5;
 int columns = 3;

 for (int i = 1; i <= rows; i++) {
  for (int j = 1; j <= columns; j++) {
   printf("(%d, %d) ", i, j);
  }
  printf("\n");
 }
}
```

In this example, we have an outer loop that iterates over the rows and an inner loop that iterates over the columns. The outer loop is controlled by the variable `i`, which represents the current row, while the inner loop is controlled by the variable `j`, which represents the current column.

The inner loop prints the coordinates `(row, column)` for each position in the matrix-like pattern. After printing the values for a row, we insert a newline character using `printf("n")` to move to the next row.

```
(1, 1) (1, 2) (1, 3)
(2, 1) (2, 2) (2, 3)
(3, 1) (3, 2) (3, 3)
(4, 1) (4, 2) (4, 3)
(5, 1) (5, 2) (5, 3)
```

As you can see, the nested loops allow us to iterate over each row and column combination, printing out the corresponding coordinates in the pattern.

Nested loops are commonly used when working with multi-dimensional arrays, matrix operations, nested data structures, or any situation that requires iteration over multiple levels or dimensions. They provide a powerful tool for handling complex repetitive tasks in a structured manner.

2.6.6 Loops and Iterations: Controlling the Loop

Controlling loops in C involves using certain statements like `break`, `continue`, and `goto` to alter the flow of execution within the loop. These statements provide control over how and when the loop iterations are affected or terminated.

- `break` statement is used to immediately exit the loop, regardless of the loop condition. When encountered, the program flow continues to the next statement after the loop. It is typically used to prematurely terminate a loop based on a specific condition.
- `continue` statement is used to skip the current iteration of the loop and move to the next iteration. It allows you to bypass the remaining code in the current iteration and start the next iteration immediately. It is often used to skip certain iterations based on a condition.
- `goto` statement allows you to transfer the control of the program to a labelled statement elsewhere in the code. It is a powerful but potentially risky construct, as it can lead to complex and less readable code. It is generally advised to use `goto` sparingly and with caution.

Here's an example that demonstrates the usage of `break`, `continue`, and `goto` statements within a loop:

```c
#include <stdio.h>

int main() {
 int i;

 // Example with break
 for (i = 1; i <= 10; i++) {
  if (i == 5) {
   break;
  }
  printf("%d ", i);
 }
 printf("\n");

 // Example with continue
 for (i = 1; i <= 10; i++) {
  if (i % 2 == 0) {
   continue;
  }
  printf("%d ", i);
 }
 printf("\n");

 // Example with goto
 for (i = 1; i <= 3; i++) {
  printf("Outer loop iteration: %d\n", i);
  for (int j = 1; j <= 3; j++) {
   printf("Inner loop iteration: %d\n", j);
   if (j == 2) {
    goto end;
   }
  }
 }
 end:
 printf("Goto example finished.\n");
}
```

In this example, we have three loops that demonstrate the usage of `break`, `continue`, and

`goto`:

The first loop uses break to exit the loop when `i` is equal to **5**. As a result, the loop terminates prematurely and only prints the numbers from **1** to **4**.

The second loop uses `continue` to skip printing even numbers. When `i` is divisible by 2, the loop skips the remaining code and moves to the next iteration. As a result, only odd numbers from 1 to 10 are printed.

The third loop demonstrates the usage of `goto`. In this case, when the inner loop reaches iteration 2, the `goto` statement is encountered, and the control jumps to the `end` label, bypassing the remaining iterations of the inner and outer loops. Finally, the message "Goto example finished" is printed.

These control statements provide flexibility and allow you to alter the flow of execution within loops, making your code more efficient and concise in certain situations. However, it's important to use them judiciously and maintain code readability and clarity.

```
1 2 3 4
1 3 5 7 9
Outer loop iteration: 1
Inner loop iteration: 1
Inner loop iteration: 2
Goto example finished.
```

2.6.7 Variable Scope

Variable scope refers to the portion of a program where a variable is accessible and can be used. In C, the scope of a variable is determined by its declaration and the block of code within which it is declared. Understanding variable scope is crucial for writing well-structured and maintainable code.

Let's consider an example using a for loop to illustrate different scenarios of variable scope:

```c
#include <stdio.h>

int main() {
 int x = 5;  // Variable x declared in the main function

 printf("Before the for loop: x = %d\n", x);

 for (int i = 0; i < 3; i++) {
```

```
    int y = i * 2;  // Variable y declared inside the for loop

    printf("Inside the for loop: y = %d\n", y);
    printf("Inside the for loop: x = %d\n", x);
  }

  printf("Outside the for loop: y = %d\n", y);

  printf("After the for loop: x = %d\n", x);
}
```

In this code, we have two variables: x and y. Here's a breakdown of the variable scope in different scenarios:

x has a global scope as it is declared in the main function. It can be accessed and used anywhere within the main function, including inside the for loop.

y has a local scope limited to the block of code within the for loop. It is only accessible inside the for loop's block and ceases to exist once the loop iteration ends. Each iteration of the loop creates a new instance of the variable y.

Inside the for loop, both x and y are accessible because variables declared in outer scopes can be accessed in inner scopes.

Outside the for loop, attempting to access y will result in a compilation error since it is no longer in scope. The y variable is limited to the block of code within the for loop.

By observing the output of the `printf` statements, you can see the value of x remains the same throughout the program since it has a wider scope. However, the value of y changes with each iteration of the for loop, demonstrating the limited scope of the variable.

Compiling this program will result the following error:

```
Pedram.c: In function ‘main’:
pedram.c:16:46: error: ‘y’ undeclared (first use in this function)
```

If we declare the variable `y` without initialization (like the following code), by `int y;` right after `int x = 5;`, there is an address in memory allocated to save it. If I compile and run it, in **my computer**, every time I see an irrelevant and different value for `y`. This is the same problem as we saw in Declaring and Naming with and without Initializing.

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

```c
int main() {
 int x = 5;  // Variable x declared in the main function
 int y;

 printf("Before the for loop: x = %d\n\n", x);

 for (int i = 0; i < 3; i++) {
  int y = i * 2;  // Variable y declared inside the for loop

  printf("Iteration: %d\n",i);
  printf("Inside the for loop: y = %d\n", y);
  printf("Inside the for loop: x = %d\n\n", x);
 }

 printf("Outside the for loop: y = %d\n", y);

 printf("After the for loop: x = %d\n", x);
}
```

This is because inside the loop, the variable `y` is declared again and another address in memory is allocated to save it, which is not the same as previous one. Let's remove re-declaration inside the loop by:

```c
#include <stdio.h>

int main() {
 int x = 5;  // Variable x declared in the main function
 int y;

 printf("Before the for loop: x = %d\n\n", x);

 for (int i = 0; i < 3; i++) {
  y = i * 2;

  printf("Iteration: %d\n",i);
  printf("Inside the for loop: y = %d\n", y);
  printf("Inside the for loop: x = %d\n\n", x);
 }

 printf("Outside the for loop: y = %d\n", y);
```

```
  printf("After the for loop: x = %d\n", x);
 }
```

This time I get the following results. Inside the loop, the value of `y` in every iteration is updated and save in the initial address given to this variable. Another example variable that has been declared inside the loop is `i`. Try different scenarios of this variable at home!

```
Iteration: 0
Inside the for loop: y = 0
Inside the for loop: x = 5

Iteration: 1
Inside the for loop: y = 2
Inside the for loop: x = 5

Iteration: 2
Inside the for loop: y = 4
Inside the for loop: x = 5

Outside the for loop: y = 4
After the for loop: x = 5
```

If you don't need the variable `y` after the loop the best code is just not initialize and not print it out after the loop is over like:

```
#include <stdio.h>

int main() {
 int x = 5;  // Variable x declared in the main function

 printf("Before the for loop: x = %d\n\n", x);

 for (int i = 0; i < 3; i++) {
  int y = i * 2;  // Variable y declared inside the for loop

  printf("Iteration: %d\n",i);
  printf("Inside the for loop: y = %d\n", y);
  printf("Inside the for loop: x = %d\n\n", x);
 }
```

```
    printf("After the for loop: x = %d\n", x);
}
```

2.7 Arrays

In C, an array is a collection of elements of the same data type that are stored in contiguous memory locations. Arrays provide a way to store and access multiple values under a single variable name. They are widely used for storing and manipulating data efficiently.

The general format of declaring an array is:

```
type arrayName[numberOfElements];
```

Here's an example of declaring an array without initializing the values for its elements:

```
int numbers[5];
```

In this example, we declare an array named `numbers` that can hold 5 integer elements. The individual elements within the array are accessed using indices ranging from 0 to 4 (`numbers[0]` to `numbers[4]`).

If you want to initialize the values for the elements at the time of declaration, you can use the following format:

```
type arrayName[numberOfElements] = {value1, value2, ..., valueN};
```

Here's an example of declaring an array and initializing the values for its elements:

```
int numbers[5] = {10, 20, 30, 40, 50}; // Initializing an integer
    array with specific values
```

In this example, we declare and initialize an integer array named `numbers` with 5 elements. The elements of the array are initialized with the values 10, 20, 30, 40, and 50 respectively.

It's important to note that if you provide fewer values in the initialization list than the size of the array, the remaining elements will be automatically initialized to the default value for their respective type (e.g., 0 for integers, 0.0 for floating-point numbers, and '\0' for characters).

The following example shows the concept discussed in about arrays.

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

```c
int main() {
 int numbers1[5];  // Declaration of an integer array with size 5

 int numbers2[5] = {10, 20, 30, 40, 50};

 // Printing the array elements without initializing
 printf("without initializing:\n");
 for (int i = 0; i < 5; i++) {
  printf("numbers1[%d] = %d\n", i, numbers1[i]);
 }

 printf("\nwith initializing\n");

 // Printing the array elements with initializing
 for (int i = 0; i < 5; i++) {
  printf("numbers2[%d] = %d\n", i, numbers2[i]);
 }
}
```

Tips! In C, arrays are zero-indexed, which means the first element in an array is accessed using the index 0. This indexing convention is consistent throughout the language and is an important concept to understand when working with arrays.

In my machine I get the following results:

```
without initializing:
numbers1[0] = -648048640
numbers1[1] = 32764
numbers1[2] = 16777216
numbers1[3] = 257
numbers1[4] = 2

with initializing
numbers2[0] = 10
numbers2[1] = 20
numbers2[2] = 30
numbers2[3] = 40
numbers2[4] = 50
```

You can declare and initialize the array with an initializer list:

```
int numbers[5] = {0};  // Initializes all elements to zero
```

In this example, the first element is explicitly initialized to 0, and the remaining elements will be automatically initialized to 0 as well.

Accessing an array element out of its valid range in C leads to undefined behavior. It means that the program's behavior becomes unpredictable, and it may result in crashes, errors, or unexpected output. Here's an example that demonstrates accessing an array element out of range:

```
#include <stdio.h>

int main() {
 int numbers[5] = {10, 20, 30, 40, 50};

 printf("Accessing elements within the valid range:\n");
 for (int i = 0; i < 5; i++) {
  printf("numbers[%d] = %d\n", i, numbers[i]);
 }

 printf("\nAccessing elements out of the valid range:\n");
 printf("numbers[6] = %d\n", numbers[6]);  // Accessing element
    outside the valid range
}
```

What I get in **my machine** is:

```
Accessing elements within the valid range:
numbers[0] = 10
numbers[1] = 20
numbers[2] = 30
numbers[3] = 40
numbers[4] = 50

Accessing elements out of the valid range:
numbers[6] = -317639680
```

The general format of a multi-dimensional array in C is as follows:

```
type arrayName[size1][size2]...[sizeN];
```

Here, type represents the data type of the elements in the array, and `size1`, `size2`, ..., `sizeN` represent the sizes of each dimension of the array. Here's an example of a two-dimensional array and how to print its elements:

```c
#include <stdio.h>

int main() {
 int matrix[3][4] = {
  {1, 2, 3, 4},
  {5, 6, 7, 8},
  {9, 10, 11, 12}
 };

 printf("Printing the elements of the two-dimensional array:\n");
 for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 4; j++) {
   printf("%d ", matrix[i][j]);
  }
  printf("\n");
 }
}
```

In this example, we declare a two-dimensional array named matrix with 3 rows and 4 columns. The elements of the array are initialized using an initializer list. The outer loop iterates over the rows, and the inner loop iterates over the columns.

The nested loops allow us to access and print each element of the two-dimensional array using the indices `matrix[i][j]`, where i represents the row index and j represents the column index. The outer loop controls the row iteration, and the inner loop controls the column iteration.

By iterating through the rows and columns, we can print out each element of the two-dimensional array in a structured manner.

Keep in mind that you can extend this pattern to higher-dimensional arrays by adding additional nested loops to iterate through each dimension.

The results should be:

```
Printing the elements of the two-dimensional array:
1 2 3 4
5 6 7 8
9 10 11 12
```

2.8 Functions

Functions in C are reusable blocks of code that perform a specific task. They help organize and modularize code by breaking it into smaller, manageable units. Functions provide a way to encapsulate a set of instructions, making code more readable, maintainable, and reusable.

Here are some examples of functions that we used so far:

- `int main()`: The main function serves as the entry point of a C program. It is required in every C program and acts as the starting point for execution.
- `printf`: The `printf` function is part of the standard C library and is used to output formatted text to the console or other output streams. It takes a format string and additional arguments, allowing you to display values and formatted text.

The general format of a function declaration in C is as follows:

```
returnType functionName(parameter1, parameter2, ... parameterN) {
 // Function body
 // Statements and computations
 // Optional return statement
 return output; // the types of variable output is returnType
}
```

- `returnType` specifies the data type of the value that the function returns. It can be `void` if the function does not return any value.
- `functionName` represents the name of the function, which is used to call the function from other parts of the program. Let's say in `printf()` function the name of the function is `printf`
- parameters are optional and define the variables that the function receives as input. They act as placeholders for values that are passed to the function when it is called. Let's say in `printf()` the parameter that function can receive is a string, placeholder, and a value to print out.
- FunctionBody contains the statements and computations that make up the function's logic. It specifies what the function does when it is invoked.

It's important to note that functions cannot be nested in C. Nested functions are not supported in standard C; only the main function can be defined within another function.

Functions provide a way to modularize code, improve code re-usability, and enhance code readability. They allow you to break down complex tasks into smaller, more manageable pieces, making it easier to understand and maintain your code. By organizing code into functions, you can also promote code reuse, as functions can be called multiple times from different parts of a program.

Here's an example that includes two functions: one to calculate the surface area of a circle and another to calculate the area of a circle based on its radius.

```c
#include <stdio.h>

float calculateSurfaceArea(float radius) {
 const float pi = 3.14159;
 float surfaceArea = 2 * pi * radius;
 return surfaceArea;
}

float calculateArea(float radius) {
 const float pi = 3.14159;
 float area = pi * radius * radius;
 return area;
}

int main() {
 float radius = 5.0; // in meter

 float surfaceArea = calculateSurfaceArea(radius);
 printf("Surface area of the circle [m]: %.2f\n", surfaceArea);

 float area = calculateArea(radius);
 printf("Area of the circle [m^2]: %.2f\n", area);

 return 0;
}
```

In this example, we define two functions: `calculateSurfaceArea` and `calculateArea`. The `calculateSurfaceArea` function takes a float parameter radius and returns the surface area of the circle using the formula $2 \cdot \pi \cdot radius$. Similarly, the `calculateArea` function also takes a float parameter radius and returns the area of the circle using the formula $pi \cdot \pi \cdot radius$.

In the main function, we declare a float variable radius with a value of 5.0. We then call

the `calculateSurfaceArea` function with radius as an argument and store the result in the `surfaceArea` variable. We also call the `calculateArea` function with radius as an argument and store the result in the area variable.

Finally, we use `printf` to display the calculated surface area and area of the circle on the console, with two decimal places of precision (%.2f). The result must be:

```
Surface area of the circle [m]: 31.42
Area of the circle [m^2]: 78.54
```

By encapsulating the calculation logic within separate functions, we can easily reuse these functions for different radii in our program. This approach promotes code modularity and improves readability by separating the specific calculations into individual functions.

Here's an example of a function that prints a two-dimensional array without a return value:

```
#include <stdio.h>

void printArray(int rows, int cols, int array[rows][cols]) {
 for (int i = 0; i < rows; i++) {
  for (int j = 0; j < cols; j++) {
   printf("%d ", array[i][j]);
  }
  printf("\n");
 }
}

int main() {
 int matrix[3][4] = {
  {1, 2, 3, 4},
  {5, 6, 7, 8},
  {9, 10, 11, 12}
 };

 printArray(3, 4, matrix);  // Call the function to print the
    array

 return 0;
}
```

The result must be exactly when we printed the element inside the `main` function in the previous section.

In this example, the function `printArray` takes three parameters: `rows`, `cols`, and `array`. It receives the dimensions of the array as well as the array itself. The function then uses nested loops to iterate over each element of the two-dimensional array and prints its value.

Inside the main function, we declare a two-dimensional array named matrix with 3 rows and 4 columns. We initialize the array with some values. Then, we call the `printArray` function, passing the dimensions of the array (3 for rows and 4 for columns) as well as the array itself (matrix). The function prints the elements of the array in a structured manner.

Since the `printArray` function does not need to return a value, its return type is specified as void. The function solely focuses on printing the array and does not perform any other operations.

By using a function with a void return type, we can encapsulate the logic of printing a two-dimensional array and reuse it whenever needed.

2.8.1 Variable Scope in Functions

The same concept about Variable Scope matters here. Take a look at the following example:

```c
#include <stdio.h>

void printNumber() {
 int number = 10;  // Variable declared inside the function

 printf("Number inside the function: %d\n", number);
}

int main() {
 int number = 5;  // Variable declared inside the main function

 printf("Number inside the main function: %d\n", number);

 printNumber();
}
```

In this example, we have two variables named number declared in different scopes: one inside the `main` function and another inside the `printNumber` function.

Inside the main function, we declare and initialize the variable number with a value of `5`. We then print the value of number within the main function, which outputs `5`.

Next, we call the `printNumber` function from within the main function. Inside the `printNumber`

function, we declare and initialize a separate variable number with a value of `10`. We then print the value of number within the `printNumber` function, which outputs `10`.

The key point here is that the two number variables have separate scopes. The number variable inside the `printNumber` function is local to that function and exists only within the function's block. It does not interfere with the number variable declared in the main function. The result must be:

```
Number inside the main function: 5
Number inside the function: 10
```

2.8.2 Passing a Constant Value to a Function

Passing a constant value to a function can be beneficial in several ways:

- It provides clarity: Declaring a parameter as a constant indicates that the function will not modify the value, making the function's behavior more explicit and self-documenting.
- It prevents unintentional modifications: Using a constant parameter ensures that the value passed to the function remains unchanged within the function's scope. This can help prevent accidental modifications and maintain the integrity of the original value.
- It enhances code safety: By using constants, you establish a contract between the calling code and the function, ensuring that the passed value will not be modified. This promotes safer and more predictable code execution.

Here's an example that demonstrates passing a constant value to a function:

```c
#include <stdio.h>

void printNumber(const int value) {
 printf("Value inside the function: %d\n", value);
}

int main() {
 int number = 5;

 printf("Number inside the main function: %d\n", number);

 printNumber(number);
}
```

Since value is declared as a constant parameter in the `printNumber` function, it cannot be modified within the function. This provides the assurance that the value passed to the function will not be accidentally changed within the function's scope.

A function can also itself. When a function calls itself, it is known as **recursion**. Recursion is a powerful programming technique where a function solves a problem by breaking it down into smaller, similar subproblems. It is a fundamental concept in computer science and often provides elegant solutions for problems that exhibit repetitive or self-similar structures.

Recursion involves the following key elements:

- Base Case: It is the condition that defines the simplest form of the problem and provides the termination condition for the recursive calls. When the base case is reached, the recursion stops, and the function starts returning values.
- Recursive Case: It is the condition where the function calls itself, typically with modified input parameters, to solve a smaller instance of the same problem. The recursive case leads to further recursive calls until the base case is reached.
- Progress towards the Base Case: Recursive functions must ensure that each recursive call brings the problem closer to the base case. Otherwise, the recursion may result in an infinite loop.

Here's an example of a recursive function to calculate the factorial of a positive integer:

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

uint64_t factorial(uint32_t n) {
 // Base case: factorial of 0 or 1 is 1
 if (n == 0 || n == 1) {
  return 1;
 }

 // Recursive case: factorial of n is n multiplied by factorial
    of (n-1)
 return n * factorial(n - 1);
}

int main() {
```

```
    uint32_t num = 5;

    uint64_t result = factorial(num);
    printf("Factorial of %"PRIu32" is %"PRIu64"\n", num, result);
}
```

In this example, the factorial function takes an unsigned integer `n` as input and recursively calculates the factorial of `n`. The base case is defined for n equal to 0 or 1, where the function returns 1 since the factorial of 0 or 1 is 1. In the recursive case, the function calls itself with the parameter n - 1, effectively reducing the problem size with each recursive call until the base case is reached.

When the program runs, the main function calls the factorial function with `num` as an argument. The factorial function uses recursion to calculate the factorial of `num`, and the result is printed. The result must be:

```
Factorial of 5 is 120
```

Recursion is a powerful technique that allows functions to solve complex problems by dividing them into smaller, manageable sub-problems. However, it's essential to ensure that recursive functions have a well-defined base case and progress towards that base case to avoid infinite recursion.

2.8.3 Forward Declaration of a Function

The provided code demonstrates the definition of the `factorial` function before the `main` function. This approach is valid and does not cause any errors.

However, if you attempt to define the `factorial` function after the `main` function, it will result in a compilation error. Compiler the following code:

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

int main() {
    uint32_t num = 5;

    uint64_t result = factorial(num);
    printf("Factorial of %"PRIu32" is %"PRIu64"\n", num, result);
}
```

```
uint64_t factorial(uint32_t n) {
 // Base case: factorial of 0 or 1 is 1
 if (n == 0 || n == 1) {
  return 1;
 }

 // Recursive case: factorial of n is n multiplied by factorial
    of (n-1)
 return n * factorial(n - 1);
}
```

This is because C requires functions to be declared or defined before they are used. When the `factorial` function is defined after the `main` function, the compiler encounters the function call in main before it sees the actual definition of factorial. As a result, the compiler doesn't have information about the function's implementation, leading to an error.

To overcome this issue, you can provide a function declaration before the `main` function. A function declaration specifies the function's name, return type, and parameter types without including the function body. It informs the compiler about the existence and signature of the function, allowing you to call it before the actual definition.

Here's an example of how you can declare the `factorial` function before the `main` function:

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

uint64_t factorial(uint32_t n);  // Function declaration

int main() {
 uint32_t num = 5;

 uint64_t result = factorial(num);
 printf("Factorial of %" PRIu32 " is %" PRIu64 "\n", num, result)
    ;
}

uint64_t factorial(uint32_t n) {
 // Function definition
 if (n == 0 || n == 1) {
```

```
    return 1;
  }

  return n * factorial(n - 1);
}
```

This must give you the same output we have got in the previous section. In this updated code, the function `factorial` is declared before the `main` function with a function prototype that specifies the function's name, return type, and parameter types. This informs the compiler about the existence of the factorial function and its signature.

By providing a function declaration, you can call the `factorial` function in `main` even before its actual definition. The function definition is later provided after the `main` function, and the code compiles successfully.

This approach of declaring a function before its actual definition is known as providing a forward declaration or function prototype. It allows you to use the function before its implementation, satisfying the requirement of declaring functions before they are used in C.

It was said that `int main` is also a function that can receive inputs. Here's an example of a C code that demonstrates the usage of input arguments in the `main` function, checks the number of inputs, and utilizes the input arguments with different types:

```
#include <stdio.h>
#include <stdlib.h> // Include this header for atoi and atof

int main(int argc, char *argv[]) {
  // Check the number of inputs
  if (argc != 3) {
    printf("Incorrect number of inputs. Expected 2 inputs.\n");
    return 1; // Exit the program with an error status
  }

  // Retrieve the input arguments
  int arg1 = atoi(argv[1]); // Convert the first argument to an
     integer
  float arg2 = atof(argv[2]); // Convert the second argument to a
     float

  // Use the input arguments
  printf("First argument: %d\n", arg1);
```

```
    printf("Second argument: %.2f\n", arg2);

    return 0; // Exit the program with a success status
}
```

In this example, the main function receives two input arguments: `argc` and `argv`. `argc` represents the number of input arguments passed to the program, and `argv` is an array of strings that holds the actual input arguments.

The code first checks if the number of inputs is equal to `3` (the program name itself counts as an argument). If it's not, an error message is printed, and the program exits with an error status (`return 1`).

If the number of inputs is correct, the code converts the first argument (`argv[1]`) to an integer using the `atoi` function and stores it in the `arg1` variable. Similarly, the second argument (`argv[2]`) is converted to a float using the `atof` function and stored in the `arg2` variable.

Finally, the program uses the input arguments by printing them to the console. The `%d` format specifier is used to print the integer `arg1`, and the `%.2f` format specifier is used to print the float `arg2` with two decimal places.

To compile you can simply use `gcc -o pedram pedram.c`. To run the code as an example you can execute `./pedram 10 3.14`. Here, 10 and 3.14 are the input arguments passed to the program. You can modify them as needed.

```
First argument: 10
Second argument: 3.14
```

2.9 Global Variables

A global variable in C is a variable that is defined outside of any function and is accessible throughout the entire program. It has global scope, meaning it can be accessed and modified by any function within the program.

Global variables are declared outside of any function, typically at the top of the program, before the main function. They can be used to store values that need to be shared across multiple functions or accessed from different parts of the program. Here's an example of a global variable:

```
#include <stdio.h>

// Global constant variable
const int MAX_VALUE = 100;
```

```c
// Global non-constant variable
int globalVariable = 50;

void function1() {
 // Access the global constant variable
 printf("Max value: %d\n", MAX_VALUE);

 // Access the global non-constant variable
 printf("Global variable: %d\n", globalVariable);

 // Modify the global non-constant variable
 globalVariable = 75;
}

void function2() {
 // Access the modified global variable from function1
 printf("Updated global variable: %d\n", globalVariable);
}

int main() {
 function1();
 function2();

}
```

In this example, we have both a global constant variable named `MAX_VALUE` and a global non-constant variable named `globalVariable`.

The global constant variable `MAX_VALUE` is declared with the `const` keyword, indicating that its value cannot be modified throughout the program. It can be accessed from any function, and its value remains constant.

The global non-constant variable `globalVariable` is declared without the `const` keyword, allowing its value to be modified. It can also be accessed from any function, and any changes made to it will be reflected in other parts of the program that access the variable.

The functions `function1` and `function2` are able to access and modify the global variables. `function1` accesses and modifies the global non-constant variable `globalVariable`, and `function2` accesses the modified value of `globalVariable` from function1.

Global variables can be useful for sharing data between functions, but it is important to use them judiciously, as they can make code harder to understand and maintain due to their global scope. It's generally recommended to limit the use of global variables and favour local variables within functions whenever possible.

2.9.1 Using `define`

The `#define` preprocessor directive in C is used to define symbolic constants and perform textual substitutions during the compilation process. It allows you to define a name as a replacement for a value or a piece of code, which is then substituted wherever the name is encountered in the source code.

Here are a few key points about `#define`:

- Symbolic Constants: With `#define`, you can define symbolic names for constant values, making the code more readable and maintainable. These names act as placeholders for specific values or expressions, providing a way to give meaningful names to commonly used values or configurations.
- Textual Substitution: `#define` performs textual substitution, replacing every occurrence of the defined name with its associated value during the pre-processing phase of compilation. The substitution is done before the actual compilation of the code begins.
- No Memory Allocation: `#define` does not allocate memory. It simply replaces text, allowing you to define aliases or constants without occupying memory space.
- No Type Checking: `#define` does not perform type checking because it is a simple textual substitution. It is important to ensure that the substituted text is valid and compatible with the context where it is used.
- No Scope Limitation: `#define` constants have global visibility and are not bound to any specific scope. They can be accessed and substituted throughout the entire program, regardless of their location.
- No Runtime Overhead: Since `#define` is resolved during the compilation process, there is no runtime overhead associated with its usage. The substituted values or code are already present in the compiled program.

Take a look at the following example:

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

```c
// Global variable
int globalVariable = 50;

// Using #define
#define CONSTANT_VALUE 50

void function1() {
 globalVariable = 75;  // Modify the global variable
}

void function2() {
 printf("Global variable: %d\n", globalVariable);
 printf("Constant value: %d\n", CONSTANT_VALUE);
}

int main() {
 function1();
 function2();
}
```

In this example, we have a global variable named `globalVariable` and a constant value defined using `#define` called `CONSTANT_VALUE`, both set to 50.

The global variable `globalVariable` is mutable, and its value can be modified by functions within the program. In the function1 function, we modify the value of `globalVariable` to 75.

On the other hand, the constant value `CONSTANT_VALUE` defined using `#define` is a symbolic name that represents the value 50. It cannot be modified or reassigned since it is a pre-processor directive for text substitution. In the `function2` function, we print both the value of the global variable and the constant value.

The difference between the two becomes apparent when considering their characteristics:

The global variable `globalVariable` has a data type (integer in this case) and occupies memory space. It can be modified and accessed within the program.

The `#define` constant `CONSTANT_VALUE` is a symbolic representation of the value 50. It does not occupy memory space since it is substituted during the compilation process. It cannot be modified or assigned a different value.

In summary, the global variable allows for mutable data storage, while the `#define` constant provides a way to define symbolic names for values without occupying memory.

3 Intermediate Topics in C

3.1 Debugging in C

Debugging in C using the GNU Debugger (GDB) is a powerful technique for identifying and resolving issues in your C programs. GDB allows you to examine and manipulate the execution of your program, helping you understand and fix bugs, analyze program behavior, and gain insights into code execution.

Do you have GDB on your OS?! GDB is developed by the GNU Project, GDB is the standard debugger for many Unix-like operating systems, including Linux. It is widely used and supported across various platforms. To check if you have GDB installed on your OS you can use `gdb --version`. For any reason if it was not installed:

- On Linux: GDB comes with the compiler but you didn't have it, use `sudo apt install gdb`.
- macOS or iOS: Follow these steps:
  1. Open Terminal and Install Homebrew, a popular package manager for macOS, by executing the following command in Terminal: `/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"`
  2. Once Homebrew is installed, use it to install GDB by running the following command: `brew install gdb`
  3. After the installation is complete, you may need to create a certificate to allow GDB to control other processes on your system. Run the following command: `codesign --entitlements gdb-entitlement.xml -fs gdb-cert /usr/local/bin/gdb`
     This step is necessary to grant GDB the necessary permissions for debugging.
  4. Finally, you can verify the installation by running: `gdb --version`

  Please note that starting with macOS Catalina (10.15) and later versions, the system's security measures restrict the use of GDB for debugging certain processes, such as system processes or processes that you do not have appropriate permissions for. Additionally, you may need to adjust your system's security settings to allow GDB to function properly. Refer to the GDB documentation or online resources for more information on using GDB on macOS and any additional steps that may be required.

To enable debugging with GDB, you need to compile your C code with the `-g` flag. This flag includes debugging information in the compiled executable, such as symbol tables, source file names, and line number information. This information is crucial for GDB to provide meaningful debugging capabilities.

Instead of `-g` flag we may use `-ggdb3` which the debugging information provided in executable object file is in the format of GDB debugger, while `-g` is more generic and can be used with different debuggers. So, if you are using LLDB, you must use `-g`.

Still have problem with GDB?! There is another debugger called `lldb` that you can use both on Linux or macOS. LLDB (LLVM Debugger) is developed as part of the LLVM project, LLDB is a relatively newer debugger and was designed to be a replacement for GDB. Initially focused on macOS and iOS, it has since expanded to support other platforms like Linux and Windows. This time I am sure `lldb` should be on macOS because it comes with the compiler!! But if it doesn't please let me know. Because I don't have macOS I might be wrong!
On Linux although you may need to install it. To check if you have it use: `lldb --version`. If you don't install it with `sudo apt install lldb`.

Here are the differences between different levels of `-ggdb` flags:

- `-ggdb0`: This level disables debugging information generation. It is equivalent to not using the -g flag at all.
- `-ggdb1`: This level generates minimal debugging information. It includes basic symbol table entries and line number information. It is the minimum level recommended for effective debugging.
- `-ggdb2`: This level generates additional debugging information, including macro definitions and more detailed symbol table entries. It provides more comprehensive debugging support than -ggdb1.
- `-ggdb3`: This level generates the maximum amount of debugging information. It includes extra information for local variables and optimizations. It provides the most detailed debugging support but may increase compilation time and executable size.

When using GDB or LLDB, you can set breakpoints, step through the code, inspect variable values, examine the call stack, and perform various debugging operations to understand the program's behavior. GDB or LLDB allows you to interactively debug your program, making it a valuable tool for troubleshooting complex issues.

To start debugging with GDB, use the command `gdb <executable_name>` or in your terminal, where `<executable_name>` is the name of the compiled executable. If you are using LLDB you can do the same by `lldb <executable_name>`. Once in the GDB environment (or LLDB), you can use various commands to navigate, inspect, and manipulate your program's execution.

Remember to remove the `-ggdb` or `-g` flag when compiling your code for production or release builds, as it adds extra information and increases the executable's size. The `-ggdb` or `-g` flags are intended for development and debugging purposes only.

Here's an example of C code that uses a while loop, along with some common GDB commands:

```c
#include <stdio.h>

int main() {
 int i = 0;

 while (i < 10) {
  printf("Iteration %d\n", i);
  i++;
 }
 printf("The end of loop\n");
}
```

To compile this code with the -ggdb3 flag for maximum debugging information, you can use the following command:

`gcc -ggdb3 pedram.c -o pedram`

Tips! There is no different between
`gcc -ggdb3 pedram.c -o pedram`
and
`gcc -ggdb3 -o pedram pedram.c`

Once compiled, you can start debugging the program using `gdb ./pedram`(or `lldb ./pedram` if you are using LLDB). Here's an example of using some common GDB or LLDB commands:

- **Setting a Breakpoint**: You can set a breakpoint at a specific line of code using the break command. For example, to set a breakpoint at line 8 (inside the while loop), use `break 8`. If you are using LLDB you can do the same by `breakpoint set --line 8`.
- **Starting Execution**: In both GDB and LLDB, once a breakpoint is set, you can start the

execution of the program using the `run` command. Although you may set more breakpoints during debugging.

- **Stepping through the Code**: In both GDB and LLDB, to step through the code line by line, you can use the `next` or `n` command. It will execute the current line and stop at the next line.
- **Continuing Execution**: In both GDB and LLDB, if you want to continue execution after hitting a breakpoint or stopping at a specific line, you can use the `continue` or `c` command.
- **Printing Variable Values**: In both GDB and LLDB, to print the value of a variable during debugging, you can use the `print` or `p` command. For example, `print i` will print the value of the variable `i`.
- **Quitting GDB or LLDB**: To exit the GDB or LLDB debugger, you can use the `quit` command.

Here's an example of how you might interact with GDB using the code provided:

```
For help, type "help".
--Type <RET> for more, q to quit, c to continue without paging--c
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./pedram...
(gdb) break 7
Breakpoint 1 at 0x117e: file pedram.c, line 7.
(gdb) run
Starting program: /home/pedram/MECHTRON2MP3/pedram
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at pedram.c:7
7               printf("Iteration %d\n", i);
(gdb) continue
Continuing.
Iteration 0

Breakpoint 1, main () at pedram.c:7
7               printf("Iteration %d\n", i);
(gdb) next
Iteration 1
8               i++;
```

```
(gdb) print i
$1 = 1
(gdb) c
Continuing.

Breakpoint 1, main () at pedram.c:7
7               printf("Iteration %d\n", i);
(gdb) c
Continuing.
Iteration 2

Breakpoint 1, main () at pedram.c:7
7               printf("Iteration %d\n", i);
(gdb) p i
$2 = 3
(gdb) exit
A debugging session is active.

Inferior 1 [process 61784] will be killed.

Quit anyway? (y or n) y
```

Using these additional flags can provide more comprehensive warning messages and enforce stricter adherence to the C language standard. They can help catch potential issues, improve code quality, and ensure compliance with best practices. Here is a list of some useful flags:

- `-Wall` enables additional warning messages during compilation. It enables common warnings that help catch potential issues and improve code quality.
- `-Wextra` enables even more warning messages beyond those enabled by -Wall. It includes additional warnings that are not included in -Wall.
- `-Wconversion` generates warnings for implicit type conversions that may cause loss of data or unexpected behavior.
- `-Wsign-conversion` generates warnings for implicit sign conversions, where signedness is changed during assignments or comparisons.
- `-Wshadow` generates warnings for variable shadowing, which occurs when a variable in an inner scope has the same name as a variable in an outer scope.

- `-Wpedantic` generates warnings for strict ISO C adherence. It enables additional warnings that follow the strictest interpretation of the C language standard.
- `-std=c17` specifies the C language standard to be used during compilation. In this case, it specifies the C17 standard, which is the ISO/IEC 9899:2017 standard for the C programming language.

Quite easy to use! To compile the same program you can use:

```
gcc -ggdb3 -Wall -Wextra -Wconversion -Wsign-conversion -Wshadow -Wpedantic
-std=c17 pedram.c -o pedram
```

Debugger uses these flags to include almost full information about the code in the object file during compiling. Sometimes you are testing your program and you may need to compile your code many times. Every time adding these flags might be time consuming. That's where we might need Makefile.

3.2 Makefile

A Makefile is a text file that contains a set of rules for compiling and building programs. It is used to automate the compilation process by specifying dependencies, compilation flags, and target outputs. Makefiles are commonly used in C projects to simplify building and managing complex codebases.

Why we need Makefile? Makefiles provide a convenient way to manage the compilation process and handle dependencies in C projects. They allow developers to specify the relationships between different source files and ensure that only the necessary files are recompiled when changes are made. Makefiles also make it easier to manage build configurations, compilation flags, and linking options. Overall, Makefiles streamline the build process and help maintain code consistency and reproducibility.

Makefiles consist of rules, targets, dependencies, and commands. Here's an overview of some key components:

- **Rules**: Rules define the relationship between targets and dependencies. They specify how to build targets from dependencies.
- **Targets**: Targets represent the desired outputs, such as executable files or object files. They can be source files, intermediate files, or final build artifacts.
- **Dependencies**: Dependencies are files or other targets that are required to build a specific target. If a dependency is modified, the corresponding target needs to be rebuilt.

- **Commands**: Commands are the actual shell commands executed to build a target. They specify how to compile source files, link object files, and generate the final output.

A simple example of Makefile could be:

```
CC = gcc
CFLAGS = <this is where flags are added>
SOURCES = main.c
OBJECTS = $(SOURCES:.c=.o)
EXECUTABLE = myprogram

all: $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(OBJECTS) -o $(EXECUTABLE)

%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm -f $(OBJECTS) $(EXECUTABLE)
```

The Makefile defines the following rules:

- `all`: The default target that builds the executable.
- `$(EXECUTABLE)`: Specifies the dependencies and commands to build the executable.
- `%.o`: Specifies the rule for compiling object files from C source files.
- `clean`: A target to clean the generated object files and executable.

By running make in the directory with this Makefile, it will compile the source files using the specified flags and generate the `myprogram` executable. Running make clean will remove the generated object files and executable.

Let's take a look at the following example save in the source code named `factorial.c`.

```
#include <stdio.h>

int factorial(int n) {
 if (n == 0 || n == 1) {
```

```
    return 1;
  }
  return n * factorial(n - 1);
}

int main() {
  int num = 5;
  int result = factorial(num);
  printf("Factorial of %d is %d\n", num, result);
  return 0;
}
```

Open a terminal and use `nano Makefile` commend to make a new makefile. Copy and paste the following code and save the file.

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99
EXECUTABLE = pedram

all: $(EXECUTABLE)

$(EXECUTABLE): factorial.c
	$(CC) $(CFLAGS) -o $(EXECUTABLE) factorial.c

clean:
	rm -f $(EXECUTABLE)
```

Make sure that you have created `Makefile` in the same directory that the source code `factorial.c` is saved. In a terminal with the directory that both files are located in, execute `make`. This will generate an executable object file (`EXECUTABLE`) named `pedram`. To run the code I can simply do `./pedram`. To remove `EXECUTABLE` created I can use `make clean`.

Warning! I you have noticed there a `TAB` space before the line starting with `$(CC) ...` and `rm -f ...`. For some reason that baffle even the most seasoned programmers!!! This `TAB` space format is different in VScode when you are writing a Makefile and probably you would encounter errors like:

```
Makefile:7: *** missing separator.  Stop.
```

To avoid this problem forget about VScode when writing Makefiles. Format you Makefile when you run `nano Makefile` by removing extra spaces and entering `TAB` spaces before mentioned lines. You can also edit your Makefile in the **Text Editor** environment by running `open Makefile`, which opens the file by **Text Editor** after making it with `nano`

3.3 Splitting Code into Multiple Files

Splitting code into multiple files is a common practice in programming, including in languages like C. This modular approach offers several benefits and allows for better organization, readability, and maintainability of codebases. Splitting code into multiple files offers advantages such as:

- **Modularization**: Breaking code into smaller, more manageable units improves organization and readability.
- **Re-usability**: Code can be reused across different projects by linking or including the appropriate files.
- **Maintainability**: Isolating different functionalities or modules in separate files makes it easier to update or modify specific parts without affecting the entire codebase.
- **Compilation Efficiency**: When changes are made to a single file, only that file and its dependencies need to be recompiled, saving compilation time.

By separating code into multiple files, developers can better structure their projects, collaborate more effectively, and build scalable and maintainable software systems.

So far we have programmed only in Source Code Files (`.c` in C or `.cpp` in C++).

- Source code files contain the actual implementation of functions, variables, and other program logic.
- Each source code file typically corresponds to a specific module or functionality of the program.

- They include the necessary header files to gain access to the declarations and definitions needed for the code to compile and run.

You can make your own Header Files (`.h` in C or `.hpp` in C++).

- Header files contain function prototypes, type definitions, macro definitions, and other declarations that need to be shared across multiple source code files. They typically define interfaces and provide a way to communicate between different parts of a program.
- Header files are meant to be included in source code files (.c or .cpp) using the `#include` directive.
- They help in maintaining a separation between interface and implementation, making the code more modular and reusable.

Here's an example of C code with a main source file (`main.c`) and a corresponding header file (`functions.h`).

Save the following code in `main.c`:

```
#include <stdio.h>
#include "functions.h"

int main() {
 int num = 5;
 int result = square(num);

 printf("Square of %d is %d\n", num, result);
}
```

By now you can see VScode even without compiling is giving you an error indicating that it cannot find `functions.h`. Create a file named `functions.h` and paste the following code into it:

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

int square(int num) {
 return num * num;
}

#endif
```

The `#ifndef FUNCTIONS_H` and `#define FUNCTIONS_H` directives are known as include guards or header guards. They are used to prevent multiple inclusions of the same header file within a single compilation unit.

When a header file is included in multiple source files, there is a possibility of multiple definitions and declarations, which can lead to compilation errors due to redefinition of symbols. The include guards help avoid these errors by ensuring that the contents of the header file are processed only once during the compilation process.

Create a file named Makefile (no file extension) and paste the following code into it. To avoid the format errors mentioned in the previous section do it on terminal or Text Editor.

```
CC = gcc
CFLAGS = -Wall -Wextra

all: main

main: main.c functions.h
	$(CC) $(CFLAGS) -o main main.c

clean:
	rm -f main
```

Let's go through each line of the provided Makefile syntax:

- `CC = gcc`: This line assigns the value `gcc` to the variable `CC`. Here, `CC` represents the compiler to be used for compilation.
- `CFLAGS = -Wall -Wextra`: This line assigns the value `-Wall -Wextra` to the variable `CFLAGS`. Here, `CFLAGS` represents the compiler flags or options that are passed to the compiler during compilation. In this case, `-Wall and -Wextra` are flags that enable additional compiler warnings.
- `all: main`: This line defines a target **named** `all`. The `all` target is considered the default target, meaning that it will be executed if no specific target is provided when running make. In this case, the all target depends on the `main` target.
- `main: main.c functions.h`: This line defines the main target. It states that the `main` target **depends** on `main.c` and `functions.h` files. If any of these files are modified, the main target will be considered outdated and need to be rebuilt.
- `$(CC) $(CFLAGS) -o main main.c`: This line is the recipe for building the `main` target. It specifies the commands to be executed to create the `main` executable file.

Here's a breakdown of the syntax:

1. `$(CC)`: This expands the value of the CC variable, which is gcc. So, this represents the compiler command.
2. `$(CFLAGS)`: This expands the value of the CFLAGS variable, which is -Wall -Wextra. So, this represents the compiler flags.
3. `-o main`: This specifies the output file name as main.
4. `main.c`: This is the source file that is passed to the compiler for compilation.
5. `clean: rm -f main`: This line defines the clean target. The clean target is typically used to remove generated files or clean up the project directory. In this case, the clean target specifies the command `rm -f main` to remove the `main` executable file.

In summary, this Makefile specifies a compilation process for building the main executable file. It uses the gcc compiler with the flags -Wall and -Wextra to compile main.c and functions.h. The resulting executable file is named main. Additionally, there is a clean target to remove the generated main file.

Open a terminal, navigate to the directory where the files are saved, and run the following command to compile `make`. This will compile the code and generate an executable file named `main`. Run the program by executing `./main`. You should see the output: "**Square of 5 is 25**".

In this example the implementation of the `square` function was defined directly inside the `functions.h`. The other scenario is to declare the function in the header file `function.h`, but write the actual implementation of `square` in another source code. This approach has some considerations:

- **Code Organization**: Separating the function implementation into a separate source file (functions.c) allows for better code organization. By having separate files for declarations (header file) and definitions (source file), the codebase becomes more modular and maintainable. It also helps in managing larger projects with multiple functions.
- **Compilation Efficiency**: If the function square is defined directly in the header file and included in multiple source files, each source file would have its own copy of the function code. This can lead to code duplication and potentially larger executable sizes. By placing the function definition in a separate source file, the function is compiled only once, and all source files can share the same compiled code.
- **Reducing Rebuilds**: When modifications are made to the function implementation in functions.c, only that file needs to be recompiled. If the function definition is directly in the header file, any change to the function will require recompiling all source files that include the header file, even if they don't directly use the function.

- **Encapsulation and Information Hiding**: Separating the function implementation in a source file helps hide the implementation details from other source files. The header file provides a clean interface (declarations) for other source files to use the functions without exposing the internal implementation.

If you remember when we are using `math.h` by including the header file we still get some error saying that the compiler cannot find the actual implementation of the functions used in our code. That's why we used `-lm` flag to tell compiler where it can find the corresponding source code containing the implementations.

The C library is typically distributed as compiled code, and its source code may not be readily available or accessible to users. The implementation details of library functions, including `sin()` from `math.h`, are considered part of the library's internal implementation and are not exposed to the user.

However, the behavior and specifications of these functions are defined in the C standard, and their functionality is well-documented. The C standard provides guidelines on how functions like `sin()` should behave and what the expected results and behavior are for different input values. In this way, the developer, will not share the codes with users.

This is the most important reason of why sometimes we need to declare the functions in `.h` files but leave the definitions in a another source code with `.c` extension. Keep the `main.c` the same way it was. Change the file `functions.h` into following code where it contains only declaration of the function:

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

int square(int num);

#endif
```

Create a file named `functions.c` and paste the following code into it:

```
#include "functions.h"

int square(int num) {
 return num * num;
}
```

Create a file named Makefile (no file extension) and paste the following code into it:

```
CC = gcc
CFLAGS = -Wall -Wextra

all: main

main: main.o functions.o
$(CC) $(CFLAGS) -o main main.o functions.o

main.o: main.c functions.h
$(CC) $(CFLAGS) -c main.c

functions.o: functions.c functions.h
$(CC) $(CFLAGS) -c functions.c

clean:
rm -f main *.o
```

Let's go through each line of the provided syntax in the context of a Makefile:

- `main: main.o functions.o`: This line specifies the target main and lists its dependencies as `main.o` and `functions.o`. This means that the target main depends on the object files `main.o` and `functions.o`. If any of these object files are modified, the main target will be considered outdated and need to be rebuilt.
- `main.o: main.c functions.h`: This line specifies the target `main.o` and lists its dependencies as `main.c` and `functions.h`. This means that the target `main.o` depends on the source file `main.c` and the header file `functions.h`. If any of these files are modified, the `main.o` target will be considered outdated and need to be rebuilt.
- `functions.o: functions.c functions.h`: This line specifies the target `functions.o` and lists its dependencies as `functions.c` and `functions.h`. This means that the target functions.o depends on the source file `functions.c` and the header file `functions.h`. If any of these files are modified, the `functions.o` target will be considered outdated and need to be rebuilt.

In summary, this Makefile defines targets for building the main executable, `main.o`e object file, and `functions.o` object file. The main target depends on the `main.o` and `functions.o` object files, which in turn depend on the corresponding source files and header files. The compiler commands specified in the recipes compile the source files into object files using the provided flags (`$(CFLAGS)`) and link the object files together to create the final executable. If any of the source

files or header files are modified, the respective targets will be considered outdated and need to be rebuilt.

To practice this program, put some breakpoints in the code, using GDB or LLDB, to see the order of the lines being executed. To so this:

1. First you need to put `-g` flag in the Makefile. So the object file made by compiler will include some information about GDB. If you skip this part, you might see messages like:

   ```
   No symbol table is loaded.  Use the "file" command.
   ```

   when trying to set break points, indicating that `gdb` or `lldb` cannot define your make file must be like:

   `CFLAGS = -g -Wall -Wextra`

2. Execute `make` in the Terminal to make the object file, in this case it must `main`.

3. Run the `main` executable under GDB compiler by `gdb ./main`.

4. If you have copied the code exactly with the same format mentioned above you should have the same order of numbering for the line. Let's say in the `main.c`, the line number 7 is: `int result = square(num);`. Define the following break points:

   - `break main.c:7` will set a breakpoint at line 7:
     `int result = square(num);`

   - `break main.c:9` will set a breakpoint at line 9:
     `printf("Square of %d is %d\n", num, result);`.

   - `break functions.h:4` will set a breakpoint in `functions.h` at line 4:
     `int square(int num);`.

   - `break functions.c:5` will set a breakpoint in `functions.c` at line 5:
     `return num * num;`.

5. Start the debugging by executing `run` in the Terminal.

6. Use `print`, `next`, `continue` and so on, to go through the code. If you don't remember this part go back to Debugging in C.

3.4 Pointer

I think Pointer are the most confusing concept in C. I need your full attention here!!

In C programming, memory is divided into bytes, and each byte consists of 8 bits (one byte). Each byte in memory has a unique address that can be used to access or manipulate the data stored in that memory location.

A pointer in C is a variable that holds the address of another variable. Pointers allow us to indirectly access and manipulate the data stored in a particular memory location. The syntax for declaring a pointer variable is to use an asterisk (`*`) before the variable name, followed by the data type it points to. For example:

```
int *p;           // p is a pointer to an integer
double *Pedram;   // Pedram is a pointer to a double
```

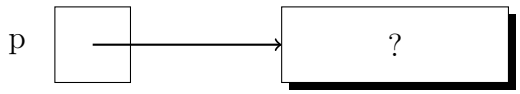
When we declare a pointer variable like `int *p`;, the pointer `p` is capable of storing memory addresses that point to an integer. The unary operator `*` is used to **de-reference** a pointer, which means accessing the value at the memory location that the pointer points to. For example, if `p` points to a memory location that contains an integer, `*p` gives us the value of that integer. The other way around is if a variable is initialized and you want to access the address. If `a` is a variable in memory, then `&a` represents the address where the value of a is stored. In another word, `*` is the inverse of `&`, like a=*&b is equal to a = b!

Take a look at the following example:

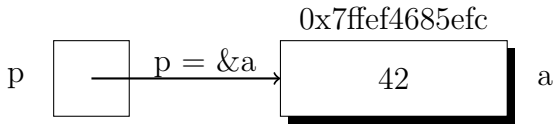
```
#include <stdio.h>

int main() {
 int a = 42;    // Declare and initialize an integer variable

 int *p;        // Declare a pointer to an integer
 p = &a;        // Assign the address of 'a' to the pointer 'p'
 /*or we could use int *p = &a;*/

 printf("Value of 'a': %d\n", a);
 printf("Address of 'a': %p\n", &a); // %p is placeholder
 printf("Value of 'p' (address of 'a'): %p\n", p);

 printf("Value pointed by 'p': %d\n", *p); // De-reference p

 *p = 500;
```

```
  printf("\nPrint out after *p=500.\n");
  printf("Value of 'a': %d\n", a);
  printf("Value pointed by 'p': %d\n", *p);

  a = 98;
  printf("\nPrint out after a = 98.\n");
  printf("Value of 'a': %d\n", a);
  printf("Value pointed by 'p': %d\n", *p);
}
```

In this example, we declared an integer variable `a` and a pointer `p` to an integer. At this moment the pointer is not initialized to point to any address. This figure shows how it looks like:



The, we assigned the address of `a` to the pointer `p` using the address-of operator `&`, like the following figure.

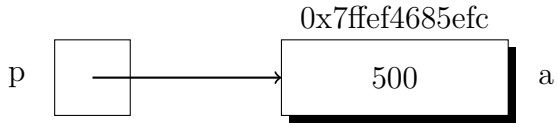


By de-referencing `p` with `*p`, we can access the value stored at the address pointed by `p`, which is the value of `a`. Here is the output in **my computer**. I am saying **my computer**, because the address given to the value `a` to save it, in your computer will be different. Actually, every time you run the code, you can see a new address is given to save this value. Here is results in my computer:

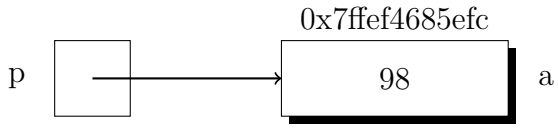
```
Value of 'a': 42
Address of 'a': 0x7ffef4685efc
Value of 'p' (address of 'a'): 0x7ffef4685efc
Value pointed by 'p': 42

Print out after *p=500.
Value of 'a': 500
Value pointed by 'p': 500

Print out after a = 98.
Value of 'a': 98
```

```
Value pointed by 'p': 98
```

After `*p = 500`:



At last after `a = 98`:



It is important that the address `0x7ffef4685efc` was not changed during the whole time, since we changed the value saved in the same location of the memory. Doesn't make sense right!? I know I have been there! Let's try another example:

```c
#include <stdio.h>

int main()
{
 int a, b, *p1, *p2;

 printf("The initial values:\n");
 printf("a=%d\n", a);
 printf("b=%d\n", b);
 printf("The address of a is: %p\n", &a);
 printf("The address of b is: %p\n", &b);
 printf("The address p1 is: %p\n", p1);
 printf("The address p2 is: %p\n", p2);
 printf("\n");

 p1 = &a;
 p2 = p1;
 printf("Result after p1 = &a and p2 = p1:\n");
 printf("Value of *p1: %d\n", *p1);
 printf("Value of *p2: %d\n", *p2);
 printf("Value of a: %d\n", a);
```

```
  printf("The address p1 is: %p\n", p1);
  printf("The address p2 is: %p\n", p2);
  printf("\n");

  *p1 = 1;
  printf("Step 1, after *p1 = 1:\n");
  printf("Value of *p1: %d\n", *p1);
  printf("Value of *p2: %d\n", *p2);
  printf("Value of a: %d\n", a);
  printf("The address p1 is: %p\n", p1);
  printf("The address p2 is: %p\n", p2);
  printf("\n");

  *p2 = 2;
  printf("Step 2, after *p2 = 2;\n");
  printf("Value of *p1: %d\n", *p1); // Output: 2
  printf("Value of *p2: %d\n", *p2); // Output: 2
  printf("Value of a: %d\n", a);     // Output: 2
  printf("The address p1 is: %p\n", p1);
  printf("The address p2 is: %p\n", p2);
 }
```

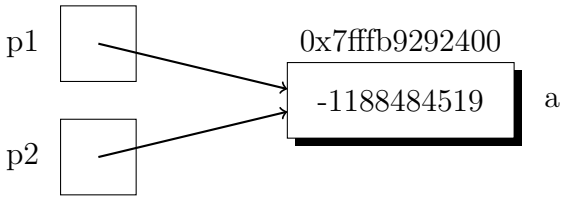
In the first part, `p1` and `p2` are declared as integer pointers. `p1` is set to point to the address of variable `a`. Then `p2` is assigned the value of `p1`. Now both `p1` and `p2` point to the address of `a`, and the value of `a` becomes 1.

Next, `*p2` is modified to 2. Since `p2` is pointing to the same address as `p1`, both `*p1` and `*p2` change to 2, and the value of a also becomes 2. The output in **my computer** is:

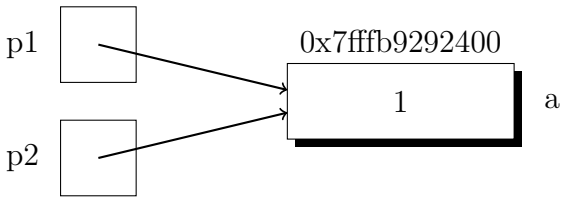
```
The initial values:
a=-1188484519
b=32767
The address of a is: 0x7fffb9292400
The address of b is: 0x7fffb9292404
The address p1 is: 0x64
The address p2 is: 0x1000

Result after p1 = &a and p2 = p1:
Value of *p1: -1188484519
```

```
Value of *p2: -1188484519
Value of a: -1188484519
The address p1 is: 0x7fffb9292400
The address p2 is: 0x7fffb9292400

Step 1, after *p1 = 1:
Value of *p1: 1
Value of *p2: 1
Value of a: 1
The address p1 is: 0x7fffb9292400
The address p2 is: 0x7fffb9292400

Step 2, after *p2 = 2;
Value of *p1: 2
Value of *p2: 2
Value of a: 2
The address p1 is: 0x7fffb9292400
The address p2 is: 0x7fffb9292400
```

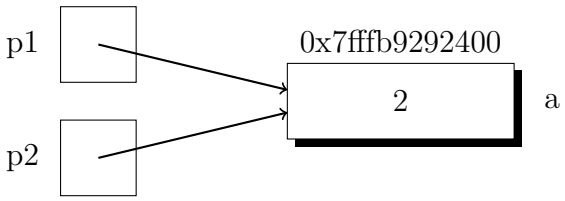
Let's go step by steps through schematic process! At the beginning `a` and `b` integers were declared without initialization. The same as `p1` and `p2` pointers. After `p1=&a`, `p1` will point at the the address of `a`. Using `p1=p2`, `p2` will point at the same address (`&a`) that `p1` is pointing at:



By `*p1 = 1` we are changing the value saved in `p1` address which is the same as `p2` and `&a`:



Like it was said, `p2` is pointing at the same address, so we can change the value saved in this address using `*p2=2`:



Warning! Do not confuse `p1 = p2` with `*p1 = *p2`. Take a look at the following example and compare it with the previous one!

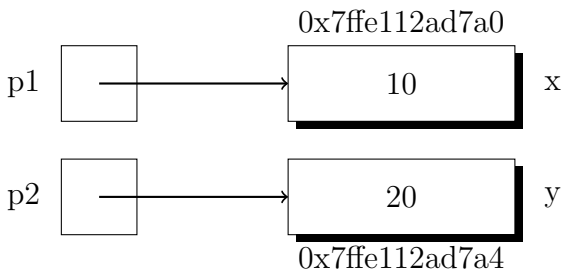
```
#include <stdio.h>

int main()
{

 int x = 10, y = 20, *p1, *p2;
 p1 = &x;
 p2 = &y;

 printf("The initial values:\n");
 printf("x=%d\n", x);
 printf("y=%d\n", y);
 printf("The address of x is: %p\n", &x);
 printf("The address of y is: %p\n", &y);
 printf("The address p1 is: %p\n", p1);
 printf("The address p2 is: %p\n", p2);
 printf("\n");


 *p1 = *p2;
 printf("After *p1 = *p2:\n");
 printf("x=%d\n", x);
 printf("y=%d\n", y);
 printf("Value at address pointed by p1: %d\n", *p1);
 printf("Value at address pointed by p2: %d\n", *p2);

 printf("The address of x is: %p\n", &x);
 printf("The address of y is: %p\n", &y);
 printf("The address p1: %p\n", p1);
 printf("The address p2: %p\n", p2);
}
```

In this example, new integer variables `x` and `y` are declared, and `p1` is set to point to the address of `x`, while `p2` is set to point to the address of `y`. Then, the addresses and values stored at those addresses are printed.

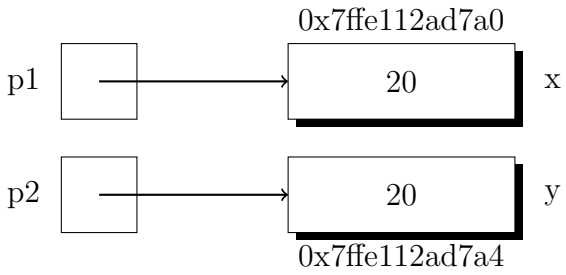
Finally, `*p1` is assigned the value of `*p2`. This means the value stored at the address pointed by `p2` (value of `y`) is **copied** to the address pointed by `p1` (value of `x`). After this step, both `*p1` and `*p2` become 20, and the addresses of `p1` and `p2` remain unchanged.

```
The initial values:
x=10
y=20
The address of x is: 0x7ffe112ad7a0
The address of y is: 0x7ffe112ad7a4
The address p1 is: 0x7ffe112ad7a0
The address p2 is: 0x7ffe112ad7a4

After *p1 = *p2:
x=20
y=20
Value at address pointed by p1: 20
Value at address pointed by p2: 20
The address of x is: 0x7ffe112ad7a0
The address of y is: 0x7ffe112ad7a4
The address p1: 0x7ffe112ad7a0
The address p2: 0x7ffe112ad7a4
```

At the first place we used `p1` and `p2` to point at the addresses of saved variables `x` and `y`, using `p1 = &x` and `p2 = &y`:



After `*p1 = *p2`, we are saying the value saved in the address `p1` (accessible by `*p1`) must be equal to the value saved in the address `p2`:



Let's take a break! Are you still following? Good! Why we are doing this? **why we need pointers?** Pointers are used when passing variables to functions for several reasons:

- **Passing by Reference**: In C, function arguments are typically passed by value, which means a copy of the argument is made and passed to the function. However, when we need to modify the original variable inside the function and reflect those changes outside the function, we use pointers. By passing the address of the variable (a pointer) to the function, the function can directly modify the original variable in memory, not just a copy of it.
- **Memory Efficiency**: When dealing with large data structures or arrays, passing them by value can be memory-intensive because it creates copies. By passing pointers to these structures or arrays, we avoid unnecessary memory consumption and improve the program's efficiency.
- **Dynamic Memory Allocation**: Pointers are essential when working with dynamically allocated memory. Functions that allocate memory (e.g., using malloc) return pointers to the allocated memory, allowing us to access and manage the allocated memory effectively (Dynamic Memory Allocation).
- **Sharing Data Across Functions**: Pointers enable sharing data between different functions without the need for global variables. Functions can access and modify the same data by using pointers, promoting modularity and encapsulation.
- **Function Return Multiple Values**: C functions can return only a single value, but using pointers as function arguments, we can return multiple values from a function.
- **Data Structures**: Pointers are widely used in creating complex data structures like linked lists, trees, and graphs, where each element points to the next or previous element.

3.4.1 Constant Pointers

Like any other data type, pointers can also be defined as constant using the const keyword. When a pointer is defined as constant, it means that the memory address it points to cannot be changed, making it a constant pointer.

Let's consider an example to illustrate the concepts. This example is taken from Prof. Barak Shoshany:

```
#include <stdio.h>

int main() {
 int variable1 = 10;
 double variable2 = 3.14;

 const int const1 = 20;
 const double const2 = 2.71;

 int *variable_pointer_to_variable = &variable1;
 int *const const_pointer_to_variable = &variable2;

 const int *variable_pointer_to_const = &const1;
 const int *const const_pointer_to_const = &const2;

 // Allowed: var is not const, so can be changed.
 variable1 = 12;
 // Not Allowed: con is const, so cannot be changed.
 const1 = 30;

 // Allowed: pointer itself is not const, so can be changed.
 variable_pointer_to_variable = &variable2;
 // Allowed: variable pointed to is not const, so can be changed.
 *variable_pointer_to_variable = 30;

 // Not Allowed: pointer itself is const, so cannot be changed.
 const_pointer_to_variable = &variable1;
 // Allowed: variable pointed to is not const, so can be changed.
 *const_pointer_to_variable = 30;

 // Allowed: pointer itself is not const, so can be changed.
```

```
    variable_pointer_to_const = &const2;
    // Not Allowed: variable pointed to is const, so cannot be
       changed.
    *variable_pointer_to_const = 50;

    // Not Allowed: pointer itself is const, so cannot be changed.
    const_pointer_to_const = &const1;
    // Not Allowed: variable pointed to is const, so cannot be
       changed.
    *const_pointer_to_const = 30;
}
```

- `type <variable>`: This declares a variable of type `type`. The value of the variable can be modified throughout its lifetime!
- `const type <variable>`: This declares a constant variable of type `type`. The value of the variable cannot be modified after it is initialized.
- `type *<pointer>`: This declares a pointer variable of type `type*`. The pointer can store the memory address of a variable of type `type`, and the value pointed to by the pointer can be modified.
- `type *const <pointer>`: This declares a constant pointer variable of type `type*`. The memory address stored in the pointer cannot be modified after it is initialized, but the value pointed to by the pointer can be modified.
- `const type *<pointer>`: This declares a pointer variable of type `const type*`. The pointer can store the memory address of a variable of type `const type`, and the value pointed to by the pointer cannot be modified.
- `const type *const <pointer>`: This declares a constant pointer variable of type `const type*`. The memory address stored in the pointer cannot be modified after it is initialized, and the value pointed to by the pointer cannot be modified.

Tips! There is no difference between:
`const int *variable_pointer_to_const`
and
`int const *variable_pointer_to_const`
In both cases, `const` is before `*`, indicating that the variable itself is constant NOT the pointer.

3.4.2 Pointers and Arrays

In C, arrays have a close relationship with pointers. When an array is declared, it automatically creates a pointer that points to the memory location of its first element. This means that arrays are essentially a contiguous block of memory, and each element in the array can be accessed using pointer arithmetic.

Here's a C code that demonstrates initializing an array, printing out the address for each element, and printing the value of each element:

```c
#include <stdio.h>

int main() {
 int arr[] = {10, 20, 30, 40, 50};  // Initializing an array with
     values

 // Printing the address and value of each element in the array
 for (int i = 0; i < 5; i++) {
  printf("arr[%d] = %d with address %p\n", i, arr[i], &arr[i]);
 }
}
```

In this example, we declare an array `arr` with five elements. We then use a loop to iterate through each element of the array. By using the `&` operator, we can get the address of each element and print it using `%p` format specifier. Additionally, we print the value of each element using `%d` format specifier. The output shows that each element of the array is stored at a unique memory address, and we can access their values using the pointers to those addresses.

Array of Strings

In C, strings are represented as arrays of characters, where each character represents a single element of the string. A C string is terminated with a null character `'\0'`, which indicates the end of the string. Therefore, a string of length n requires an array of n+1 characters, with the last element being the null character.

Both of the following statements are valid ways to initialize a string in C:

```c
char date[] = "July 24"; // Initializing using an array
char *date = "July 24";  // Initializing using a pointer
```

In the first example, date is declared as an array of characters, and the compiler automatically determines the size of the array based on the length of the string literal "July 24". Here is how `date` will look like:

J	u	l	y		2	4	\0	date

In the second example, date is declared as a pointer to a character, and it is assigned the address of the string literal "July 24". Note that in this case, the size of the array is not explicitly specified, as the compiler automatically allocates the appropriate memory to store the string literal. To find the last element of an array you can use a the following code:

```
#include <stdio.h>

// Function to find the end of a string using a pointer
void findEndOfString(const char *str) {
 while (*str != '\0') {
  str++; // Move the pointer to the next character
 }
 // Print the last character of the string
 printf("End of string: %c\n", *(str - 1));
}

int main() {
 // Single string
 const char myString[] = "Hello, this is a test string.";

 // Send the string as a pointer to the function to find the end
 findEndOfString(myString);
}
```

Or if you want to find the length of a string something like strlen function you can use the following code:

```
#include <stdio.h>

// Function to find the end of a string using a pointer with a
   for loop
int findEndOfString(const char *str)
{
 int n;
 for (n = 0; *str != '\0'; str++)
 {
  n++;
 }
```

```
 return n; // Return the last character of the string
}

int main()
{
 // Single string
 const char myString[] = "Hello, this is a test string.";

 // Send the string as a pointer to the function to find the end
 int lastCharacter = findEndOfString(myString);
 printf("End of string: %d\n", lastCharacter);

 // this one also counts the last character which is a null
    character '\0'
 int length_str = sizeof(myString)/sizeof(myString[0]);
 printf("Size of string: %d\n", length_str);
}
```

Both codes use `'\0'` to find the last character. I can do the same by using an array notation for function parameter:

```
#include <stdio.h>

// Function to find the end of a string using an array
int findEndOfString(const char str[]) {
 int n;
 for (n = 0; str[n] != '\0'; n++) {
  // Loop until the null terminator is found
  // it was better to use while
  // I did to make it similar to the previous version
 }
 return n - 1; // Return the index of the last character in the
    string
}

int main() {
 // Single string
 const char myString[] = "Hello, this is a test string.";

 // Send the string as an array to the function to find the end
```

```
    int lastCharacter = findEndOfString(myString);

    printf("End of string: %c\n", myString[lastCharacter]);
}
```

The difference between `(const char str[])` and `(const char *str)` lies in how the function can access the characters of the string.

`(const char str[])`: This is an array notation for function parameters, also known as array notation for passing strings. When you pass a string as `const char str[]`, the function treats it as an array of characters. Inside the function, you can access the characters of the string using array indexing `(str[n])`. The compiler will automatically adjust the pointer to the first element of the array, so you can use it as if it were a regular array.

`(const char *str)`: This is a pointer notation for function parameters, also known as pointer notation for passing strings. When you pass a string as `const char *str`, the function treats it as a pointer to the first character of the string. Inside the function, you can access the characters of the string using pointer arithmetic (`*(str + n)` or `str[n]`). In this case, you explicitly use pointer arithmetic to traverse the characters.

Both notations allow you to pass strings to functions, and both versions of the function will work correctly to find the end of the string. In both the pointers are sent and there will not be another copy of array (waste of memory)in the function. You can used **GDB** to print out the addresses. The choice between array notation and pointer notation is mostly a matter of preference and coding style. Array notation may be more intuitive and familiar to some developers, while pointer notation may be preferred for its similarity to working with arrays and dynamic memory. Ultimately, both notations achieve the same goal of accessing the characters of the string within the function.

Let's discuss the `scanf` function and why we use `&` before the string variable when reading input:

```
char str[50];
scanf("%s", &str);
```

The `scanf` function is used for reading input from the user. When using`scanf` to read a string, you need to provide the address of the variable where the string will be stored. Since str is an array of characters, it already represents a memory address. However, when using `scanf`, you need to **explicitly** specify the address using the `&` (address-of) operator.

In this case, `&str` represents the address of the first element of the `str` array, which is the starting address where the string entered by the user will be stored. The `scanf` function reads

characters from the standard input and stores them in the memory pointed to by `&str`, until it encounters a whitespace character (space, tab, or newline), effectively reading a single word (no spaces) as input. The null character `'\0'` is automatically appended at the end of the input, ensuring that the array `str` is properly terminated as a C string.

3.4.3 Pointers as Arguments of a Function

It was mentioned that one of the benefits of Pointers is **Passing by Reference**, allowing us to modify a variable in another scope. In Section Variable Scope in Functions, we had a function `printNumber()` that declares a local variable number inside the function. We also have a variable `number` declared in the `main()` function. Let's modify the code to explain how using pointers can help us modify a value passed to a function.

```
#include <stdio.h>

void printNumber(int *ptr) {
 // Using a pointer to modify the value passed to the function
 *ptr = 10;

 printf("Number inside the function: %d\n", *ptr);
}

int main() {
 int number = 5;  // Variable declared inside the main function

 printf("Number inside the main function: %d\n", number);

 // Pass the address of 'number' to the function
 printNumber(&number);

 // The value of 'number' has been modified by the function
 printf("Number after the function call: %d\n", number);
}
```

In this modified code, we have made the following changes:

The function `printNumber()` now takes a pointer to an integer (`int *ptr`) as an argument instead of having a local variable. By passing the address of number to this function, we can access and modify the original number variable inside the `main()` function.

Inside the `printNumber()` function, we use the pointer `ptr` to modify the value of number to 10. We do this by de-referencing the pointer using `*ptr`, which gives us access to the value stored at the address pointed to by the pointer.

After the function call, we print the value of number again in the `main()` function. Since we passed the address of number to the `printNumber()` function and modified it using the pointer, the value of number has been changed to 10 even outside the `printNumber()` function. The result must be:

```
Number inside the main function: 5
Number inside the function: 10
Number after the function call: 10
```

Using pointers allows us to directly access and modify the original variable's value inside the function, which can be helpful when we need to modify the value of a variable passed to a function. This is particularly useful when we want to achieve pass-by-reference behavior in C, as C functions are typically pass-by-value by default.

Let's try another example. This C code passes a double value and pointers to a function, performs some calculations inside the function, and updates the values pointed to by the pointers:

```c
#include <stdio.h>

void calculate(double num, double *square, double *cube) {

 // update the value stored at square pointer
 *square = num * num;

 // update the value stored at cube pointer
 *cube = num * num * num;
}

int main() {
 // Initial value
 double number = 5.0;

 // Variables to store the results
 double result_square, result_cube;

 // Call the function to calculate the square and cube
 calculate(number, &result_square, &result_cube);
```

```
    // Print the results
    printf("Number: %.2f\n", number);
    printf("Square: %.2f\n", result_square);
    printf("Cube: %.2f\n", result_cube);
}
```

Let's go through the code using GDB (LLDB). So compile the code with `gcc -g -o pedram pedram.c`. Run the executable object with `gdb ./pedram` (lldb ./pedram). Define the following breakpoints. If you have copied the code with same format of spaces between lines, the numbering here must be the same as yours, otherwise you need to change the number mentioned here.

- `break 20` at `calculate(number, &result_square, &result_cube);` right before calling the function.
- `break 6` at `*square = num * num;` right before updating the value stored at the address pointed by pointer `square`.
- `break 9` at `*cube = num * num * num` before updating the value stored at the address pointed by pointer `cube`.

Enter `run` to start debugging the program. Follow the output in **my computer** line by line. **This is really important!**

```
Breakpoint 1, main () at pedram.c:20
20          calculate(number, &result_square, &result_cube);
$1 = 6.9533558070931648e-310
(gdb) p result_cube
$2 = 4.9406564584124654e-322
(gdb) n
Breakpoint 2, calculate (num=5, square=0x7fffffffbef0, cube=0x7fffffffbef8)
6               *square = num * num;
(gdb) p num
$3 = 5
(gdb) p square
$4 = (double *) 0x7fffffffbef0
(gdb) p cube
$5 = (double *) 0x7fffffffbef8
(gdb) p *square
$6 = 6.9533558070931648e-310
(gdb) p *cube
```

```
$7 = 4.9406564584124654e-322
(gdb) n

Breakpoint 3, calculate (num=5, square=0x7fffffffbef0, cube=0x7fffffffbef8)
9           *cube = num * num * num;
(gdb) p square
$8 = (double *) 0x7fffffffbef0
(gdb) p *square
$9 = 25
(gdb) n
10      }
(gdb) p *cube
$10 = 125
(gdb) n
main () at pedram.c:23
23          printf("Number: %.2f\n", number);
(gdb) p result_square
$11 = 25
(gdb) p result_cube
$12 = 125
(gdb) c
Continuing.
Number: 5.00
Square: 25.00
Cube: 125.00
[Inferior 1 (process 95816) exited normally]
(gdb) q
```

Let's go through it step by step:

- The program starts at `main()` on line 20. It halts at Breakpoint 1 on line 20, where the call to the `calculate()` function is about to be made.
- When we print the values of `result_square` and `result_cube`, GDB shows that their values are initially uninitialized and contain garbage values. The garbage values are displayed in scientific notation.
- We proceed with the program execution using `n` (next) command, and it reaches Breakpoint 2 inside the `calculate()` function.

- When we print the values of `num`, `square`, and `cube`, GDB shows the correct values of the arguments passed to the function. `num` is 5, and square and cube are pointers to `result_square` and `result_cube` in the `main()` function.
- Printing `*square` and `*cube` shows that they still contain the initial garbage values.
- We continue the execution using `n`, and it reaches Breakpoint 3 inside the `calculate()` function.
- After calculating the square and cube and updating the values using pointers, we print `*square` again, which now correctly shows 25, the square of `num`. Similarly, `*cube` correctly shows 125, the `cube` of `num`.
- We continue the execution using `n`, and it returns to `main()`.
- After the `calculate()` function call, we print the values of `result_square` and `result_cube` in `main()`, which now correctly show 25 and 125, respectively.
- The program continues execution using `c` (continue) command, and it reaches the end. The final output shows the values of number, `result_square`, and `result_cube`, confirming that the calculations were performed correctly.

How we can pass an array to a function?

1. Passing one dimensional arrays: Here's an example of passing a one-dimensional array to a function using pointers and printing the value and the address of each element inside the function:

```
#include <stdio.h>

void printArrayElements(int *arr, int size) {
 for (int i = 0; i < size; i++) {
  printf("Element %d: Value = %d, Address = %p\n", i, arr[i], &
     arr[i]);
 }
}

int main() {
 int array[] = {10, 20, 30, 40, 50};
 int size = sizeof(array) / sizeof(array[0]);

 printf("Array elements in main function:\n");
 printArrayElements(array, size);
```

```
}
```

In this example, we define a function called `printArrayElements`, which takes a pointer to an integer array `arr` and the `size` of the array size. Inside the function, we use a `for` loop to iterate through the array and print the value and address of each element using the pointer arithmetic `&arr[i]`.

In the `main` function, we declare an integer array `array` and initialize it with some values. We then calculate the size of the array using `sizeof`, and call the `printArrayElements` function, passing the array and its size as arguments.

The output shows the value and address of each element in the array, printed inside the `printArrayElements` function.

Output:

```
Array elements in main function:
Element 0: Value = 10, Address = 0x7ffd9b1aa940
Element 1: Value = 20, Address = 0x7ffd9b1aa944
Element 2: Value = 30, Address = 0x7ffd9b1aa948
Element 3: Value = 40, Address = 0x7ffd9b1aa94c
Element 4: Value = 50, Address = 0x7ffd9b1aa950
```

2. Passing multi dimensional arrays: To pass a 2-dimensional array to a function using pointers and print out the value and address of each element inside the function, you can use pointer-to-pointer notation to handle the array. Here's an example:

```
#include <stdio.h>

// Function to print the value and address of each element in a
   2-dimensional array
void printArrayElements(int rows, int cols, int arr[rows][cols])
    {
 for (int i = 0; i < rows; i++) {
  for (int j = 0; j < cols; j++) {
   printf("Element [%d][%d]: Value = %d, Address = %p\n", i, j,
      arr[i][j], &arr[i][j]);
  }
 }
}

int main() {
```

```
    int array[][3] = {{10, 20, 30}, {40, 50, 60}, {70, 80, 90}};
    int rows = sizeof(array) / sizeof(array[0]);
    int cols = sizeof(array[0]) / sizeof(array[0][0]);

    printf("Array elements in main function:\n");
    printArrayElements(rows, cols, array);
}
```

In this example, we declare a 2-dimensional array `array` in the `main` function and initialize it with some values. We then calculate the number of rows and columns in the array. Next, we call the `printArrayElements` function, passing the 2-dimensional array, the number of rows, and the number of columns as arguments.

The `printArrayElements` function takes a pointer to an array of integers as its first argument, which allows it to receive a 2-dimensional array of any size. Inside the function, we use nested `for` loops to iterate through the 2-dimensional array and print the value and address of each element using pointer arithmetic `&arr[i][j]`.

The output shows the value and address of each element in the 2-dimensional array, printed inside the `printArrayElements` function.

It is important to know, the function `printArrayElements` receives the 2-dimensional array `arr` as a pointer to its first element. This means that `arr` is not a copy of the original `array` but rather a pointer to the same memory location where array is stored. The address of the first element of the 2-dimensional array is passed to the function.

When you pass a multi-dimensional array to a function, the compiler treats it as a pointer to an array. In this case, `arr` is treated as a pointer to an array of `cols` integers, where each element of this array is itself an array of `int`. The size of this array is not known to the function, so the dimensions `rows` and `cols` are required to be passed as separate arguments.

Therefore, modifications made to the `arr` variable inside the `printArrayElements` function will directly affect the original `array` in the `main` function because they point to the same memory location.

Run the following code using GDB and set breakpoints and see the address of elements inside `printArrayElements` and `main` separately, to check if I am right!

```
#include <stdio.h>

// Function to print the value and address of each element in a
   2-dimensional array
void printArrayElements(int rows, int cols, int arr[rows][cols])
```

```
    {
 for (int i = 0; i < rows; i++) {
  for (int j = 0; j < cols; j++) {
   int* elementPtr = &arr[i][j]; // Pointer to the element
   printf("Element [%d][%d]: Value = %d, Address = %p\n", i, j,
      *elementPtr, elementPtr);

   // Modify the element using the pointer
   *elementPtr = *elementPtr * 2; // Doubling the value of the
      element
  }
 }
}

int main() {
 int array[][3] = {{10, 20, 30}, {40, 50, 60}, {70, 80, 90}};
 int rows = sizeof(array) / sizeof(array[0]);
 int cols = sizeof(array[0]) / sizeof(array[0][0]);

 printf("Array elements in main function before modification:\n"
    );
 printArrayElements(rows, cols, array);

 printf("\nArray elements in main function after modification:\n
    ");

 for (int i = 0; i < rows; i++) {
  for (int j = 0; j < cols; j++) {
   printf("Element [%d][%d]: Value = %d, Address = %p\n", i, j,
      array[i][j], &array[i][j]);

  }
 }
}
```

Warning! To implement the function you need to pass `rows` and `cols` before `arr[rows][cols]` like:
`void printArrayElements(int rows, int cols, int arr[rows][cols])`.
But the following format will cause errors during compile:
`void printArrayElements(int arr[rows][cols], int rows, int cols)`.
Why? Try it and see what happens!

Tips! Since the arrays are sent to function as pointers not as a copy, I still see the same results if instead of

```
int* elementPtr = &arr[i][j]; // Pointer to the element
printf("Element [%d][%d]: Value = %d, Address = %p\n", i, j,
    *elementPtr, elementPtr);
*elementPtr = *elementPtr * 2; // Doubling the value of the
   element
```

we use

```
printf("Element [%d][%d]: Value = %d, Address = %p\n", i, j,
    arr[i][j], &arr[i][j]);
arr[i][j] = arr[i][j] * 2; // Doubling the value of the
   element
```

Need more example!? Write a C program that prompts the user to enter 5 double numbers and stores them in an array. Define a constant n with a value of 5 to indicate the size of the array. After reading the numbers, the program should find and print the minimum and maximum values among the entered numbers using a separate function.

Solution:

```
#include <stdio.h>

#define n 5

// Function to find the minimum and maximum values in an array
void findMinMax(const double arr[], int size, double* min, double
   * max) {
 *min = arr[0];
 *max = arr[0];
```

```
    for (int i = 1; i < size; i++) {
      if (arr[i] < *min) {
        *min = arr[i];
      }

      if (arr[i] > *max) {
        *max = arr[i];
      }
    }
}

int main() {
    double numbers[n];
    double max, min;

    printf("Enter %d double numbers:\n", n);

    // Read numbers from the user and store them in the array
    for (int i = 0; i < n; i++) {
      scanf("%lf", &numbers[i]);
    }

    // Call the function to find the minimum and maximum values
    findMinMax(numbers, n, &min, &max);

    // Print the result
    printf("Minimum value: %.2lf\n", min);
    printf("Maximum value: %.2lf\n", max);
}
```

3. Arrays of strings:

In C, an array of strings is a two-dimensional array of characters, where each element of the array represents a string. Each string is a sequence of characters terminated by a null character `'\0'`. The array is organized in rows, where each row represents a separate string. Since each array might have combination of long and short stings we need to save them in a two dimensional array with different length in each row, which is called **ragged array**.

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

```
int main()
{
 // Ragged array of strings
 char names[][11] = {"Anne-Marie", "Anna", "Mahmoud", "Kian", "
    Raouf", "Nikki"};

 // Accessing and printing the elements and their lengths
 for (int i = 0; i < 6; i++)
 {
  int length = 0;
  while (names[i][length] != '\0')
  {
   length++;
  }
  printf("Name %d: %s (Length: %d) - After '\\0': '%c'\n", i + 1,
      names[i], length, names[i][length]);
 }
}
```

In this example the size of stings is between 4 to 10, and this is how it is saved:

A	n	n	e	-	M	a	r	i	e	\0
A	n	n	a	\0	\0	\0	\0	\0	\0	\0
M	a	h	m	o	u	d	\0	\0	\0	\0
K	i	a	n	\0	\0	\0	\0	\0	\0	\0
R	a	o	u	f	\0	\0	\0	\0	\0	\0
N	i	k	k	i	\0	\0	\0	\0	\0	\0

The given code is not ragged because it uses a 2-dimensional array to store the strings. In a ragged array, the array elements are pointers, and each element can point to an array of different sizes. In the provided code, names is a 2-dimensional array of characters, and all the strings have a fixed length of 11 characters (including the null-terminating character `'\0'`).

A **ragged array** is a two-dimensional array where the rows can have different sizes (different number of elements). In contrast, a regular two-dimensional array has fixed sizes for all rows and columns. Ragged arrays are useful when you have data with varying sizes or when you want to

optimize memory usage. For example, if you have a table of data with different lengths of rows, a ragged array can efficiently store this data without wasting memory on empty elements. For example:

```
#include <stdio.h>

int main()
{
 // Ragged array of strings
 char *names[] = {"Anne-Marie", "Anna", "Mahmoud", "Kian", "Raouf
    ", "Nikki"};

 // Accessing and printing the elements and their lengths
 for (int i = 0; i < 6; i++)
 {
  int length = 0;
  while (names[i][length] != '\0')
  {
   length++;
  }
  printf("Name %d: %s (Length: %d) - After '\\0': '%c'\n", i + 1,
      names[i], length, names[i][length]);
 }
}
```

Here's an explanation of the code:

`char *names[]`: This declares an array of pointers to characters. Each element of names is a pointer that can point to the first character of a string.

`"Anne-Marie", "Anna", "Mahmoud", "Kian", "Raouf", "Nikki";`: This initializes the names array with string literals. Each string literal is stored in a separate memory location, and the pointers in the names array point to the first character of each string.

The for loop iterates through each element of the names array.

Inside the loop, length is initialized to 0. The while loop is used to find the length of each string in the names array. It continues until it reaches the null-terminating character `'\0'`, which marks the end of the string.

After finding the length of the string, the code prints the details using `printf`. This line prints the information for each string. It displays the name's index, the string itself, its length, and the

character that appears after the null-terminating character. Note that `names[i][length]` is used to check the character after `'\0'`! This is how the array is save while each pointer to the row will point only to the first element of the row (`*names[i]`).

A	n	n	e	-	M	a	r	i	e	\0
A	n	n	a	\0						
M	a	h	m	o	u	d	\0			
K	i	a	n	\0						
R	a	o	u	f	\0					
N	i	k	k	i	\0					

Output:

```
Name 1: Anne-Marie (Length: 10) - After '\0': ''
Name 2: Anna (Length: 4) - After '\0': ''
Name 3: Mahmoud (Length: 7) - After '\0': ''
Name 4: Kian (Length: 4) - After '\0': ''
Name 5: Raouf (Length: 5) - After '\0': ''
Name 6: Nikki (Length: 5) - After '\0': ''
```

The effects of using a ragged array are that the individual strings (sub-arrays) can have different lengths, making it more flexible in handling data with varying sizes.

The difference between the first code and this ragged array is that the ragged array code used a 1-dimensional array of pointers, and each pointer pointed to a separate memory location holding a string of different lengths. In contrast, the updated code uses a 2-dimensional array of characters, where each row represents a string with a fixed length. To see the effects of one dimensional array in ragged code, we can print out `names[i][length+1]` which is the character after null-terminating character. In the first one, it still prints out `''` (`'\0'`). But in the ragged array it gives us the first character of of the next string:

```
Name 1: Anne-Marie (Length: 10) - After '\0': 'A'
Name 2: Anna (Length: 4) - After '\0': 'M'
Name 3: Mahmoud (Length: 7) - After '\0': 'K'
Name 4: Kian (Length: 4) - After '\0': 'R'
Name 5: Raouf (Length: 5) - After '\0': 'N'
Name 6: Nikki (Length: 5) - After '\0': ''
```

3.4.4 Pointers as Return Values

In C, a function can return a pointer as its return value. This allows the function to dynamically allocate memory on the heap and return a pointer to that memory. When using a pointer as a return value, you should be careful to manage the memory properly to avoid memory leaks or accessing invalid memory locations.

Here's an example of a function that returns a pointer to the minimum of two integers:

```
#include <stdio.h>

int *min(int *n1, int *n2) {
 return (*n1 < *n2) ? n1 : n2;
}

int main() {
 int num1 = 10;
 int num2 = 5;
 int *result = min(&num1, &num2);

 printf("The minimum value is: %d\n", *result);
}
```

This will give us:

```
The minimum value is: 5
```

3.5 Dynamic Memory Allocation

3.5.1 Allocating Memory with `malloc` and `calloc`

Dynamic memory allocation in C allows you to request memory from the heap at runtime. This is particularly useful when you need to allocate memory for data whose size is not known at compile time or when you want to manage memory manually.

The stack is a region of memory that is used for storing function call frames and local variables with the range of a few megabytes to tens of megabytes. Memory allocation and de-allocation on the stack are handled automatically by the compiler. The size of the stack is usually limited and defined at compile-time, and exceeding this limit can lead to a stack overflow.

On the other hand, the size of the heap is limited by the memory that machine has (check yours

with `free -h`) and can grow or shrink dynamically during the program's execution. It allows for more flexible memory allocation and deallocation compared to the stack.

So to answer the question why we need it,

- Dynamic memory allocation allows you to allocate memory for data structures such as arrays, linked lists, and trees without knowing their size in advance.
- It is essential when dealing with user input, files, or network data, where the size of the data may vary and is determined at runtime.
- Dynamic memory allocation provides flexibility in memory management and helps avoid wasting memory or running out of memory in certain situations.

To perform memory allocation in C, you need to use the functions provided by the `<stdlib.h>` header. Here's a simple example of memory allocation in C:

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

int main() {
 int n;
 printf("Enter the number of elements: ");
 scanf("%d", &n);

 // Allocate memory for an array of integers
 int *arr = malloc(n * sizeof(int));

 // Check for allocation failure
 if (arr == NULL) {
  printf("Memory allocation failed. Exiting the program.\n");
  return 1;
 }

 // Print the number of elements and the size of each element in
    bytes
 printf("Number of elements: %d\n", n);
 printf("Size of each element: %zu bytes\n", sizeof(*arr));

 // Print the value in the first element (garbage value, since it
    's not initialized)
```

```
    printf("Value in the first element: %d\n", arr[0]);

    // Deallocate memory
    free(arr);
}
```

In this example, we use the `malloc` function to allocate memory for an array of integers. We first ask the user to input the number of elements they want in the array. We then use `malloc` to allocate memory for the array, and check if the allocation was successful by ensuring that `arr` is not equal to `NULL`.

We print the number of elements, the size of each element in **bytes** using `sizeof(*arr)`, and the value stored in the first element of the array (which will be zero because `malloc` initialized all the elements with zero). Finally, we de-allocate the memory using `free(arr)` to release the memory back to the system.

Warning! Dynamic memory allocation is a very common reason behind many bugs and crashes in programs, to avoid them:

- **Always check for allocation failure**: When using dynamic memory allocation functions like malloc, it is essential to check if the allocation was successful or if it returned a NULL pointer. If the allocation fails, it means that there is not enough available memory, and attempting to access or use that memory could lead to unexpected behavior or crashes.
- **Always de-allocate memory** using `free` when you are done with it: Dynamically allocated memory is not automatically released when it is no longer needed. It is the programmer's responsibility to free the memory using the free function once it is no longer required. Failure to do so will result in memory leaks, where memory is allocated but not released, leading to a gradual loss of available memory and potential performance issues.
- **Never try to use memory before allocating it, or after freeing it**: Attempting to access memory before it has been allocated or after it has been freed is undefined behavior. In some cases, it might result in accessing random or garbage data, leading to unexpected program behavior or crashes.

Tips! There is no difference between
1. `int *arr = malloc(n * sizeof(int));`: This line directly assigns the return value of `malloc` to the **pointer** `arr`. In this case, the result of `malloc` is **implicitly cast** to the type `int*`, as `malloc` returns a `void*` pointer. In C, there's an implicit conversion from `void*` to other pointer types, so this syntax is valid and often used in C code.
and
2. `int *arr = (int *)malloc(n * sizeof(int));`: This line **explicitly** casts the return value of `malloc` to an `int*`. This is done to make the code more explicit and self-documenting. In some codebases or for some developers, this kind of explicit casting is preferred to clearly indicate the type of pointer being used.

In the code int `*arr = (int *)malloc(n * sizeof(int));`, the return value of `malloc` is **a pointer to the first element** of the dynamically allocated memory block. The variable `arr` is also a pointer that holds the memory address of the first element of the dynamically allocated integer array.

So, after the allocation, `arr` is a pointer that points to the memory location where the first element of the dynamically allocated integer array is stored. The elements themselves are stored in memory **consecutively**, starting from the address pointed to by `arr`

This means to access **the values of elements**:

1. you can use `arr[i]`:
```
for (int i = 0; i < n; i++) {
 printf("arr[%d] = %d\n", i, arr[i]);
}
```
2. or you can use `*(arr+i)`
```
for (int i = 0; i < n; i++) {
 printf("*(arr + %d) = %d\n", i, *(arr + i));
}
```

And to access **the address of elements**:

1. you can use `&arr[i]`:
```
for (int i = 0; i < n; i++) {
 printf("Address of arr[%d]: %p\n", i, &arr[i]);
}
```

2. or you can use `arr + i`

```
for (int i = 0; i < n; i++) {
 printf("Address of arr[%d]: %p\n", i, arr + i);
}
```

I can also allocate the memory using `calloc(number, element_size)` function, instead of `malloc(tot_size)`. `calloc` allocates the memory for an array with the number of `number` elements and the memory size of `element_size` for each element. So you can say $tot\_size = number \times element\_size$. Here also **all elements are initialized to zero**, and a pointer is returned to the from the function:

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

int main() {
 int n;
 printf("Enter the number of elements: ");
 scanf("%d", &n);

 // Allocate memory for an array of integers using calloc
 int *arr = calloc(n, sizeof(int));

 // Check for allocation failure
 if (arr == NULL) {
  printf("Memory allocation failed. Exiting the program.\n");
  return 1;
 }

 // Print the number of elements and the size of each element in
    bytes
 printf("Number of elements: %d\n", n);
 printf("Size of each element: %zu bytes\n", sizeof(*arr));

 // Print the value in the first element (initialized to zero by
    calloc)
 printf("Value in the first element: %d\n", arr[0]);

 // Deallocate memory
 free(arr);
```

```
}
```

3.5.2 Stack Overflow

Like it was mentioned, A stack overflow occurs when the stack size is exceeded due to the allocation of too much memory on the stack. The stack is a region of memory used to store function call information, local variables, and other data related to function calls. It has a fixed size, and if you allocate too much memory on the stack, it can lead to a stack overflow.

Run `ulimit -s` command on your terminal to see the stack memory size you have on your machine. In my computer it is `8192` KB (8 MB or 8192 * 1024 bytes). In Using `printf` and `limits.h` to Get the Limits of Integers, we found out what is the memory size taken by each integer value saved on the memory. Take a look at the following example:

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

void allocateOnStack(const int size) {
 int largeArray[size]; // Attempt to allocate the array on the
    stack
 printf("Stack memory allocation Succeed!\n");
}

void allocateOnHeap(const int size) {


 int* largeArray = (int*)malloc(size * sizeof(int)); // Allocate
    the array on the heap
 // Check if memory allocation is successful
 if (largeArray != NULL) {
  printf("Heap memory allocation Succeed!\n\n");
  free(largeArray);
 } else {
  printf("Memory allocation failed!\n\n");
 }
}

int main() {
```

```
    // run this to see how much size each ‘int‘ takes in your
       machine
    // mine Size of int: 4 bytes
    printf("Size of int: %zu bytes\n", sizeof(int));

    // so the maximum number of integers I can Save on stack is:
    // 8192 * 1024 / 4 = 2,097,152

    // this show the maximum integer value I can save by type ‘int‘
    printf("Maximum value of int: %d\n\n", INT_MAX);


    int SIZE = 2098100;

    printf("Attempting to allocate on heap...\n");
    allocateOnHeap(SIZE); // This will successfully allocate memory
       on the heap

    printf("Attempting to allocate on stack...\n");
    allocateOnStack(SIZE); // This will cause a stack overflow
}
```

In the provided example, the functions `allocateOnHeap` and `allocateOnStack` are used to allocate memory dynamically on the heap and stack, respectively. The function `allocateOnHeap` uses `malloc` to allocate memory on the heap, which has more available memory compared to the stack.

However, in the function `allocateOnStack`, the code attempts to allocate a large array on the stack using the line `int largeArray[size];`. Since the size of the array (`SIZE = 2098100`) is too large, it exceeds the stack size limit, and a stack overflow occurs.

When I run the program, I will see the following output:

```
Size of int: 4 bytes
Maximum value of int: 2147483647

Attempting to allocate on heap...
Heap memory allocation Succeed!
```

```
Attempting to allocate on stack...
Segmentation fault (core dumped)
```

The size of each integer in my machine is 4 bytes. So on stack I can save up to $8192 \times 1024/4 = 2,097,152$ integer numbers if stack is completely available which is not possible! That's why I have defined `int SIZE = 2098100;` to make sure that stack overflow will happen! you may need to change this value in your machine! But before initializing `SIZE`, I have printed to the maximum value that `int` type can save to make sure integer overflow, discussed in A Simple Example for Integer Overflows, is not going to happen.

The "**Segmentation fault (core dumped)**" error occurs when the program tries to access memory that is outside its allocated region, which happens in the `allocateOnStack` function due to the stack overflow.

To avoid stack overflow, you should avoid allocating large arrays or objects on the stack. Instead, use dynamic memory allocation (e.g. `malloc`, `calloc`) for large data structures that exceed the stack size limit. By doing so, you can utilize the heap's larger memory space and avoid running into stack size limitations

3.5.3 Resize Dynamically Allocated Memory with `realloc`

`realloc` is a function in C that allows you to resize dynamically allocated memory. It is used to change the size of the previously allocated memory block pointed to by pointer. The general format of `realloc` is:

```
void *realloc(void *pointer, size_t new_size);
```

Where `realloc(pointer, new_size)` resizes the memory allocated to `new_size`. Take a look at the following example:

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

int main() {

 int initial_size = 5, reSize = 10;
 int *arr = NULL, *new_arr=NULL;

 // Initial allocation of memory for initial_size integers
 arr = (int*)malloc(initial_size * sizeof(int));
```

```
 if (arr == NULL) {
  printf("Memory allocation failed. Exiting the program.\n");
  return 1;
 }

 printf("Memory Allocation succeeded. The array with %p address:
    \n", arr);
 // Storing values in the array
 for (int i = 0; i < initial_size; i++) {
  arr[i] = i + 1;
  printf("%d ", arr[i]);
 }

 printf("\n");

 // Reallocate memory to fit reSize integers
 new_arr = (int*)realloc(arr, reSize * sizeof(int));

 if (new_arr == NULL) {
  printf("Memory reallocation failed. Exiting the program.\n");
  free(arr); // Free the previously allocated memory before
     exiting
  return 1;
 }

 printf("\nMemory Re-allocation succeeded. The new array with %p
    address: \n", new_arr);
 // Storing values in the array
 for (int i = 0; i < reSize; i++) {
  printf("%d ", new_arr[i]);
 }

 printf("\n");

 // Don't forget to free the allocated memory when it's no longer
     needed
 free(new_arr);

}
```

Here, I have initialized pointers `arr` and `new_arr` with NULL, **null pointer**, to make sure they are not going to randomly point at any address. This is what I see in **my machine**:

```
Memory Allocation succeeded. The array with 0x55ae64c0f2a0 address:
1 2 3 4 5

Memory Re-allocation succeeded. The new array with 0x55ae64c0f6d0 address:
1 2 3 4 5 0 0 0 0 0
```

If we try to resize the allocated memory to a smaller one, let's say `initial_size = 20` and `reSize = 10`, this will be the output:

```
Memory Allocation succeeded. The array with 0x5595bd89f2a0
   address:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Memory Re-allocation succeeded. The new array with 0x5595bd89f2a0
    address:
1 2 3 4 5 6 7 8 9 10
```

Look at the addresses, this time both are the same address. **Why**?

4 More Advanced Topics in C!

4.1 Input and Output

So far, we have discussed two ways to pass the inputs from used to the program:

1. From Terminal during run time: using `scanf`.
2. From command line: with the format of `int main(int argc, char *argv[])` mentioned in Forward Declaration of a Function

In this section we will mainly discuss passing input using a file. In C, a stream is a fundamental concept used for input and output operations. A stream represents a sequence of data that can be read from or written to. Streams can be associated with various sources or destinations, such as files, the standard input (stdin), the standard output (stdout), or the standard error (stderr).

A File pointer is a data type in C that points to a file stream. It is used to manage file input and output operations, allowing you to read data from files or write data to files.

To work with files in C, you need to use the `FILE` data type and the file pointer functions provided by the C standard library. The general format for the `fopen` function, which is used to open a file, is as follows:

```
FILE *fopen(const char *filename, const char *mode);
```

The `fopen` function takes two arguments: `filename` and `mode`. `filename` is a **string** that specifies the name of the file to be opened (or the path to the file), and `mode` is a **string** that specifies the type of file access you want. The `fopen` function **returns a pointer** to a `FILE` structure that represents the file stream if the file is successfully opened, or `NULL` if an error occurs.

The `mode` argument can take various values, including:

- `"r"`: Opens the file for reading. The file must exist, or the operation will fail.
- `"w"`: Opens the file for writing. If the file already exists, its contents are truncated to zero-length, and if it doesn't exist, a new file is created.
- `"a"`: Opens the file for appending. Data is written at the end of the file, and if the file doesn't exist, a new file is created.

Additionally, the mode argument can be extended with the following characters:

- `"b"`: Binary mode. Used for binary file access, like `"rb"` means reading a file in the binary format.
- `"+"`: Allows both reading and writing to the file.

For example, to open a file named "data.txt" for writing in binary mode, you would use the following:

```
FILE *file = fopen("data.txt", "wb");
if (file == NULL) {
 // Error handling if fopen fails
} else {
 // File successfully opened, do operations
}
```

Remember to always check if the file pointer returned by `fopen` is `NULL`, as it indicates that the file couldn't be opened successfully. Proper error handling is essential to ensure that your program behaves correctly when dealing with file I/O operations. Take a look at the following example:

```c
#include <stdio.h>

int main() {
 FILE *file = fopen("A.txt", "r");
 if (file == NULL) {
  printf("Error opening the file.\n");
  return 1;
 }

 double number;
 while (fscanf(file, "%lf", &number) == 1) {
  printf("%.3lf ", number);
  char ch = fgetc(file); // Read the next character
  if (ch == '\n' || ch == EOF) {
   printf("\n");
  }
 }

 fclose(file);
}
```

The `while` loop is used to read the numbers from the file until `fscanf` encounters an error (e.g., reaching the end of the file or encountering invalid input).

`fscanf(file, "%lf", &number)` reads a floating-point number from the file pointed to by file and stores it in the variable number. It returns the number of successfully read items, which will be **1** if a number is read successfully and **0** otherwise. `printf("%.3lf ", number);` prints the value of number with three decimal places.

`char ch = fgetc(file);` reads the next character from the file using `fgetc` and stores it in the variable `ch`.

`if (ch == '\n' || ch == EOF)` checks if the character read (`ch`) is either a newline character (`'\n'`) or the end-of-file marker (`EOF`). If either condition is true, it means that the current line is complete (or the end of the file is reached), and a new line is printed using `printf("\n");`.

```
1.100 2.200 3.300
4.440 5.550 6.660
7.777 8.888 9.999
```

You can I also open a file with the purpose of editing it. For example in this code:

```c
#include <stdio.h>

int main() {
 FILE *inputFile = fopen("A.txt", "r");
 FILE *outputFile = fopen("B.txt", "w"); // Open "B.txt" in write
     mode

 if (inputFile == NULL || outputFile == NULL) {
  printf("Error opening the files.\n");
  return 1;
 }

 double number;
 while (fscanf(inputFile, "%lf", &number) == 1) {
  printf("%.3lf ", number * 2); // Print the multiplied value to
     the console
  fprintf(outputFile, "%.3lf ", number * 2); // Write the
     multiplied value to "B.txt"
  char ch = fgetc(inputFile); // Read the next character
  if (ch == '\n' || ch == EOF) {
   printf("\n");
   fprintf(outputFile, "\n"); // Write a new line to "B.txt"
      after each line
  }
 }

 fclose(inputFile);
 fclose(outputFile); // Close the output file
}
```

The file "B.txt" with the `"w"` mode is opened (If it doesn't exit it will create it). Using `fprintf` with the following general format of

```c
fprintf(FILE *file, <palceholder>, value);
```

the variable `value` can be saved on pointer to the object `file` with the format of `<placeholder>`.

4.2 Structures

Structures, often referred to as "structs," are user-defined data types in C that allow you to group multiple variables of different data types into a single entity. They are used to represent a collection of related data elements, making it easier to organize and manage complex data.

The general format of a struct in C is as follows:

```
struct struct_name {
 data_type member1;
 data_type member2;
 // More members...
};
```

- `struct_name`: The name of the `sstruct` type. It can be any valid identifier in C.
- `sdata_type`: The data type of each member variable in the `struct`.

Structs are commonly used when you want to store different pieces of data together, like representing a point in 2D space (x and y coordinates) or information about a person (name, age, address). Here's a simple C example using structs:

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

// Define the struct
struct Person {
 char name[50];
 int age;
 float height;
};

int main() {
 // Declare and initialize a struct variable
 struct Person person1;
 strcpy(person1.name, "John Brown");
 person1.age = 29;
 person1.height = 1.77;

 // Accessing and printing the struct members
 printf("Name: %s\n", person1.name);
```

```
    printf("Age: %d\n", person1.age);
    printf("Height: %.2f\n", person1.height);
}
```

In this example, we define a `struct` called `Person` with three **members**: `name`, `age`, and `height`. We then declare a `struct` variable `person1` of **type** `Person` and **initialize** its members with values. Finally, we access and print the values of the `struct` members using dot notation (`person1.name`, `person1.age`, and `person1.height`).

Structs are essential in C for organizing and working with complex data, allowing you to create custom data types that can represent various entities in your program.

You can use an **array of structs** to store multiple persons' information. Here's an updated code that demonstrates how to use an array of structs:

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

// Define the struct
struct Person {
    char name[50];
    int age;
    float height;
};

int main() {
    // Declare and initialize an array of struct variables
    struct Person people[3];

    // Initialize the array elements with data
    strcpy(people[0].name, "Walter White");
    people[0].age = 29;
    people[0].height = 1.77;

    strcpy(people[1].name, "Rick Sanchez");
    people[1].age = 25;
    people[1].height = 1.65;

    strcpy(people[2].name, "Mike Wazowski");
    people[2].age = 34;
    people[2].height = 0.85;
```

```
    // Accessing and printing the struct members for each person
    for (int i = 0; i < 3; i++) {
        printf("Person %d:\n", i + 1);
        printf("Name: %s\n", people[i].name);
        printf("Age: %d\n", people[i].age);
        printf("Height: %.2f\n", people[i].height);
        printf("\n");
    }
}
```

In this example, we define an array of struct variables people with a size of 3, allowing us to store information for three persons. We then initialize each array element with data for each person. Finally, we use a loop to access and print the information for each person in the array.

5 Effective Code Development Practices

5.1 Compiler Optimizations

Compiler optimizations are techniques used by compilers to improve the performance of generated machine code. These optimizations aim to make the compiled code execute faster, use less memory, or both. Different optimization levels, often specified through compiler flags, allow developers to balance between compilation speed and the runtime performance of the generated code.

Here are some common optimization levels used in GCC (GNU Compiler Collection) and similar compilers:

- `-O0` (No Optimization): This level turns off almost all optimizations. It is useful during development and debugging because the generated code closely reflects the original source code, making it easier to debug. Compilation is faster, but the resulting code might be slower than optimized versions.
- `-O1` (Basic Optimization): Enables basic optimizations such as inlining functions, simplifying expressions, and eliminating unused variables. Compilation is faster compared to higher optimization levels. Suitable for development and debugging.
- `-O2` (Moderate Optimization): Includes more aggressive optimizations like loop unrolling and strength reduction. Generates faster code at the expense of longer compilation times. A good balance for many projects seeking improved performance.

- `-O3` (High Optimization): Enables more aggressive optimizations than `-O2`. Can result in significantly faster code but with longer compilation times. May not always be beneficial for all programs due to increased compile time.
- `-Ofast` (Fastest, Aggressive Optimization): Combines `-O3` with additional flags that might not be standard-compliant. Can result in the fastest code but may sacrifice some precision in floating-point calculations. Useful for performance-critical applications when strict adherence to standards is not essential.

It's important to note that higher optimization levels might introduce trade-offs, such as increased code size and longer compilation times. Additionally, certain optimizations might affect the behavior of the program in terms of precision, so it's essential to test thoroughly when changing optimization levels.

Let's have an example of matrix multiplication. Compile and run the following code.

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

#define N 2000

void matrixMultiplication(double A[N][N], double B[N][N], double
   C[N][N])
{
 for (int i = 0; i < N; i++)
 {
  for (int j = 0; j < N; j++)
  {
   double t = 0;
   for (int k = 0; k < N; k++)
   {
    t += A[i][k] * B[k][j];
   }
   C[i][j] = t;
  }
 }
}

int main()
{
```

```
  double A[N][N];
  double B[N][N];
  double C[N][N];


  // Initialize matrices A and B
  for (int i = 0; i < N; i++)
  {
   for (int j = 0; j < N; j++)
   {
    A[i][j] = i + j;
    B[i][j] = i - j;
   }
  }

  clock_t start, end;
  double cpu_time_used;

  start = clock();

  // Call the matrix multiplication function
  matrixMultiplication(A, B, C);

  end = clock();
  cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

  printf("CPU Time: %f seconds\n", cpu_time_used);
 }
```

Make sure you run the code before going to the next part. You should see the following error!

```
Segmentation fault (core dumped)
```

I just wanted to show you this error! To find the problem, I would set up multiple break points using GDB, and see run the code step by step, and see between which two breakpoints the segmentation error happens. Try it! The problems is we try to allocate the memory from stack part of memory to keep the matrices `A`, `B`, and `C` but there is not enough space (Stack Overflows).

Below is the modified code with dynamic memory allocations for matrices `A`, `B`, and `C`:

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

#define N 2000

void matrixMultiplication(double **A, double **B, double **C)
{
 for (int i = 0; i < N; i++)
 {
  for (int j = 0; j < N; j++)
  {
   double t = 0;
   for (int k = 0; k < N; k++)
   {
    t += A[i][k] * B[k][j];
   }
   C[i][j] = t;
  }
 }
}

void freeMatrix(double **matrix)
{
 for (int i = 0; i < N; i++)
 {
  free(matrix[i]);
 }
 free(matrix);
}

int main()
{
 // Allocate memory for matrices A, B, and C
 double **A = (double **)malloc(N * sizeof(double *));
 double **B = (double **)malloc(N * sizeof(double *));
 double **C = (double **)malloc(N * sizeof(double *));
```

```
  for (int i = 0; i < N; i++)
  {
   A[i] = (double *)malloc(N * sizeof(double));
   B[i] = (double *)malloc(N * sizeof(double));
   C[i] = (double *)malloc(N * sizeof(double));
  }

  // Initialize matrices A and B
  for (int i = 0; i < N; i++)
  {
   for (int j = 0; j < N; j++)
   {
    A[i][j] = i + j;
    B[i][j] = i - j;
   }
  }

  clock_t start, end;
  double cpu_time_used;

  start = clock();

  // Call the matrix multiplication function
  matrixMultiplication(A, B, C);

  end = clock();
  cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

  printf("CPU Time: %f seconds\n", cpu_time_used);

  // Free allocated memory
  freeMatrix(A);
  freeMatrix(B);
  freeMatrix(C);

  return 0;
}
```

In the section:

```
for (int i = 0; i < N; i++) {
 A[i] = (double *)malloc(N * sizeof(double));
 B[i] = (double *)malloc(N * sizeof(double));
 C[i] = (double *)malloc(N * sizeof(double));
}
```

you are dynamically allocating memory for each row of the matrices A, B, and C. This is necessary because you're working with a 2D array represented as an array of pointers. `A[i]`, `B[i]`, and `C[i]` are pointers to the `i`th row of matrices `A`, `B`, and `C` respectively.

`(double *)malloc(N * sizeof(double))` allocates memory for an array of `N` doubles (a row in your matrices) and returns a pointer to the allocated memory. `A[i] = ...`, `B[i] = ...`, and `C[i] = ...` assign these pointers to the `i`th row of matrices `A`, `B`, and `C`.

If you are working with a contiguous block of memory and want to allocate space for an N by N 2D array without using a loop, you can do it simply in the following way:

```
double (*A)[N] = malloc(N * sizeof(*A));
double (*B)[N] = malloc(N * sizeof(*B));
double (*C)[N] = malloc(N * sizeof(*C));
```

Here, `A`, `B`, and `C` are pointers to arrays of size `N`. This way, you have a contiguous block of memory for each matrix.

The `sizeof(*A)` calculates the size of each row in terms of the whole array, and then `N * sizeof(*A)` allocates space for `N` rows.

Remember to free the memory when you are done:

```
free(A);
free(B);
free(C);
```

Going back to optimization flags. If I ran the code using the following command:

`gcc -O0 -o pedram pedram.c`

The CPU time in my computer is around 73 seconds. If I used the following command:

`gcc -O3 -o pedram pedram.c`

The time taken by CPU is around 47 second in **my computer**.

5.2 Profiling

Profiling is a technique used to analyze the performance of a program, identify bottlenecks, and optimize its execution. It provides insights into the time and resources consumed by different parts of the code, helping developers make informed decisions about where to focus their optimization efforts.

5.2.1 Using gprof

gprof is a profiling tool available for GNU Compiler Collection (GCC).

To use gprof:

1. You need to compile your code with profiling information:

   `gcc -pg -o my_program my_program.c`

2. Run your program:

   `./my_program`

   This generates a file named `gmon.out`.

3. To view the profiling results, use:

   `gprof my_program gmon.out > analysis.txt`

   Open `analysis.txt` to see a detailed breakdown of function execution times and call graphs.

Using **gprof** on the previous program with N=1000, I'll have:

```
Flat profile:

Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
99.44      7.07     7.07        1     7.07     7.07  main
0.42       7.10     0.03                             _start
0.14       7.11     0.01                             frame_dummy
0.00       7.11     0.00        3     0.00     0.00  deregister_tm_clones
```

perf is another powerful performance analysis tool available on Linux systems. I leave studying **perf** to you!

5.2.2 VTune Profiler - From Installation to Case Study

Intel **VTune** Profiler is a performance profiling tool that can be used to analyze and optimize the performance of applications on Intel architecture. Here are the general steps to install Intel VTune Profiler:

1. Download Intel VTune Profiler:

- Go to the Intel VTune Profiler Download Page.
- Sign in with your Intel account or create one if you don't have it.
- Follow the instructions to download the Intel oneAPI Base Toolkit, which includes VTune Profiler.
- Follow the instructions and install Intel oneAPI Base Toolkit. Reach out your TAs if you have any problems with installing it.

2. Configure Environment Variables: After the installation, you might need to set up environment variables. The installer should provide instructions on how to do this. Typically, you will need to add something like the following lines to your shell profile file (e.g., `nano /.bashrc`):

```
source /opt/intel/oneapi/setvars.sh
```

Adjust the path based on your installation location.

3. Verify Installation:

- Open a new terminal or run `source /.bashrc` to apply the changes to the current terminal.
- Check if VTune Profiler is installed and configured:

  ```
  vtune --version
  ```

  This command should display the version number if the installation was successful.

4. Open VTune using `vtune-gui` command in the terminal. When VTune software is opened, it asks you to the following change in order to collect the data: from terminal change "1" to "0". Run the following command and change the value:

```
sudo nano /proc/sys/kernel/yama/ptrace_scope
```

Then, in the "Application" section add your directory before your executable object. For example:

"/home/pedram/MECHTRON2MP3/prac/pedram2"

Where `pedram2` is the executable object file created after compiling the source code `pedram2.c`.

If you run the code using `./pedram2 input1 input2 input3 input4`, in the "Application parameters" section add your inputs like "input1 input2 input3 input4".

Let's run VTune for matrix multiplication we had.

5.2.3 Code Coverage

code overage with gcov

5.3 Documentation

(not written yet)

Emphasize the importance of documentation in code development. Introduce tools like Doxygen or Javadoc for automatic documentation generation. Discuss what makes effective code comments and documentation.

5.4 GitHub and Version Control

(not written yet)

Teach the basics of version control using Git and GitHub. Cover branching strategies, pull requests, and collaborative workflows. Highlight how version control enhances collaboration and code integrity.

6 Parallel Computing - Studying This Section is Optional

Parallel computing is a paradigm that enables the simultaneous execution of multiple tasks, breaking them down into smaller sub-tasks that can be performed concurrently. This approach contrasts with traditional sequential computing, where tasks are executed one after another. Parallel computing is a response to the growing demand for increased computational power to tackle complex problems in science, engineering, and other fields. In this section we just discuss basics of Parallel Computing. If you want to learn more about it, study High Performance Parallel Computing by Prof. Ned Nedialkov.

Consider a simple operation where you want to compute the sum of two vectors aa and bb element-wise:

```
for (int i = 0; i < m; i++) {
 x[i] = a[i] + b[i];
}
```

In this loop, `m` operations are performed sequentially. The time t_1 is taken to execute this loop on a single processor. If we have p processors, we can distribute the workload among them. Each processor handles a portion of the array elements, performing the same operation in parallel. The time taken t_p on pp processors is given by:

$$t_p = \frac{t_1}{p}$$

This is based on the assumption that the workload can be perfectly divided among the processors, and there is no overhead in coordinating their work. In practice, achieving perfect parallelism can be challenging due to issues like load imbalance and communication overhead.

First, we need to distribute segments of arrays a, b, and x among the processors. For example, if you have 4 processors, each processor gets $\frac{m}{4}$ elements of a, b, and x. Each processor performs the addition independently on its segment of the arrays. After the parallel execution, results from all processors need to be combined to get the final result x.

The loop

```
for (int i = 0; i < m; i++)
 x[i + 1] = a[i] * x[i] + b[i];
```

cannot be parallelized straightforwardly due to a dependency between iterations. Each iteration depends on the result of the previous one (`x[i+1]` depends on `x[i]`), creating a data dependency. This is often referred to as a "loop-carried dependency."

Understanding the computer architecture is crucial for effective parallel computing. Here's an overview of key components and their interplay:

1. **Registers**: Registers are small, fast storage locations within the CPU. Instructions and data are typically loaded into registers for processing. They provide extremely fast access but have limited capacity.

2. **Caches**:

   - **L1 cache** is the first level of cache and is usually on the same chip as the processor

cores. It has a small capacity but offers very fast access. It stores frequently accessed instructions and data. It's divided into separate instruction and data caches.

- **L2 cache** is larger but slightly slower than L1. It serves as a backup to L1, providing additional space for frequently used data. Located either on the same chip as the processor or very close.
- Some systems have an **L3 cache** shared among multiple cores or processors.

3. **Memory (RAM)**: RAM (Random Access Memory) is the primary memory of a computer. It has a much larger capacity compared to caches but is slower. When data isn't found in the cache, it's fetched from RAM.

Data moves through this hierarchy, with the fastest and smallest storage (registers) at the top and the largest but slowest (memory) at the bottom. Registers are the fastest storage but have the least capacity. Then, L1 cache is faster than L2, and L2 is faster than RAM. When data is initially accessed, it's brought from main memory to cache. Subsequent accesses to the same data might benefit from the faster cache access, if the data is in cache.

Data isn't transferred between memory and cache one byte at a time; instead, it's done in blocks known as cache lines. A cache line is typically 64 or 128 bytes. This helps utilize spatial locality — if one part of data is accessed, it's likely that nearby data will be accessed soon. All information I have mentioned here about computer architecture in my Linux OS can be seen by `lscpu` or `cat /proc/cpuinfo`.

Let's have an example of matrix multiplication. Compile and run the following code:

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

#define N 2000

void matrixMultiplication(double **A, double **B, double **C)
{
 for (int i = 0; i < N; i++)
 {
  for (int j = 0; j < N; j++)
  {
   double t = 0;
   for (int k = 0; k < N; k++)
   {
    t += A[i][k] * B[k][j];
```

```
    }
    C[i][j] = t;
   }
  }
 }

void freeMatrix(double **matrix)
{
 for (int i = 0; i < N; i++)
 {
  free(matrix[i]);
 }
 free(matrix);
}

int main()
{
 // Allocate memory for matrices A, B, and C
 double **A = (double **)malloc(N * sizeof(double *));
 double **B = (double **)malloc(N * sizeof(double *));
 double **C = (double **)malloc(N * sizeof(double *));

 for (int i = 0; i < N; i++)
 {
  A[i] = (double *)malloc(N * sizeof(double));
  B[i] = (double *)malloc(N * sizeof(double));
  C[i] = (double *)malloc(N * sizeof(double));
 }

 // Initialize matrices A and B
 for (int i = 0; i < N; i++)
 {
  for (int j = 0; j < N; j++)
  {
   A[i][j] = i + j;
   B[i][j] = i - j;
  }
 }
```

```
 clock_t start, end;
 double cpu_time_used;

 start = clock();

 // Call the matrix multiplication function
 matrixMultiplication(A, B, C);

 end = clock();
 cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

 printf("CPU Time: %f seconds\n", cpu_time_used);

 // Free allocated memory
 freeMatrix(A);
 freeMatrix(B);
 freeMatrix(C);

 return 0;
}
```

We can do the same matrix multiplication by a different algorithm (Second Algorithm) like:

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

#define N 2000

void matrixMultiplication(double **A, double **B, double **C) {
 for (int i = 0; i < N; i++) {
  for (int k = 0; k < N; k++) {
   for (int j = 0; j < N; j++) {
    C[i][j] += A[i][k] * B[k][j];
   }
  }
 }
}

void freeMatrix(double **matrix) {
```

```
 for (int i = 0; i < N; i++) {
  free(matrix[i]);
 }
 free(matrix);
}

int main() {
 // Allocate memory for matrices A, B, and C
 double **A = (double **)malloc(N * sizeof(double *));
 double **B = (double **)malloc(N * sizeof(double *));
 double **C = (double **)malloc(N * sizeof(double *));

 for (int i = 0; i < N; i++) {
  A[i] = (double *)malloc(N * sizeof(double));
  B[i] = (double *)malloc(N * sizeof(double));
  C[i] = (double *)malloc(N * sizeof(double));
 }

 // Initialize matrices A and B
 for (int i = 0; i < N; i++)
 {
  for (int j = 0; j < N; j++)
  {
   A[i][j] = i + j;
   B[i][j] = i - j;
  }
 }

 clock_t start, end;
 double cpu_time_used;

 start = clock();

 // Call the matrix multiplication function
 matrixMultiplication(A, B, C);

 end = clock();
 cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
```

```
    printf("CPU Time: %f seconds\n", cpu_time_used);

    // Free allocated memory
    freeMatrix(A);
    freeMatrix(B);
    freeMatrix(C);

    return 0;
}
```

In the **first algorithm**, `C[i][j]` has temporal locality. This means that the same memory location is accessed frequently over a short period. Since it's being updated in a nested loop, it's likely that this value stays in cache.

`A[i][k]` is accessed by rows (Spatial locality). This means that nearby memory locations are accessed in sequence. Given the structure of the nested loops, elements in the same row of `A` are accessed consecutively, which is good for cache performance.

`B[k][j]` is accessed by columns (No locality). This means that elements are scattered across memory. The lack of spatial locality might result in more cache misses.

In the **second algorithm**, `C[i][j]` has Spatial locality. As the computation is done for each element in a row of `C`, spatial locality is exploited.

`A[i][k]` has temporal locality. Elements from the same row are accessed consecutively, which benefits from temporal locality.

`B[k][j]` is accessed by rows (Spatial locality). Elements in the same row of `B` are accessed consecutively.

Run both codes and see **which one is faster**!?

6.1 A Simple Example of Parallel Computing

In the context of parallel computing, two commonly used frameworks are OpenMP and OpenMPI

- **OpenMP (Open Multi-Processing):** OpenMP is typically used for shared-memory parallelism. It extends the capabilities of standard programming languages (like C, C++, and Fortran) to support parallelism using a set of compiler directives. It assumes a shared address space, making communication between threads relatively straightforward. OpenMP is known for its ease of use, especially for parallelizing loops and simple parallel regions.

- **OpenMPI (Message Passing Interface):** OpenMPI is designed for distributed-memory parallelism. It allows processes on different nodes to communicate and coordinate using message-passing protocols. It's suitable for parallel computing across clusters, where each node has its own memory. While powerful, OpenMPI often involves more complex communication strategies compared to OpenMP.

OpenMPI is best suited for large-scale parallelism, where the problem can be decomposed into smaller tasks that can be distributed across **multiple machines**.

OpenMP, on the other hand, is best suited for fine-grained parallelism, where the problem can be decomposed into smaller tasks that can be executed in parallel on a **single machine**.

In this lecture note, our focus is on OpenMP, which is well-suited for shared-memory systems. OpenMP is particularly effective for parallelizing loops and simple code regions. Its simplicity makes it an excellent choice for introductory parallel programming, and it seamlessly integrates into existing codebases. For distributed-memory scenarios, where nodes do not share memory, other approaches like OpenMPI might be more appropriate.

To use OpenMP on Linux, you typically don't need to install anything additional. OpenMP support is often included with the compiler you're using. Here are some common compilers on Linux and the corresponding flags to enable OpenMP:

1. GCC (GNU Compiler Collection):
   - OpenMP is usually included in the GCC compiler.
   - To enable OpenMP, you can use the `-fopenmp` flag when compiling.
   - Example: `gcc -fopenmp my_program.c -o my_program`
2. Clang:
   - Clang, the LLVM compiler, also supports OpenMP.
   - Use the `-fopenmp` flag similarly to GCC.
   - Example: `clang -fopenmp my_program.c -o my_program`
3. Intel Compiler:
   - If you are using the Intel Compiler (icc), OpenMP is supported.
   - No additional flags are required as OpenMP is enabled by default.

For most Linux distributions, these compilers are available through package managers. For instance, on Ubuntu, you can install GCC using:

```
sudo apt-get install build-essential
```

This installs the essential build tools, including GCC, on your system. If you're using a different distribution, the process might vary slightly.

Here's the modified code using OpenMP to parallelize the matrix multiplication and print out the number of threads being used:

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

#define N 2000

void matrixMultiplication(double **A, double **B, double **C) {
 #pragma omp parallel for
 for (int i = 0; i < N; i++) {
  for (int k = 0; k < N; k++) {
   for (int j = 0; j < N; j++) {
    C[i][j] += A[i][k] * B[k][j];
   }
  }
 }
}

void freeMatrix(double **matrix) {
 for (int i = 0; i < N; i++) {
  free(matrix[i]);
 }
 free(matrix);
}

int main() {
 // Set the number of threads
 omp_set_num_threads(4);  // You can adjust this based on your
    system

 // Allocate memory for matrices A, B, and C
 double **A = (double **)malloc(N * sizeof(double *));
 double **B = (double **)malloc(N * sizeof(double *));
```

```
    double **C = (double **)malloc(N * sizeof(double *));

    for (int i = 0; i < N; i++) {
     A[i] = (double *)malloc(N * sizeof(double));
     B[i] = (double *)malloc(N * sizeof(double));
     C[i] = (double *)malloc(N * sizeof(double));
    }

    // Initialize matrices A and B
    for (int i = 0; i < N; i++)
    {
     for (int j = 0; j < N; j++)
     {
      A[i][j] = i + j;
      B[i][j] = i - j;
     }
    }

    clock_t start, end;
    double cpu_time_used;

    start = clock();

    // Call the matrix multiplication function
    matrixMultiplication(A, B, C);

    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("CPU Time: %f seconds\n", cpu_time_used);
    printf("Number of Threads: %d\n", omp_get_max_threads());

    // Free allocated memory
    freeMatrix(A);
    freeMatrix(B);
    freeMatrix(C);

    return 0;
}
```

You need to have OpenMP library installed on your machine. To compile the code add `-fopenmp` flag. OpenMP is a parallel programming API that simplifies parallelism. When you set the number of threads using `omp_set_num_threads` and use directives like `#pragma omp parallel for`, you're instructing the compiler to generate code that executes parts of your program concurrently on multiple threads.

Each thread runs a separate instance of the specified loop, and the threads work together to complete the computation faster than a single thread would. The number of threads can be adjusted based on the characteristics of your system and the problem at hand.

The "for" keyword indicates that the upcoming loop should be parallelized. This is a specific construct in OpenMP to express parallel iterations of a loop.

Here's a breakdown:

`#pragma omp`: This is the directive that indicates the beginning of an OpenMP pragma. Pragmas are special instructions for the compiler. `omp` stands for Open Multi-Processing.

`parallel`: This keyword indicates that the block of code should be executed by multiple threads in parallel.

`for`: This keyword indicates that what follows is a loop that should be parallelized.

What is a thread? A thread is the smallest unit of execution that a program can schedule. It's a sequence of programmed instructions that can be managed independently by a scheduler, which is typically part of the operating system. A program, including your C code, can consist of multiple threads that execute independently but share the same resources, such as memory space and file descriptors.

Here are some key points about threads:

Independence: Threads within the same process share the same data space and resources, but they execute independently. Each thread has its own set of registers and its own stack, but they can read and write the same data.

Concurrency: Multiple threads in a program can execute simultaneously, providing a form of parallelism. This is especially useful for tasks that can be performed independently, like in the case of parallelizing a loop or performing background tasks while the main thread handles user input.

Communication: Threads within the same process can communicate with each other by sharing data. However, care must be taken to synchronize access to shared data to avoid race conditions and other concurrency issues.

Creation and Termination: Threads can be created and terminated during the execution

of a program. They share the process's resources, but each thread has its own program counter and runs independently.

Parallel Programming: In the context of parallel programming, threads are often used to divide a program into tasks that can be executed concurrently, improving performance on multi-core systems.

You can check the number of available threads in a Unix-based machine using command-line utilities. The number of available threads is often synonymous with the number of logical cores on your processor. Here are a few ways to check this:

1. Using `nproc` Command: This command returns the number of processing units available to the current process, which is often the number of logical cores.
2. Using `lscpu` Command: This command provides detailed information about the CPU architecture, including the number of cores.
3. Using `sysctl`: This command shows the number of CPUs (or processing units) on your system.
4. Using `top` or `htop`: These commands display real-time system statistics, including the number of CPUs.
5. Checking `/proc/cpuinfo`: This command reads the CPU information and counts the number of processor lines.
6. Using `lstopo`: If you have the `hwloc` package installed, you can use the `lstopo` command to get a graphical view of your system's architecture, including the number of cores.

Choose the method that suits your preference and the tools available on your Unix-based system. Each method should provide information about the number of logical cores or processing units, which typically corresponds to the number of available threads.

Even though we can feel the execution takes less time but in the terminal almost the same amount of CPU time is taken when I use one thread or four threads. For example, in **my computer** using one thread I see:

```
CPU Time: 33.383639 seconds
Number of Threads: 1
```

In my computer when I use four threads I see:

```
CPU Time: 37.663714 seconds
Number of Threads: 4
```

First question is **1. Why?** Second, **2. Why even the CPU time taken by four threads is higher?**

CPU Time VS Wall Clock Time:

When you're working with parallel computing, the total CPU time reported often includes the time taken by all the threads. To measure the actual time taken by the entire program, including all threads, you can use a wall clock timer. The wall clock time represents the actual time that elapses from the start to the end of the program, including all processes and threads. To measure the actual time taken we can use `omp_get_wtime()`, like:

```
start = omp_get_wtime();

// Call the matrix multiplication function
matrixMultiplication(A, B, C);

end = omp_get_wtime();
cpu_time_used = end - start;

printf("Wall Clock Time: %f seconds\n", cpu_time_used);
printf("Number of Threads: %d\n", omp_get_max_threads());
```

This code measures the wall clock time using `omp_get_wtime()` and prints the result along with the number of threads used. Note that `omp_get_wtime()` provides higher precision than clock() for measuring wall clock time.

Now, in **my computer** using one thread I see:

```
Wall Clock Time: 32.128032 seconds
Number of Threads: 1
```

In my computer when I use four threads I see:

```
Wall Clock Time: 9.263877 seconds
Number of Threads: 4
```

Compare these numbers with the maximum theoretical speed-up that we could get, mentioned at the beginning of this section.

But 2. Why the CPU time taken by multiple threads is higher?

The reason for the gap in CPU time when using multiple threads compared to a single thread can be attributed to parallel overhead and the nature of the problem being solved. Here are some factors to consider:

- **Parallel Overhead**: Introducing parallelism adds overhead due to the coordination (communication) and synchronization needed among threads. This overhead includes managing threads, dividing the work among them, and merging their results. For certain problems or small data sizes, the overhead may outweigh the benefits of parallelism.
- **Synchronization**: In your specific case, the matrix multiplication is a computationally intensive task, but it also involves a lot of data dependencies (each element of C depends on multiple elements of A and B). Synchronizing threads to ensure correct results may introduce overhead.
- **Memory Hierarchy and Cache**: The effectiveness of parallel processing can be influenced by the memory hierarchy and cache behavior. If the working set of your data is not fitting well in the caches, you may experience cache thrashing or inefficient use of cache, especially when multiple threads are accessing the same data.
- **Load Imbalance**: If the workload is not evenly distributed among the threads, some threads may finish their work earlier and be idle while waiting for others to complete. Load imbalance can result in suboptimal performance.

Warning! Parallel programming is a powerful approach to enhance the performance of computational tasks by executing them concurrently using multiple processors or cores. However, it is crucial to emphasize that the effectiveness of parallelization relies heavily on the quality of the initial sequential program.
A well-optimized sequential program serves as the foundation for parallelization. Before venturing into parallel computing, it is essential to ensure that the sequential version of the program is correctly implemented, thoroughly tested, and optimized for performance. This includes attention to algorithmic efficiency, appropriate data structures, and minimizing unnecessary computations.
Here are key considerations:

- Correctness: The sequential program should produce correct results. Debugging and validating the correctness of a parallel program can be significantly more challenging.
- Efficiency: Optimize the sequential code for performance. Identify and address bottlenecks, unnecessary computations, and inefficient algorithms.
- Profiling: Use profiling tools to analyze the runtime behavior of the sequential program. This helps identify performance hotspots and areas for improvement.
- Algorithmic Efficiency: Choose algorithms that are well-suited for parallelization. Some algorithms may have inherent dependencies that limit parallel efficiency.
- Data Structures: Ensure that data structures are appropriately designed for both correctness and efficiency. Access patterns should be conducive to parallelism.

In summary, the decision to parallelize a program should come after establishing a solid foundation in sequential programming. A well-optimized sequential program serves as a benchmark against which the parallel version can be evaluated. Parallelization is not a remedy for poorly performing sequential code but rather a strategy to amplify the efficiency of an already optimized program.

If I compile the same code with `-O3` compiler optimization flag, this is what I see in **my computer**:

```
Wall Clock Time: 1.427424 seconds
Number of Threads: 4
```

If you remember the initial sequential code was taking about 33 seconds in **my computer**. Now you can see we have a much faster program. Try using different number of Threads. Let's say if you have 8 threads in your machine, use all the 8 threads and compare the results with 8-2=6

threads. Why you are not seeing the results you were expecting?

6.2 Loop Unrolling

Loop unrolling is a compiler optimization technique aimed at increasing the execution speed of a program by reducing the overhead of loop control code. In loop unrolling, multiple iterations of the loop are combined into a single iteration, effectively reducing the number of loop-control instructions and improving instruction pipelining. For example if the original loop is:

```
for (int i = 0; i < N; i++) {
 c[i] = a[i] + b[i];
}
```

The vectorized unrolled loop can be:

```
for (int i = 0; i < N; i += 4) {
 c[i] = a[i] + b[i];
 c[i + 1] = a[i + 1] + b[i + 1];
 c[i + 2] = a[i + 2] + b[i + 2];
 c[i + 3] = a[i + 3] + b[i + 3];
}
// Handle remainder of the loop
```

Loop unrolling exposes more instruction-level parallelism, allowing the processor to execute multiple instructions simultaneously. This is especially beneficial on modern superscalar architectures with multiple execution units.

Unrolling reduces the number of loop control instructions, such as loop counters and branching instructions, which can improve the efficiency of the instruction pipeline. Compile and run the following code to see the difference!

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

#define N 1000000000

void addArrays(int *a, int *b, int *c, int n)
{
 for (int i = 0; i < n; i++)
 {
```

```
    c[i] = a[i] + b[i];
  }
}

void addArraysUnrolled(int *a, int *b, int *c, int n)
{
  // Unrolling the loop
  for (int i = 0; i < n; i += 4)
  {
    c[i] = a[i] + b[i];
    c[i + 1] = a[i + 1] + b[i + 1];
    c[i + 2] = a[i + 2] + b[i + 2];
    c[i + 3] = a[i + 3] + b[i + 3];
  }
}

int main()
{
  int *a, *b, *c;
  a = (int *)malloc(N * sizeof(int));
  b = (int *)malloc(N * sizeof(int));
  c = (int *)malloc(N * sizeof(int));

  // Initialize arrays
  for (int i = 0; i < N; i++)
  {
    a[i] = i;
    b[i] = i * 2;
  }

  double start_time, end_time;

  // 1. Without loop unrolling
  start_time = omp_get_wtime();
  addArrays(a, b, c, N);
  end_time = omp_get_wtime();
  printf("Time without loop unrolling: %f seconds\n", end_time -
     start_time);
```

```
    // 2. With loop unrolling
    start_time = omp_get_wtime();
    addArraysUnrolled(a, b, c, N);
    end_time = omp_get_wtime();
    printf("Time with loop unrolling: %f seconds\n", end_time -
       start_time);

    // 3. Parallel with 4 threads without loop unrolling
    start_time = omp_get_wtime();
    #pragma omp parallel for num_threads(4)
    for (int i = 0; i < N; i++)
    {
     c[i] = a[i] + b[i];
    }
    end_time = omp_get_wtime();
    printf("Time parallel (4 threads) without loop unrolling: %f
       seconds\n", end_time - start_time);

    // 4. Parallel with 4 threads with loop unrolling
    start_time = omp_get_wtime();
    #pragma omp parallel for num_threads(4)
    for (int i = 0; i < N; i += 4)
    {
     c[i] = a[i] + b[i];
     c[i + 1] = a[i + 1] + b[i + 1];
     c[i + 2] = a[i + 2] + b[i + 2];
     c[i + 3] = a[i + 3] + b[i + 3];
    }
    end_time = omp_get_wtime();
    printf("Time parallel (4 threads) with loop unrolling: %f
       seconds\n", end_time - start_time);

    free(a);
    free(b);
    free(c);
}
```

Here is the result I see in **my machine**:

```
Time without loop unrolling: 1.916951 seconds
Time with loop unrolling: 1.014297 seconds
Time parallel (4 threads) without loop unrolling: 1.003701 seconds
Time parallel (4 threads) with loop unrolling: 1.003300 seconds
```

It's important to note that the effectiveness of loop unrolling and parallelization depends on the specific characteristics of the target architecture and the workload. Additionally, loop unrolling may increase code size, so a balance between code size and performance should be considered.

6.3 Aliasing and Data Dependency

Aliasing occurs when two or more pointers or references can access the same memory location. In the context of parallel computing, aliasing can lead to unintended interactions between threads.

Here is an example of aliasing:

```
int main() {
 int a[10];
 int *b = a;

 #pragma omp parallel for
 for (int i = 0; i < 10; i++) {
  b[i] = 2;  // Aliasing between 'a' and 'b'
 }
 return 0;
}
```

Why It's a Challenge: Aliasing makes it difficult to reason about and control the flow of data between threads. If one thread modifies data that another thread is also accessing or modifying, it can lead to race conditions and incorrect results.

Data dependency refers to the relationship between the values of different program variables. There are two main types:

1. True Dependence: A write to a memory location is followed by a read from the same location.
2. Anti-dependence: A read from a memory location is followed by a write to the same location.

Here is an example of data dependency:

```
int main() {
 int a[10];
 int b[10];

 #pragma omp parallel for
 for (int i = 1; i < 10; i++) {
  a[i] = a[i - 1] + b[i];  // True dependence
 }
}
```

Why It's a Challenge: Parallel computing relies on the independence of tasks. Data dependence introduces dependencies between different tasks, making it challenging to parallelize them without introducing synchronization overhead.

In both examples, parallelizing or vectorizing the loops could lead to incorrect results due to aliasing or data dependence. Identifying and resolving these issues is crucial for effective parallelization.

Parallel computing involves dividing tasks into independent parts. Aliasing and data dependence can introduce dependencies between tasks, making it challenging to parallelize effectively.

Compilers rely on assumptions about data independence to optimize code. Aliasing and data dependence make these assumptions invalid, limiting the effectiveness of compiler optimizations.

Let's compile the same code by adding `-fopt-info-vec` **and** `-O3` flags. What we see in the terminal is:

```
pedram.c:70:11: optimized: loop vectorized using 16 byte vectors
pedram.c:70:11: optimized:

loop versioned for vectorization because of possible aliasing
pedram.c:60:11: optimized: loop vectorized using 16 byte vectors
pedram.c:60:11: optimized:

loop versioned for vectorization because of possible aliasing
pedram.c:60:11: optimized: loop vectorized using 8 byte vectors
pedram.c:9:20: optimized: loop vectorized using 16 byte vectors
pedram.c:9:20: optimized:

loop versioned for vectorization because of possible aliasing
```

```
pedram.c:9:20: optimized: loop vectorized using 8 byte vectors
pedram.c:18:20: optimized: loop vectorized using 16 byte vectors
pedram.c:18:20: optimized:

loop versioned for vectorization because of possible aliasing
pedram.c:9:20: optimized: loop vectorized using 16 byte vectors
pedram.c:35:20: optimized: loop vectorized using 16 byte vectors
```

Let's break down the messages:

Loop Vectorization:

```
loop vectorized using 16 byte vectors
loop vectorized using 8 byte vectors
```

These messages indicate that the compiler has successfully vectorized certain loops in your code. Vectorization is a process where the compiler transforms a loop to use SIMD (Single Instruction, Multiple Data) instructions, allowing multiple operations to be performed simultaneously.

The sizes mentioned (16 and 8 bytes) typically refer to the size of the vector registers used in the vectorization process.

Loop Versioning for Vectorization Due to Possible Aliasing:

```
loop versioned for vectorization because of possible aliasing
```

This message suggests that the compiler is performing loop versioning to handle possible aliasing. Aliasing occurs when two pointers or references can access the same memory location. Vectorization, in the presence of aliasing, can lead to incorrect results. To address this, the compiler may create multiple versions of the loop to handle different scenarios, ensuring correctness.

Modern compilers can produce multiversioned code containing vectorized and unvectorized versions. The correct version may be chosen at runtime.

Tips! Parallelizing loops is a crucial aspect of harnessing the power of parallel computing. Below are guidelines for optimizing loops for parallelization:

1. **Known Number of Iterations:**
   - The number of loop iterations must be known at runtime.
   - Single control flow within the loop is essential.
   - Avoid `break` statements within the loop.
   - Minimize the use of `if` and `switch` statements.
   - Note: Compilers can **sometimes** work around these constraints and apply vectorization.
2. **Avoid Function Calls:**
   - Eliminate function calls within the loop whenever possible.
   - **Inlining functions** is acceptable, as it can aid in optimization.
   - Certain vectorized functions (e.g., `sin`, `sqrt`) from libraries like the Intel Vector Math Library may be exceptions.
3. **Avoid Indirect Indexing:**
   - Steer clear of indirect indexing, such as `a[b[i]]`.
   - Ensure that data is aligned and sequential in memory for efficient access.
   - In nested loops, vectorization is typically applied to the innermost loop.
   - Compiler optimizations may include reordering loop sequences for better performance.

What is inline function? Inlining functions is a compiler optimization technique where the compiler replaces a function call with the actual body of the function at the call site. This can result in performance improvements by eliminating the overhead associated with a function call. Here's an example:

```
#include <stdio.h>

// Declare the function (no need for inline here)
int add(int a, int b);

int main() {
```

```
    int result = add(3, 4);
    printf("Result: %d\n", result);
    return 0;
}

// Define the function separately
inline int add(int a, int b) {
    return a + b;
}
```

In the above example, the inline keyword is used before the function definition, suggesting to the compiler that it should attempt to inline the function. **However**, the decision to actually `inline` the function is ultimately up to the compiler.

6.4 Speedup, Efficiency, Scalability

This section delves into crucial metrics and laws that govern the speedup, efficiency, and limitations of parallel computing. It explores concepts such as Amdahl's Law, shedding light on the trade-offs between parallelism and sequential processing. Additionally, we'll examine sources of overhead that can impact the performance of parallel algorithms.

By grasping these principles, we gain insights into how to design and optimize parallel programs effectively, understanding their limitations and potentials. These laws are essential for understanding the limits and possibilities of parallel computing and optimizing algorithms for parallel execution.

Given a problem, let $T(1)$ be the time for the fastest sequential algorithm to solve it (on one processor). Speedup ($S(p)$) is defined as:

$$S(p) = \frac{T(1)}{T(p)}$$

Efficiency ($E(p)$) is defined as:

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{pT(p)}$$

Where:

$0 < E(p) \leq 1$. It represents the fraction of time a processing element is utilized. Let's try

the matrix multiplication example with N=5000 and 1,2,4,8, and 16 threads. These results are obtained on **my machine** with 8 cores:

p	1	2	4	8	16
time (sec)	79	49	22	27	58
S(p)	1	1.61	3.59	2.93	1.36
E(p)	1	0.81	0.90	0.37	0.09

Why the time taken 8 and 16 threads are higher than 4 threads?

Amdahl's Law describes the limitation on speedup when a fraction of the code remains sequential. If r is the fraction of statements that can be executed in parallel, then $(1-r)$ is the fraction of statements that is inherently serial. Time for serial part will be $(1-r)T(1)$. The total time with p processors can be defined by:

$$T(p) = r\frac{T(1)}{p} + (1-r)T(1)$$

Then the Speedup ($S(p)$) is given by:

$$S(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{r\frac{T(1)}{p} + (1-r)T(1)} = \frac{1}{\frac{r}{p} + (1-r)}$$

The Speedup is limited by the sequential part of the program and cannot exceed $\frac{1}{(1-r)}$.

For more information go to High Performance Parallel Computing lectures 7, 8, 9, and 10.

6.5 OpenMP Basics

OpenMP (Open Multi-Processing) is a powerful API that facilitates parallel programming in C, C++, and Fortran. It simplifies the creation and management of multithreaded applications, enhancing performance by exploiting multiple cores in shared-memory systems.

Here's an example using `#pragma omp parallel` to print "Hello" from different threads:

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

int main() {
 #pragma omp parallel
 {
```

```
    int tid = omp_get_thread_num();
    printf("Hello from thread %d\n", tid);
   }

   return 0;
  }
```

`#pragma omp parallel` creates a team of threads, and the code within the parallel region is executed by each thread in the team.

In this example, each thread prints "Hello" along with its thread number obtained using `omp_get_thread_num()` function. All threads execute the same block of code within the parallel region concurrently. Here, is the output:

```
Hello from thread 3
Hello from thread 6
Hello from thread 4
Hello from thread 1
Hello from thread 2
Hello from thread 0
Hello from thread 7
Hello from thread 5
```

Here's an example using `#pragma omp parallel num_threads(n)` to specify the number of threads:

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

int main() {
 int n = 4; // Number of threads

 #pragma omp parallel num_threads(n)
 {
  int tid = omp_get_thread_num();
  printf("Hello from thread %d\n", tid);
 }

 return 0;
}
```

`#pragma omp parallel num_threads(n)` creates a parallel region with a specified number of threads (`n` in this case).

The code within the parallel region is executed by `n` threads concurrently. Each thread prints "Hello" along with its thread number obtained using `omp_get_thread_num()` function. All threads execute the same block of code within the parallel region concurrently.

```
Hello from thread 3
Hello from thread 0
Hello from thread 1
Hello from thread 2
```

The `#pragma omp` sections directive is used to divide code into sections that can be executed by different threads in a parallel region. Each section is executed by a single thread. Here's an example:

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

int main()
{
 #pragma omp parallel
 {
  #pragma omp sections
  {
   #pragma omp section
   {
    printf("Thread %d in section 1\n", omp_get_thread_num());
   }

   #pragma omp section
   {
    printf("Thread %d in section 2\n", omp_get_thread_num());
   }
  }
 }

 return 0;
}
```

`#pragma omp sections` is used to define sections of code that can be executed by different

threads.

Inside the parallel region, each `#pragma omp section` block represents a section of code that can be executed by a single thread.

In this example, two sections are defined and executed by different threads within the parallel region. Each section's content is enclosed within the `#pragma omp section` directive.

```
Thread 1 in section 1
Thread 5 in section 2
```

The combination of `#pragma omp task` and `#pragma omp single` can be used to create tasks that can be executed by multiple threads but ensures that only one thread generates these tasks. Here's an example showcasing their use:

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

void task_function(int id) {
 printf("Task %d executed by thread %d\n", id, omp_get_thread_num
    ());
}

int main() {
 #pragma omp parallel
 {
  #pragma omp single
  {
   printf("Single block on thread %d\n", omp_get_thread_num());

   // Create tasks within the single region
   #pragma omp task
   task_function(1);

   #pragma omp task
   task_function(2);

   #pragma omp task
   task_function(3);
  }
 }
```

```
    return 0;
}
```

`#pragma omp single` ensures that the enclosed block is executed by only one thread. Inside the single region, three tasks (`#pragma omp task`) are generated. These tasks will be executed by the available threads in the parallel region. Each task calls the `task_function` with a unique ID.

This setup demonstrates the creation of tasks within a single region to be executed by multiple threads in parallel. The single construct ensures that only one thread generates the tasks, avoiding multiple threads generating the same tasks simultaneously.

```
Single block on thread 5
Task 1 executed by thread 0
Task 2 executed by thread 6
Task 3 executed by thread 5
```

Let's consider the two scenarios with `#pragma omp parallel for` and `#pragma omp parallel` with nested `#pragma omp for`:

Scenario 1: Using `#pragma omp parallel for` twice:

This scenario has two separate parallel regions, each with a `#pragma omp parallel for` directive.

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

int main() {
    #pragma omp parallel for
    for (int i = 0; i < 10; i++) {
        printf("Loop 1: Thread %d, i = %d\n", omp_get_thread_num(), i);
    }

    #pragma omp parallel for
    for (int j = 0; j < 10; j++) {
        printf("Loop 2: Thread %d, j = %d\n", omp_get_thread_num(), j);
    }

    return 0;
}
```

Two separate parallel regions are created, each starting a new team of threads.

The `#pragma omp parallel for` directive creates a parallel loop for each loop separately, where each iteration can be executed by different threads.

```
Loop 1: Thread 2, i = 4
Loop 1: Thread 4, i = 6
Loop 1: Thread 3, i = 5
Loop 1: Thread 6, i = 8
Loop 1: Thread 5, i = 7
Loop 1: Thread 1, i = 2
Loop 1: Thread 7, i = 9
Loop 1: Thread 0, i = 0
Loop 1: Thread 0, i = 1
Loop 1: Thread 1, i = 3
Loop 2: Thread 2, j = 4
Loop 2: Thread 1, j = 2
Loop 2: Thread 1, j = 3
Loop 2: Thread 7, j = 9
Loop 2: Thread 5, j = 7
Loop 2: Thread 6, j = 8
Loop 2: Thread 4, j = 6
Loop 2: Thread 3, j = 5
Loop 2: Thread 0, j = 0
Loop 2: Thread 0, j = 1
```

Scenario 2: Using `#pragma omp parallel` with nested `#pragma omp for`:

In this scenario, both loops are within a single parallel region.

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

int main() {
 #pragma omp parallel
 {
  #pragma omp for
  for (int i = 0; i < 10; i++) {
   printf("Loop 1: Thread %d, i = %d\n", omp_get_thread_num(), i)
      ;
  }
```

```
    #pragma omp for
    for (int j = 0; j < 10; j++) {
     printf("Loop 2: Thread %d, j = %d\n", omp_get_thread_num(), j)
        ;
    }
   }

   return 0;
}
```

In this case, a single parallel region is created, and the two loops are within it. Both loops are parallelized within the same team of threads, potentially reducing the overhead of creating multiple parallel regions.

Using separate `#pragma omp parallel for` directives allows independent parallel execution of loops. Conversely, placing multiple loops within a single `#pragma omp parallel` directive allows them to share the same team of threads, which can be more efficient in some cases, especially if synchronization or communication between loops is needed. The reason for these variations lies in the organization of parallelism.

In this case you will get the same results. Although we know that the order of threads every time we run the code might be different.

The `#pragma omp barrier` directive ensures that all threads in the parallel region synchronize at this point. It halts the execution of each thread until all threads have reached the barrier, at which point they can continue.

It's used when you need all threads to reach a certain point in code before proceeding. For example, when one part of the code is dependent on results computed by other threads. For instance:

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

int main() {
 #pragma omp parallel num_threads(4)
 {
  // Some work done by each thread
  printf("Hello from thread %d\n", omp_get_thread_num());

  // Barrier synchronization point
```

```
    #pragma omp barrier

    // Code that depends on all threads reaching the barrier
    printf("World from thread %d\n", omp_get_thread_num());
   }

   return 0;
  }
```

```
Hello from thread 0
Hello from thread 2
Hello from thread 1
Hello from thread 3
World from thread 1
World from thread 2
World from thread 3
World from thread 0
```

`#pragma omp taskwait` is used to ensure that a task construct's children have completed execution before the current task proceeds further.

It's useful when you have tasks with dependencies, ensuring that a task doesn't continue before its dependent tasks have finished executing. For example:

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

void do_work() {
 // Some work performed by the task
 printf("Task work\n");
}

int main() {
 #pragma omp parallel
 {
  #pragma omp single
  {
   // Task constructs
   #pragma omp task
   {
```

```
      do_work();
    }
    #pragma omp taskwait  // Ensure the task completes before
       proceeding
  }
 }

 return 0;
}
```

```
Task work
```

These directives are crucial for managing synchronization among threads and tasks, ensuring proper execution order and handling dependencies in parallelized code.

6.5.1 Scheduling

Scheduling in OpenMP refers to the way iterations of a loop are divided and allocated to different threads for parallel execution. Here's a description of different scheduling types:

1. Static Scheduling:

In static scheduling, iterations of the loop are divided into chunks at compile-time and statically distributed among the available threads. Use this when the workload of each iteration is relatively uniform.

Directive: `schedule(static, chunk_size)`

2. Dynamic Scheduling:

Dynamic scheduling allocates iterations to threads dynamically at runtime. Threads pick up new iterations as they complete their previous ones. We use this when the workload is irregular, and iterations take varying time.

Directive: `schedule(dynamic, chunk_size)`

3. Guided Scheduling:

Guided scheduling initially assigns larger chunks of iterations to threads and gradually reduces the chunk size. It starts with larger chunks and ends up with smaller ones. It is useful when the workload varies significantly across iterations and more control over load balancing is required.

Directive: `schedule(guided, chunk_size)`

4. Auto Scheduling:

The auto scheduling type lets the compiler choose the scheduling type automatically based on certain criteria, often depending on the loop and system characteristics. It is suitable when you're unsure which scheduling type to use or when experimenting with different schedules.

Directive: `schedule(auto)`

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

#define NUM_THREADS 4
#define ARRAY_SIZE 20

int main() {
 int arr[ARRAY_SIZE];
 for (int i = 0; i < ARRAY_SIZE; i++) {
  arr[i] = i;
 }

 printf("Static Schedule:\n");
 #pragma omp parallel num_threads(NUM_THREADS)
 {
  #pragma omp for schedule(static)
  for (int i = 0; i < ARRAY_SIZE; i++) {
   printf("Thread %d: arr[%d] = %d\n", omp_get_thread_num(), i,
      arr[i]);
  }
 }

 printf("\nDynamic Schedule:\n");
 #pragma omp parallel num_threads(NUM_THREADS)
 {
  #pragma omp for schedule(dynamic)
  for (int i = 0; i < ARRAY_SIZE; i++) {
   printf("Thread %d: arr[%d] = %d\n", omp_get_thread_num(), i,
      arr[i]);
  }
 }

 printf("\nGuided Schedule:\n");
```

```
 #pragma omp parallel num_threads(NUM_THREADS)
 {
  #pragma omp for schedule(guided)
  for (int i = 0; i < ARRAY_SIZE; i++) {
   printf("Thread %d: arr[%d] = %d\n", omp_get_thread_num(), i,
      arr[i]);
  }
 }

 printf("\nAuto Schedule:\n");
 #pragma omp parallel num_threads(NUM_THREADS)
 {
  #pragma omp for schedule(auto)
  for (int i = 0; i < ARRAY_SIZE; i++) {
   printf("Thread %d: arr[%d] = %d\n", omp_get_thread_num(), i,
      arr[i]);
  }
 }

 return 0;
}
```

(not finished yet!)

`#pragma omp for schedule(dynamic)`

L12

reading mtx what are the indices a solver example! maybe a dense or sparse!

7 A Short Overview of ANN - Studying This Section is Optional

Artificial Intelligence (AI) is a broad field of computer science that focuses on creating systems or machines capable of performing tasks that typically require human intelligence. These tasks include understanding natural language, recognizing patterns, making decisions, solving problems, and learning from experience. AI systems aim to simulate various aspects of human intelligence, such as perception, reasoning, problem-solving, and decision-making. AI is utilized in a wide range of applications, including natural language processing, computer vision, robotics, recommendation

systems, and more.

One popular subfield of AI is machine learning, which involves the development of algorithms and models that enable systems to learn from data and make predictions or decisions based on that data. Artificial Neural Networks (ANN) are a specific type of machine learning model inspired by the structure and functioning of biological neural networks in the human brain. ANNs consist of interconnected nodes (neurons) organized into layers, and they are used for various tasks, including classification, regression, and pattern recognition.

Here's a brief overview of ANNs

- ANNs are composed of layers of interconnected artificial neurons.
- Input data is passed through the network, and each neuron processes the data and passes it to the next layer.
- ANNs can have multiple hidden layers, making them capable of handling complex data representations.
- They are commonly used for tasks like image and speech recognition, natural language processing, and more.

Let's start with simple examples of fitting. You will see the similarity between fitting and ANNs.

7.1 Fitting ax+b with Two Points

Fitting a linear model to data in the form of $y = ax + b$ involves finding the values of a and b that best describe the relationship between the dependent variable y and the independent variable x. Given two points (x_1, y_1) and (x_2, y_2) we can calculate the values of a and b using the following steps:

1. Calculate the slope (a) using the formula:

$$a = \frac{y_2 - y_1}{x_2 - x_1}$$

The slope represents the rate of change of the dependent variable y with respect to the independent variable x. It gives the direction and steepness of the linear relationship between x and y.

2. Calculate the intercept (b) using the formula:

$$b = y_1 - a \times x_1$$

The intercept (b) is the value of yy when $x = 0$, or more generally, it is the value of yy when $x = x1$. It determines the vertical position of the line on the y-axis.

After calculating both a and b, you can express the linear equation $y = ax + b$ that best fits the given two points (x_1, y_1) and (x_2, y_2). This line will be the best linear approximation of the relationship between the variables x and y based on the provided data points (Figure 1).

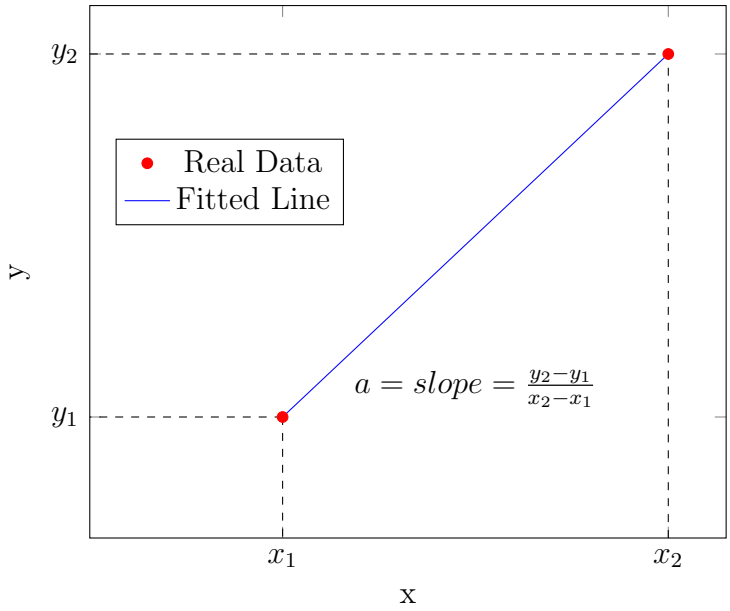


Figure 1: Fitting $y = a \times x + b$ with (x_1, y_1) and (x_2, y_2)

7.2 Fitting ax+b with Multiple Points

But what if I have multiple data?

Let's say in my chemistry lab, I have done some experiments and when I plot the points x vs y, I get the following plot. Now I want to find a mathematical equation presenting all these points.

When you have multiple data points (x_1, y_1), (x_2, y_2) ,..., (x_n, y_n), you can still fit a linear model in the form of $y = ax + b$ to the data. The goal is to find the best values of a and b that minimize the overall error between the predicted values $\hat{y}_i$ (based on the linear model) and the actual observed values y_i.

There are various techniques to perform linear regression and find the optimal values of a and b.

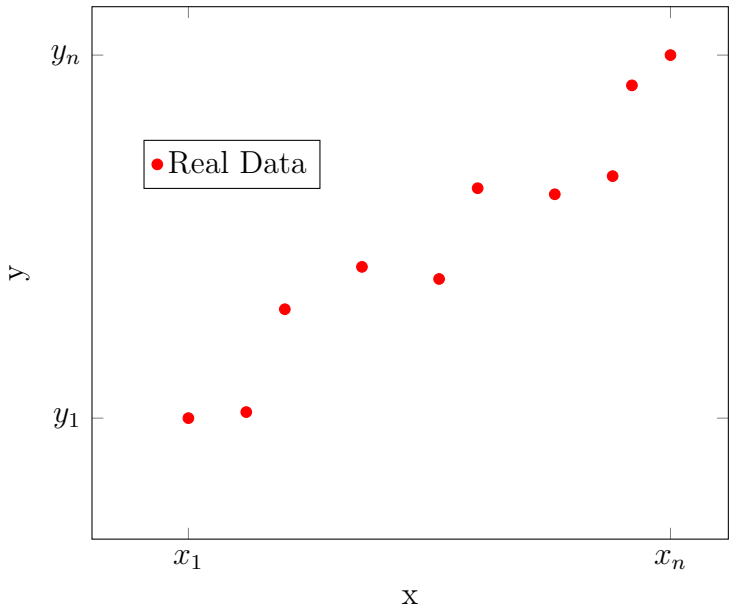


Figure 2: Some real data points.

One common approach is the method of least squares, which aims to minimize the sum of squared differences between the predicted values and the actual values. Here's an overview of the steps:

1. Set up the linear equation:
$$y = a \times x + b$$

2. Define the error for each data point i as the difference between the observed value yi and the predicted value $\hat{y}_i$ based on the linear equation:
$$e_i = y_i - \hat{y}_i = y_i - (ax + b)$$

3. Define the objective function to minimize the overall error:
$$E(a, b) = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - (ax_i + b))^2$$

4. Find the values of a and b that minimize the objective function $E(a, b)$. This can be done through calculus and solving the partial derivatives of E with respect to a and b set to zero.

5. Once you find the optimal values of a and b, you have the linear model $y = ax + b$ that best fits the given data points.

Every time for each iteration, a set of values for a and b are guessed, and the objective function will be calculated (Figure 3 dashed lines). At the end, the set of values which has the minimum

objective function will be picked (Figure 3 the blue line). If we randomly every time guess the values for a and b it might take a lot of time.

That's why researchers came with many **optimization algorithms** to learn from previous guess how to get close to the optimized solution. If I want to start talking about optimization, it will take a full semester to learn about its basic concepts!

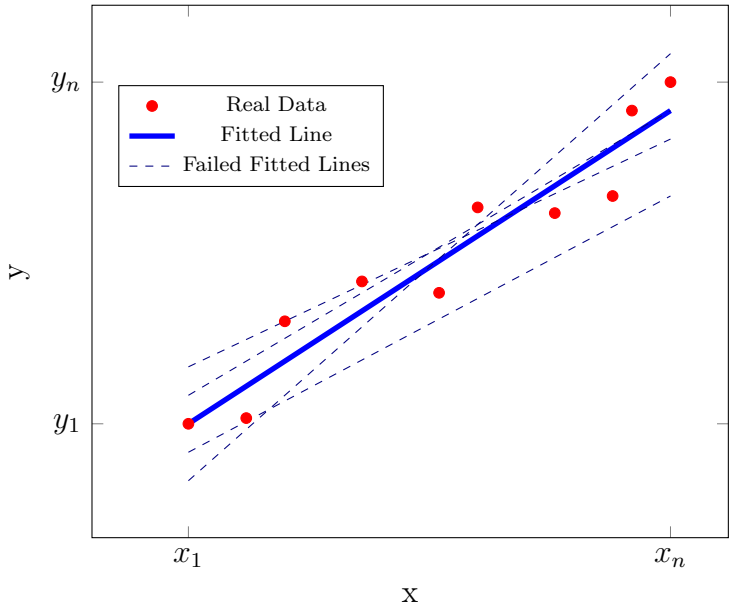


Figure 3: Fitting Process.

7.3 Fitting $a_1 \cdot x_1 + a_2 \cdot x_2 + ... + a_m \cdot x_m + b$ with Multiple Points

When you have multiple input features $(x_1, x_2, \ldots, x_m)$ and you want to fit a linear model to predict the output variable y, the general form of the model is:

$$y = a_1 \cdot x_1 + a_2 \cdot x_2 + ... + a_m \cdot x_m + b$$

where $a_1, a_2, \ldots, a_m$ are the coefficients (weights) corresponding to each input feature, and b is the bias term.

To fit this linear model to the given data points, you can use the method of least squares, which aims to minimize the sum of squared differences between the predicted values $\hat{y}_i$ (based on the linear model) and the actual observed values y_i. Here's an overview of the steps:

1. Set up the linear equation:

$$y = a_1 \cdot x_1 + a_2 \cdot x_2 + ... + a_m \cdot x_m + b$$

2. Define the error for each data point i as the difference between the observed value yi and the predicted value $\hat{y}_i$ based on the linear equation:

$$e_i = y_i - \hat{y}_i = y_i - (a_1 \cdot x_{i,1} + a_2 \cdot x_{i,2} + ... + a_m \cdot x_{i,m} + b)$$

where i starts from 1 to the maximum number of available data points.

3. Define the objective function to minimize the overall error:

$$E(a_1, a_2, ..., a_m, b) = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - (a_1 \cdot x_{i,1} + a_2 \cdot x_{i,2} + ... + a_m \cdot x_{i,m} + b))^2$$

4. Find the values of $a_1, a_2, \ldots, a_m$ and b that minimize the objective function $E(a_1, a_2, ..., a_m, b)$. This can be done through calculus and solving the partial derivatives of E with respect to $a_1, a_2, \ldots, a_m$ and b set to zero.

5. Once you find the optimal values of $a_1, a_2, \ldots, a_m$ and b, you have the linear model $y = a_1 \cdot x_1 + a_2 \cdot x_2 + ... + a_m \cdot x_m + b$ that best fits the given data points.

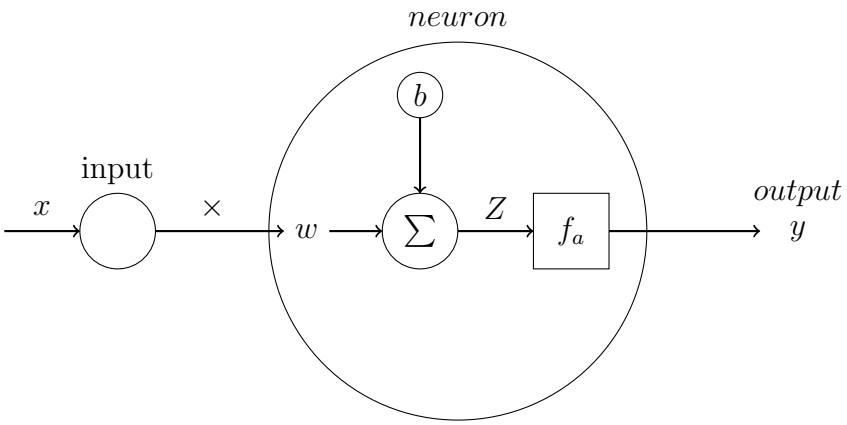
As in the previous case, there are libraries and software tools available that can handle the linear regression process for you, making it easier to find the optimal values of $a_1, a_2, \ldots, a_m$ and b without having to perform the mathematical calculations manually. These libraries use efficient algorithms to perform the regression and find the best-fitted linear model.

7.4 The Similarity Between Fitting and ANN

7.4.1 Single Input - Single Neuron - Single Output Model

ANN is kind of the same, but a lot more complex!

The concept of Artificial Neural Network is taken from the real neural network of the brain, where billions of neurons are connected to make a decision, to remember something, to process what we touch, what we feel, what eyes see and ears hear! Let's consider the simplest ANN model with only one neuron:



This neural network has a simple architecture, consisting of one input, and one neuron. This type of neural network is a basic feedforward neural network. The neuron processes the input data using its weight (w), base (b) and activation function (f_a) to produce an output. Here is the formulation:

$$Z = w \cdot x + b$$

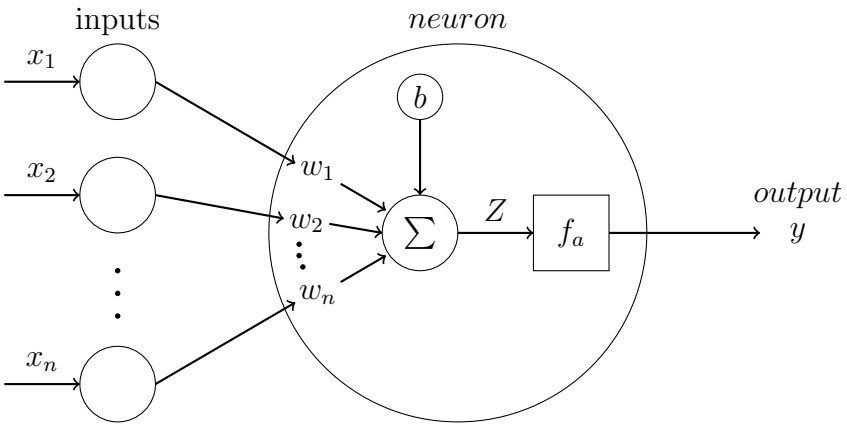
$$y = f_a(Z)$$

As you may noticed here instead of coefficient a we have w. The second difference here is the activation function. The activation function can be any function but also it can vary depending on the requirements and specific use case. Common activation functions include the sigmoid function, ReLU (Rectified Linear Unit), Tanh, etc. The activation function introduces non-linearity to the model, allowing the neural network to learn complex relationships between the input and output.

If we say $f_a(Z) = Z$ then we are not using any activation function ($y = Z$). In this case there is no difference between this ANN model and the previous sections about fitting!

7.4.2 Multiple Inputs - Single Neuron - Single Output Model

What if we have multiple inputs to the neuron, the same as Fitting with Multiple x? In this case we will have a vector of inputs $(x_1, x_2, ..., x_n)$ where n shows the number of inputs. For each input we will have one weight $(w_1, w_2, ..., w_n)$. Following figure can show this concept:



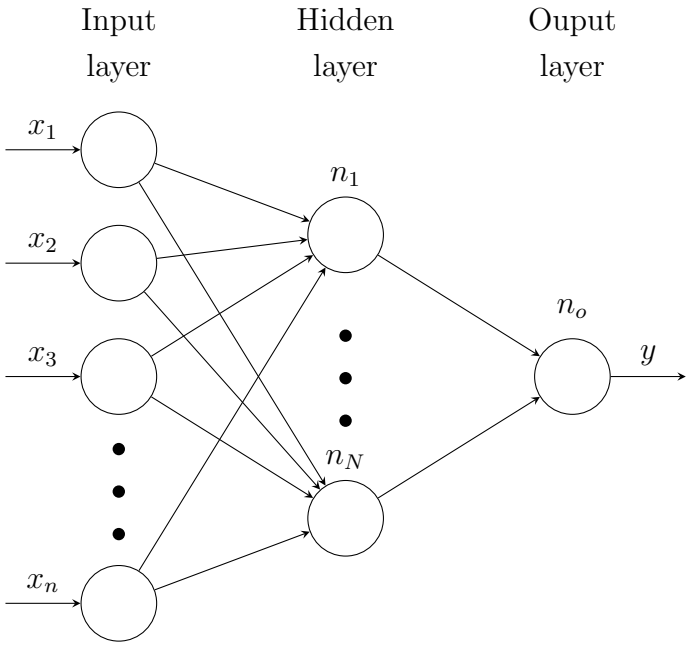
The neuron processes the input data using its weights (w_i), base (b) and activation function (f_a) to produce an output. Here is the formulation, where the equation for Z is exactly the same as Fitting with multiple x:

$$Z = \sum_{i=1}^{n} w_i \cdot x_i + b = w_1 \cdot x_1 + w_2 \cdot x_2 + ... + w_n \cdot x_n + b$$

$$y = f_a(Z)$$

7.4.3 Multiple Inputs - Multiple Neurons - Single Output Model

A more complex model is with multiple neuron in a hidden layer. In this case since we still trying to find a single output y, we will need a extra output layer, to combine the outputs from all neurons something like the following figure:



In this case, **each** neuron processes the input data using its weight**s** ($w_{i,j}$), base (b) and activation function (f_a) to produce an output. The counter i shows how many neurons are in this layer which in this case it starts from 1 to N.

The total number of weights for each neuron in the hidden layers equal to the total number of outputs (or the total number of neurons) from previous layer. So, $w_{i,j}$ shows the weights for the $i-th$ neuron in the hidden layer, and the counter j shows how many weights it will have.

The previous layer, is the input layer (x_1, x_2, ..., x_n) and it has n number of output (in case all neurons are fully connected to the previous layer). So the numbering for weights j **must** start from 1 to n. Then, for each neuron we will have:

$$Z_i = b_i + \sum_{j=1}^{j=n} w_{i,j} \times x_j$$

where Z_i is the linear output of each neuron before it goes through an activation function. If we consider all neurons have the same activation function f_a, the final output of $i-th$ neuron can be represented by:

$$a_i = f_a(Z_i)$$

Where each neuron i will have a final output a_i. a_i will be the inputs for the next layer which is the **output layer** with a single neuron named n_o. The number of weights in this neuron must be equal to the number of outputs in the previous layer which is N. So for the **output layer** we

will have:

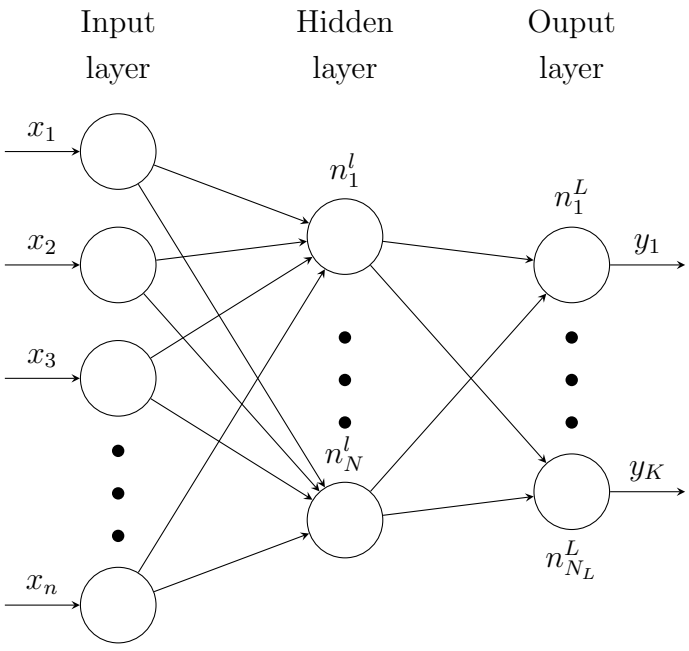
$$Z_o = b_o + \sum_{j=1}^{j=N} w_{o,j} \times a_j$$

Where Z_o is the linear output of single neuron n_o. If we say this layer is also using the same type of activation function f_a, then:

$$y = f_a(Z_o)$$

7.4.4 Multiple Inputs - Multiple Neurons - Multiple Outputs Model

Instead of having on output (y), in some models we might have multiple outputs (y_1, y_2, ..., y_K). The only difference is the **output layer** must have one neuron for each output:



First, since we have two layers (Hidden and Output) layers with different number of neurons, the name of neurons are determined by the superscript l in n_i^l or L in n_i^L, for the hidden layer and output layer, respectively. The hidden layer l has N, and output layer has N_L number of neurons.

Each neuron processes the input data using its weight**s** ($w_{i,j}$), base (b) and activation function (f_a) to produce an output. The counter i shows how many neurons are in this layer which in this case it starts from 1 to N.

The total number of weights for each neuron in the hidden layers equal to the total number of outputs (or the total number of neurons) from previous layer. So, $w_{i,j}$ shows the weights for the $i-th$ neuron in the hidden layer, and the counter j shows how many weights it will have.

The layer before hidden layer is the input layer (x_1, x_2, ..., x_n) and it has n number of outputs (in case all neurons are fully connected to the previous layer). So the numbering for weights j **must** start from 1 to n. Then, for each neuron we will have:

$$Z_i = b_i + \sum_{j=1}^{j=n} w_{i,j} \times x_j$$

where Z_i is the linear output of each neuron before it goes through an activation function. If we consider all neurons have the same activation function f_a, the final output of $i-th$ neuron can be represented by:

$$a_i = f_a(Z_i)$$

Where each neuron i will have a final output a_i. **So far everything is the same as previous section.**

a_i will be the inputs for the next layer which is the **output layer** with **multiple neurons**. The number of weights in each neuron must be equal to the number of outputs in the previous layer which is N. So, for the **output layer** we will have:

$$Z_i = b_i + \sum_{j=1}^{j=N} w_{i,j} \times a_j$$

Where Z_i is the linear output of neuron i. If we say this layer is also using the same type of activation function f_a, then:

$$y_i = f_a(Z_i)$$

7.4.5 Activation Function

Let's talk about f_a known as activation function. The choice of activation function depends on the nature of the output variable y and the specific requirements of the problem. Here are the typical activation functions for the three possible scenarios:

1. Binary Classification (Binary Output):

Let's say we want to categorize a photo say if it is a picture of a "dog". In this case, the ANN model is called **Binary Classification**. There is only two categories and ANN must determine which one it is! The output is "Yes" or "No". It is "True" or "False", or it is "1" or "0". These types of problem's can be modelled by only one output y. For example, we use the activation function like **Sigmoid** function:

$$y = f_a(Z) = \frac{1}{1 + e^{-Z}}$$

The Sigmoid function squashes the output to a value between 0 and 1, which can be interpreted as the probability of the input belonging to the positive class (class 1). For instance:

- If $Z \in (-\inf, 0)$, then $y \in [0,\ 0.5)$ and we can assign this range of y to class 1.
- If $Z \in (0 + \inf)$, then $y \in (0.5,\ 1]$, which will be class 2.

2. Multi-Class Classification (Probabilities for Each Class):

But sometimes we might have multiple outputs (y_1, y_2, ..., y_K). For example, if ANN model is responsible to determine the category between "dog", "cat", and "horse". In this case, we need to have **three** outputs (y_1, y_2, y_3), and each output must give us the probability of each category. For example, for a specific set of inputs (x_i), the outputs of the model are $y_1 = 0.25$, $y_2 = 0.65$, and $y_3 = 0.10$, then we can say the chances that this photo is showing a "cat" or y_2 is 65%.

These types of problems are called **Multi-Class Classification** (Probabilities for Each Class). In multi-class classification, each output y represents the probability of the input belonging to a particular class. The most common activation function for the output layer in this case is the **Softmax** function. The Softmax function ensures that the probabilities for all classes sum up to 1. It provides a probability distribution over the different classes. The formula for the Softmax activation function is:

$$y_i = f_a(Z_i) = \frac{e^{Z_i}}{\sum_{j=1}^{N_L} e^{Z_j}}$$

Where Z_i is the output for class i, N_L the number of classes or outputs (or the number of neurons in the output layer), and $y_i = f_a(Z_i)$ represents the probability of the inputs belonging to class i which it will be a number between 0 to 1. Since we are talking about probability, $\sum_{i=1}^{N_L} y_i = 1$, meaning the summation of all chances must be equal to 100%.

3. Regression (Continuous Real-Valued Output):

In regression problems, where the output y is a continuous real value, the most commonly used activation function for the output layer is the **Linear activation function**. The Linear activation function **simply outputs the weighted sum of the input features without any non-linearity**. It allows the neural network to predict continuous values over a wide range. The formula for the Linear activation function if there is single output is:

$$y = f_a(Z) = Z$$

Which means Z itself is the output and no activation function is applied. If there are multiple outputs then:

$$y_i = f_a(Z_i) = Z_i$$

Again the final output (y_i) from each neuron (i) in the **output layer** is Z_i

These activation functions are widely used in their respective scenarios and can be combined with various hidden layer activation functions such as ReLU, Tanh, etc., to build effective neural network architectures for different types of problems.

7.5 The General Formulation of ANN - Multiple Hidden Layers

The general mathematical formulation of an Artificial Neural Network (ANN) with different numbers of hidden layers and neurons in each layer can be represented as follows.

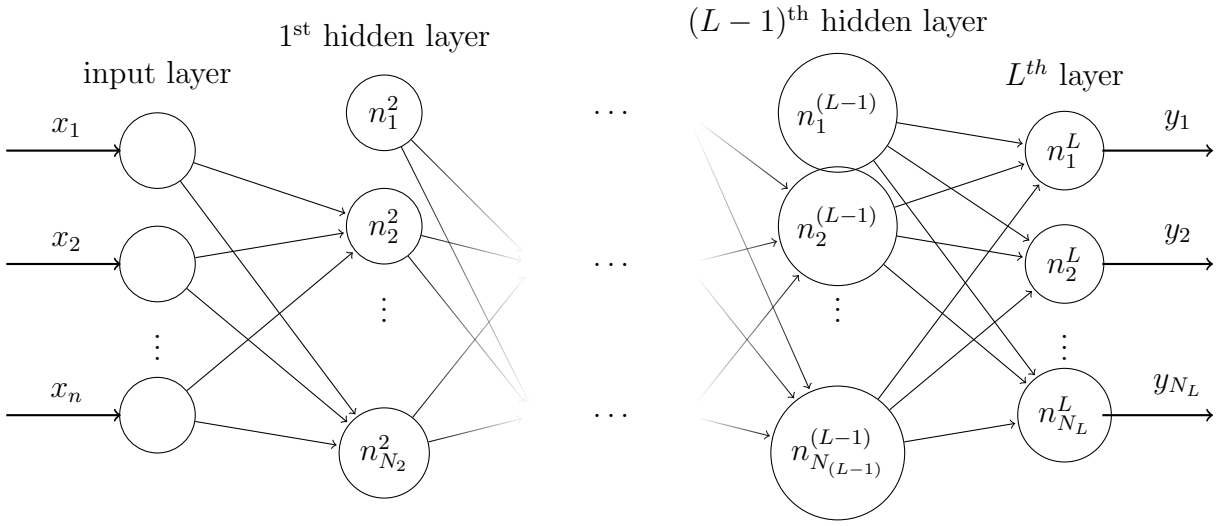


Figure 4: Network Network of a (L)-layer perceptron with n input units and N_L outputs. Layer l^{th} has N_l number of neurons. Every neuron in l^{th} layer (n_i^l) will have a bias b_i^l.

Let:

- X_j be the input of the neural network with $j = 1 : n$ where n is the total number of inputs to the model.
- L be the total number of layers (including input and output layers),
- N_l is the total number of neuron is in the layer l
- $w^l_{i,j}$ be the weights in layer l, and $i = 1 : N_l$ is defining the weights for neuron $i-th$ in this layer, and $j = 1 : N_{l-1}$ is the number of weight that each neuron **must** have. **The total number of weights for each neuron in layer l is equal to the total number of outputs (or the total number of neurons) from previous layer $l-1$.**
- b^l_i be the bias for neuron $i-th$ in layer l,
- Z^l_i the output of neuron $i-th$ in layer l
- in $f^l_{a,i}(Z^l_i)$, the $f^l_{a,i}$ is the activation function applied to the linear output of neuron $i-th$ in layer l,
- and in $a^l_i = f^l_{a,i}$, the a^l_i be the final output from neuron $i-th$ in the layer l.

The connection between layers can be represented as follows:

$l = 1$ represent the input values to the model. Since in this layer NN doesn't have any weights and activation functions it can be say that the final output of the first layer (**input layer**) $a^{l=1}_j$ is:

$$a^1_j = X_j$$

From the second layer $l = 2$ (**first hidden layer**) to the **last (output) layer** L:

$$Z^l_i = b^l_i + \sum_{j=1}^{N_{l-1}} w^l_{i,j} \times a^{l-1}_j$$

Example! For example in the layer $l = 2$:

$$Z_i^2 = b_i^2 + \sum_{j=1}^{N_1} w_{i,j}^2 \times a_j^1$$

and if we want to compute the output of the first neuron ($i = 1$) in this layer

$$Z_1^2 = b_1^2 + \sum_{j=1}^{N_1} w_{1,j}^2 \times a_{i=j}^1$$

Let's say the previous layer which was the input layer, has 4 ($n = 4$) features ($j = 1 : 4$), then we will have 4 outputs from this layer ($N_1 = 4$) including a_1^1, a_2^1, a_3^1 and a_4^1. Essentially, we can use the general formulation for the first neuron in the first hidden layer (Z_1^2):

$$Z_1^2 = b_1^2 + \sum_{j=1}^{4} w_{1,j}^2 \times a_j^1$$

which the expanded equation is:

$$Z_1^2 = b_1^2 + w_{1,1}^2 \dot{a}_1^1 + w_{1,2}^2 \dot{a}_2^1 + w_{1,3}^2 \dot{a}_3^1 + w_{1,4}^2 \dot{a}_4^1$$

Tips! Keep it in mind that in this layer every neuron i will have 4 weights, including $w_{i,1}^2$, $w_{i,2}^2$, $w_{i,3}^2$ and $w_{i,4}^2$, because the previous layer had 4 outputs ($N_1 = 1$), including a_1^1, a_2^1, a_3^1, and a_4^1. It means for each output we need one weight calibrating the significance of input and determining how much it would contribute to the final output.

Now the output of each neuron must go through an activation function:

$$a_i^l = f_{a,i}^l(Z_i^l)$$

Where a_i^l is final output of the layer l and the input values of next layer $l + 1$.

Example for $l = 2$! Now the output of every neuron in the second layer Z_i^2 goes into a the activation function of this layer $f_{a,i}^2$. For example, for the output of first neuron in this layer (Z_1^2)

$$a_1^2 = f_{a,1}^l(Z_1^2)$$

Where a_1^2 will be the final output of the first neuron in the layer 2, which can be one of the inputs for the next layer.

- If the ANN model has one output (y) (one neuron in output layer) then:

$$Z_1^L = b_1^L + \sum_{j=1}^{N_{L-1}} w_{1,j}^L \times a_j^{L-1}$$

$$y = a_1^L = f_{a,1}^L(Z_1^L)$$

- If the ANN model has multiple outputs ($y_1, y_2, ..., y_{N_L}$) (multiple neurons in output layer) then for each neuron we will have one output a_i^L:

$$Z_i^L = b_i^L + \sum_{j=1}^{N_{L-1}} w_{i,j}^L \times a_j^{L-1}$$

$$y_i = a_i^L = f_{a,i}^L(Z_i^L)$$

7.5.1 A Simple ANN Example with 4 Inputs, 2 Hidden Layers, 1 Output

The ANN model has following characteristics:

- If the number of inputs are 4 ($n = 4$).
- There is only 2 hidden layer which are $l = 2$ and $l = 3$, so ($L = 4$).
- The number of neurons in the layer $l = 2$ is 2 ($N_2 = 2$) with n_1^2 and n_2^2 names. Since the previous layer $l = 2 - 1$ had 4 outputs, every neuron in this layer must have 4 weights ($w_{i,j}^2$), where $j = 1 : 4$ is the numbering for weights and $i = 1 : 2$ shows the number of neurons in the layer $l = 2$. So, this layer will have 2 outputs (a_1^2 and a_2^2).
- The number of neurons in the layer $l = 3$ is 4 ($N_3 = 4$) with n_1^3, n_2^3, n_3^3 and n_4^3 names. Since the previous layer $l = 3 - 1$ had 2 outputs, every neuron in this layer must have 2 weights

$(w^3_{i,j})$, where $j = 1:2$ is the numbering for weights and $i = 1:4$ shows the number of neurons in the layer $l = 3$. So, this layer will have 4 outputs $(a^3_1, a^3_2, a^3_3, a^3_4)$.

- Only one output (y), then output layer will have one neuron (n^L_1). Since the previous layer $l = 4-1$ had 4 outputs, the only neuron in this layer must have 4 weights $(w^4_{i=1,j})$, where $j = 1:4$ is the numbering for weights and $i = 1$ shows the only neurons in the output layer $l = L = 4$. So, this layer will have 1 output $(y = a^4_1)$

This is how the ANN model looks like:

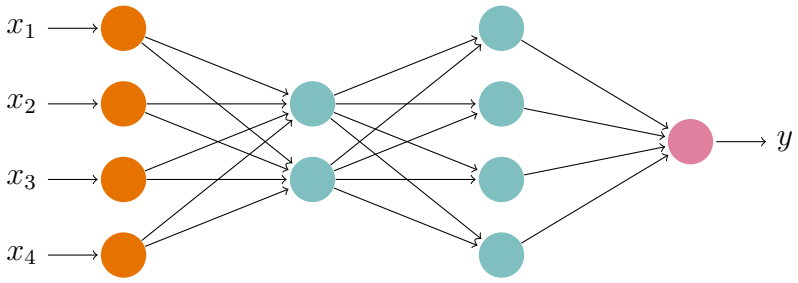


Figure 5: The orange color circles show the inputs layer, green ones shows two hidden layers, and the purple circle is the output layer with one neuron.

For the l=2, the linear output from each neuron n^2_1 and n^2_2 are:

$$Z^2_1 = w^2_{1,1} \cdot X_1 + w^2_{1,2} \cdot X_2 + w^2_{1,3} \cdot X_3 + w^2_{2,4} \cdot X_4 + b^2_1$$

$$Z^2_2 = w^2_{2,1} \cdot X_1 + w^2_{2,2} \cdot X_2 + w^2_{2,3} \cdot X_3 + w^2_{2,4} \cdot X_4 + b^2_2$$

The linear output goes through an activation function $f^2_{a,i}()$ which each neuron can have a different activation function, but to simplify the formulation lets say $f^2_a()$ which means the whole layer will have the same activation function so the final outputs of $l = 2$ layer or the inputs of the next layer $(l = 3)$ will be:

$$a^2_1 = f^2_a(Z^2_1)$$
$$a^2_2 = f^2_a(Z^2_2)$$

In the next layer $(l = 3)$, which is the last hidden layer, the model has 4 neurons each with 2 weights (the number of outputs in the previous layer):

$$Z^3_1 = w^3_{1,1} \cdot a^2_1 + w^3_{1,2} \cdot a^2_2 + b^3_1$$

$$Z_2^3 = w_{2,1}^3 \cdot a_1^2 + w_{2,2}^3 \cdot a_2^2 + b_2^3$$

$$Z_3^3 = w_{3,1}^3 \cdot a_1^2 + w_{3,2}^3 \cdot a_2^2 + b_3^3$$

$$Z_4^3 = w_{4,1}^3 \cdot a_1^2 + w_{4,2}^3 \cdot a_2^2 + b_4^3$$

So, final outputs of layer $l = 3$ will be:

$$a_1^3 = f_a^3(Z_1^3)$$
$$a_2^3 = f_a^3(Z_2^3)$$
$$a_3^3 = f_a^3(Z_3^3)$$
$$a_4^3 = f_a^3(Z_4^3)$$

The next layer which is the last layer ($l = L$) will have 4 inputs including a_1^3, a_2^3, a_3^3, a_4^3, meaning every neuron must have 4 weights. With single neuron (single output) in this layer, we have:

$$Z_1^4 = w_{1,1}^4 \cdot a_1^3 + w_{1,2}^4 \cdot a_2^3 + w_{1,3}^4 \cdot a_3^3 + w_{2,4}^4 \cdot a_4^3 + b_1^4$$

And the final output will be:

$$y = a_1^4 = f_a^4(Z_1^4)$$

7.6 Training the Model

How we find weights and biases for each neuron in layers? The process of calibrating or optimizing the weights and biases in an Artificial Neural Network (ANN) is typically done through a process called "training" or "learning." The goal of training is to adjust the parameters (weights and biases) of the network so that it can make accurate predictions on the given data. The most common approach to training an ANN is using an optimization algorithm called "Stochastic Gradient Descent" (SGD) or its variants.

Here's an overview of how the training process works:

1. **Initialization**: The weights and biases in the neural network are initialized with small

random values. It's crucial to start with random values to break any symmetry in the network.

2. **Forward Propagation**: During each training iteration (also known as an epoch), the input data is fed into the network, and the activations for each layer are calculated through forward propagation. The forward pass involves computing the weighted sum of inputs, applying the activation function, and passing it to the next layer.

3. **Loss Function**: A loss function is defined to quantify how far off the predictions are from the actual targets. The goal of training is to minimize this loss function.

4. **Backpropagation**: After the forward pass, the error (the difference between the predicted output and the actual target) is calculated using the chosen loss function. Backpropagation is then used to propagate this error backward through the network to update the weights and biases.

5. **Gradient Calculation**: In backpropagation, the gradients of the loss function with respect to each weight and bias are computed. These gradients indicate the direction and magnitude of adjustments needed to minimize the loss.

6. **Weight and Bias Update**: The gradients calculated in the previous step are used to update the weights and biases. The learning rate is a hyperparameter that determines the step size taken in the direction of the gradients. Smaller learning rates make smaller updates, while larger learning rates make larger updates. A typical learning rate value might be 0.001 or 0.01, but this can vary depending on the problem and network architecture.

7. **Repeat**: Steps 2 to 6 are repeated for multiple epochs or until the loss converges to a satisfactory value.

The training process continues until the neural network parameters reach a state where the loss is minimized, and the model makes accurate predictions on unseen data.

To find the weights and biases for each neuron in the layers using a learning rate (α), the weight and bias update equations are as follows:

For each weight $w_{i,j}^{l}$ connecting neuron i in layer l to neuron j in layer $l-1$ (j is also the number of weights for each neuron in the current layer l), and the bias b_{i}^{l} for neuron i in layer l, the updates are given by:

$$w_{i,j}^{l} \leftarrow w_{i,j}^{l} - \alpha \frac{\partial Loss}{\partial w_{i,j}^{l}}$$

$$b_i^l \leftarrow b_i^l - \alpha \frac{\partial Loss}{\partial b_i^l}$$

Where:

- α is the learning rate,
- $\frac{\partial Loss}{\partial w_{i,j}^l}$ is the partial derivative of the loss function with respect to $w_{i,j}^l$
- $\alpha \frac{\partial Loss}{\partial b_i^l}$ is the partial derivative of the loss function with respect to b_i^l.

The gradients are calculated using backpropagation, as mentioned in step 4 of the training process. The specific form of the gradient calculation depends on the loss function and the activation functions used in the network. The learning rate determines the step size in updating the weights and biases during each iteration of the optimization algorithm. A well-chosen learning rate can help in achieving faster convergence and better generalization on new data. However, setting the learning rate too high can lead to overshooting and instability, while setting it too low can result in slow convergence. It is often necessary to experiment with different learning rates to find the optimal value for a specific problem and architecture.

Let's do a simple example!

You must design your ANN model based on the given data. Let's say this the given data:

	x_1	x_2	y
sample 1	0.2	0.8	1
sample 2	0.7	0.4	0
⋮	⋮	⋮	⋮
sample m	0.1	0.1	1

Based on the given data I seems the output y, is either 0 or 1 (binary classification problem). So the output layer must have only one neuron, and an activation function that gives us a binary out put. So, I can use Sigmoid as activation function for the output layer:

$$y = f_a(Z) = \frac{1}{1 + e^{-Z}}$$

As a reminder, this functions returns values between zero and one. The following figure show th output of this function. The red line called threshold is at line with $sigmoid(Z) = 0.5$. Any number above this is rounded up to 1 ($y = 1$), and lower than this line is considered zero ($y = 0$).

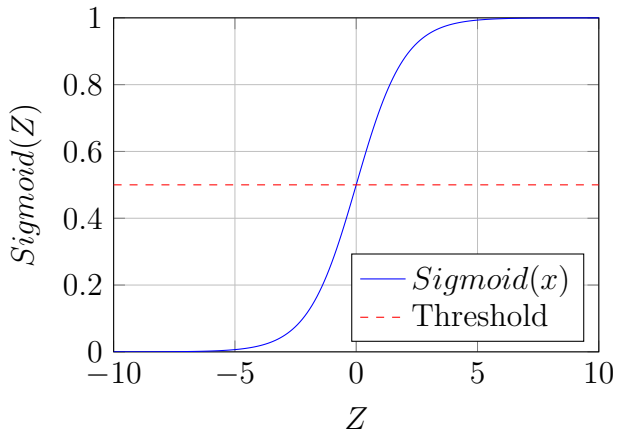


Figure 6: Sigmoid function plot with threshold line

I have two inputs (x_1 and x_2). **How many layers between input and output I need?** There is no absolute answer to this question, and you have to start with lower number of layers, and increase them gradually to get the right accuracy you are looking for. In this case, we want a simple example and we don't consider any hidden layer. The following ANN model has one neuron in the output layer (one y output) creating the linear combination of inputs (x_1 and x_2):

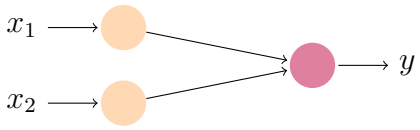


Figure 7: This ANN has two inputs, and one output layer with one neuron.

After designing the model architecture::

1. **Randomly initialize** the weight and bias for the output layer. In C I can use `rand()` function to do this. Since I have two inputs the neuron, this neuron must have two weights, lets say here is the randomized values for weights and bias:

$$\begin{bmatrix} w_1 & w_2 \end{bmatrix} = \begin{bmatrix} 0.3 & 0.5 \end{bmatrix}$$

$$\begin{bmatrix} b \end{bmatrix} = \begin{bmatrix} 0.1 \end{bmatrix}$$

2. **Formulation**:

$$Z = w_1 \cdot x_1 + w_2 \cdot x_2 + b_1$$

$$y = Sigmoid(Z)$$

3. **Training Loop** (for the first two epochs): For simplicity, we'll assume the following training data for one example:

   Epoch 1: Using **Sample 1** data in the table:

   - Forward Propagation:

$$Z = \begin{bmatrix} w_1 & w_2 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b = \begin{bmatrix} 0.3 & 0.5 \end{bmatrix} \times \begin{bmatrix} 0.2 \\ 0.8 \end{bmatrix} + 0.1 = 0.56$$

$$y_{preidcted} = Sigmoid(0.56) = 0.63$$

   - Compute the error (loss) for the output layer: The output of Sample 1 is $y_{real} = 1$. This is the type of error function I chose, based on the problem characteristic different loss function (error) might work better!

$$error = y_{real} - y_{preidcted} = 0.36354745971843361$$

   - Backpropagation and Weight Update: If $\alpha = 0.5$ is the learning rate, and δw_i and δb is the amount of changes on weights and bias:

$$w_1 \leftarrow w_1 + \delta w_1$$

$$\delta w_1 = \alpha \cdot y' \cdot error \cdot x_1$$

     Where y' is the derivative of $y = Sigmoid(Z)$ based on Z, So:

$$Sigmoid'(Z) = Sigmoid(Z) \cdot (Sigmoid(Z) - 1)$$

     or in this case we can say:

$$y' = y(y - 1)$$

     We can update w_2 and b the same way:

$$w_2 \leftarrow w_2 + \alpha \cdot y_{preidcted} \cdot (1 - y_{preidcted}) \cdot error \cdot x_2$$

$$b \leftarrow b + \alpha \cdot y_{preidcted} \cdot (1 - y_{preidcted}) \cdot error$$

     The updated values after the end are:

$w_1 = 0.3084117867258207$

$w_2 = 0.53364714690328274$

$b = 0.14205893362910343$

now the error has dropped to 0.0.3473611246515993. Let's say we have a 1000 data samples. we divide them into 10 groups each with 100 samples. We can update the values for using this 100 samples before we go to the next epoch!

Epoch 2 to end: In the previous epoch we used only **sample 1**. In this, epoch we want to use the second sample. In this case, the error value might drop or increase. But the point of ANN is to keep training over all the available data, until it reaches to the best optimized values. This is the reason why we need a lot of samples when we are training the data. If the model still was not able to decrease the error, one way is to increase the number of hidden layers and neurons to improve the ability of the model to learn! The more the neural network is bigger, more parameters are needed to be estimated, So we need more samples.

Here is the C code for the mentioned procedure. Be careful, this is not a geneal code for ANN. This is just a code for the mentioned ANN architecture and doing first 2 epochs. Use GDB or LLDB debugger to print the intermediate results. I'll send the code on teams.

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

#define INPUT_SIZE 2
#define LEARNING_RATE 0.5
#define EPOCHS 10000

typedef struct
{
 double weights[INPUT_SIZE];
 double bias;
} Perceptron;

double sigmoid(double x)
{
 return 1.0 / (1.0 + exp(-x));
}

double predict(Perceptron *perceptron, double input[INPUT_SIZE])
{
```

```
 double sum = 0.0;
 for (int i = 0; i < INPUT_SIZE; ++i)
 {
  sum += perceptron->weights[i] * input[i]; // in1*w1_1 + in2*
     x1_2
 }
 return sigmoid(sum + perceptron->bias); // activation_function(
    in1*w1_1 + in2*x1_2  +  b1)
 // sigmoid says if in f(x)
 // x => -inf so f(x)=0
 // x => +inf so f(x)=1
 // threshhold could be = 0.5!
}

void train(Perceptron *perceptron, double input[INPUT_SIZE], int
   label)
{
 double predicted = predict(perceptron, input);
 double error = label - predicted;

 for (int i = 0; i < INPUT_SIZE; ++i)
 {
  perceptron->weights[i] += LEARNING_RATE * error * predicted *
     (1 - predicted) * input[i];
 }

 perceptron->bias += LEARNING_RATE * error * predicted * (1 -
    predicted);
}

int main()
{
 // Sample dataset for XOR function
 double inputs[][INPUT_SIZE] = {
  {0.2, 0.8},
  {0.7, 0.4},
  {0.1, 0.1}};
 int labels[] = {1, 0, 1};
```

```
  Perceptron perceptron;

  // random initializing
  perceptron.weights[0] = 0.3;
  perceptron.weights[1] = 0.5;
  perceptron.bias = 0.1;
  // error = 0.36354745971843361

  // Training the perceptron
  for (int epoch = 0; epoch < EPOCHS; ++epoch)
  {
   for (int i = 0; i < 3; ++i)
   { // 4 is equal to the number of samples we have for train data
    train(&perceptron, inputs[i], labels[i]);
   }
  }

  // Validation
  printf("Predictions:\n");
  for (int i = 0; i < 3; ++i)
  {
   double prediction = predict(&perceptron, inputs[i]);
   printf("Input: (%f, %f), Prediction: %.2f, Label: %d\n", inputs
      [i][0], inputs[i][1], prediction, labels[i]);
  }
 }
```

You should see the following results:

```
Predictions:
Input: (0.200000, 0.800000), Prediction: 0.98, Label: 1
Input: (0.700000, 0.400000), Prediction: 0.03, Label: 0
Input: (0.100000, 0.100000), Prediction: 0.98, Label: 1
```

Based on Figure 6, I can say with `if` condition that any prediction higher than 0.5 is actually 1. So the actual result will be:

```
Predictions:
Input: (0.200000, 0.800000), Prediction: 1, Label: 1
Input: (0.700000, 0.400000), Prediction: 0, Label: 0
```

```
Input: (0.100000, 0.100000), Prediction: 1, Label: 1
```

Very Important! It is confusing a the beginning. Especially if you don't have background on mathematics. I strongly suggest to take look at these examples:

- Solved Example Back Propagation Algorithm,
- Backpropagation Solved Example,
- and A Step by Step Backpropagation Example.

Write them step by step on a paper, it might take you few hours! If you still had a problem to understand it, text me on Teams, I can do it in the lecture on Wednesday.

7.7 A simple example of ANN: Two Inputs and Tow Outputs - A Binary Classification

In binary classification, where the output variable y is binary (0 or 1), the most commonly used activation function for the output layer is the Sigmoid function. The Sigmoid function squashes the output to a value between 0 and 1, which can be interpreted as the probability of the input belonging to the positive class (class 1).

Download the latest version of `data.txt` and `main.c` from my GitHub. In this example, ANN is developed in `main.c` to train `data.txt`. Here is the structure of given data:

Table 1: Data in `data.txt`

Sample	x_1	x_2	y_1	y_2
1	0.961108775120079	0.576862767986208	0	1
2	0.891751713659224	0.619192234236843	0	1
3	0.955888707184407	0.729448225683374	0	1
...	...	...	...	...
48120	0.552926454894949	0.466761075682240	1	0

Based on the given data, we need a ANN model capable of predicting two binary outputs (y_1 and y_2) with given features (x_1 and x_2). Let's say if I give $x_1 = 0.961108775120079$ and $x_2 = 0.576862767986208$ to the model it should be able to predict $y_1^{predicted} = 0$ and $y_2^{predicted} = 1$, which are equal to the real values of y_1 and y_2 in the Table. So far what we know:

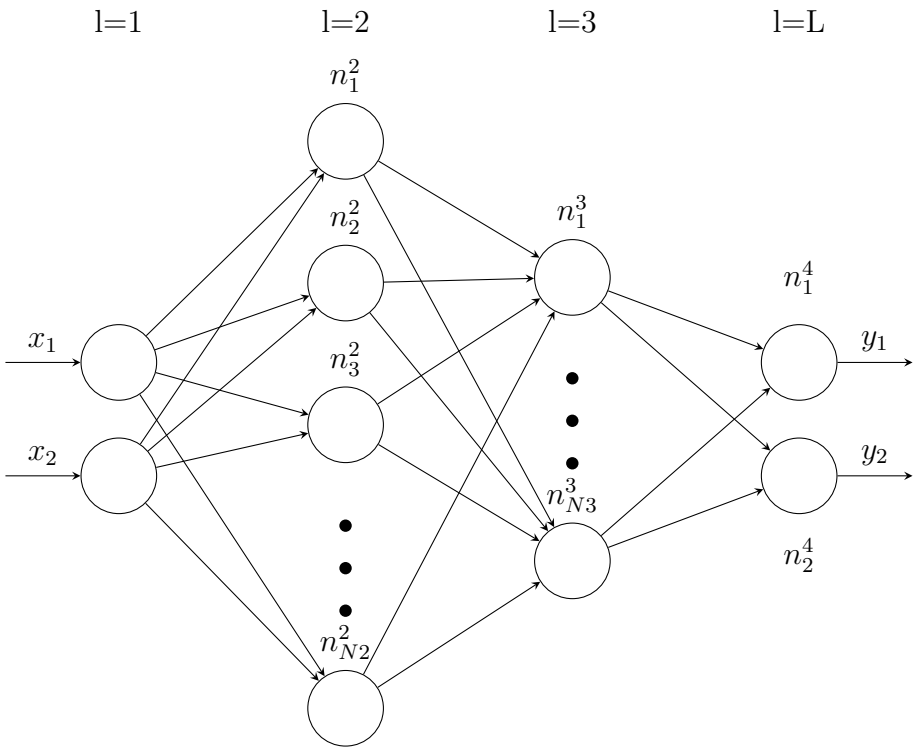
1. First layer (input layer l=1) has two inputs, including x_1 and x_2,
2. Last layer (output layer l=L) has two outputs (two neurons), including (y_1 and y_2),

3. Since the outputs are binary (either zero of one), we can use **Sigmoid()** activation function for both neurons in the last layer.

To answer the question that how many hidden layers we need to do the prediction, it is not always clear! As a rule to thumb, the more hidden layers and neurons model has, the more parameters (biases and weight) must be estimated, which can cause **over-fitting** in ANN! We must start with lower number of layers and neurons. If the accuracy for train data is not going up during the process of updating weights and bises (train) after a certain iteration (epoch), **one of the solutions** that might work is increasing the complexity of ANN (adding more hidden layers and neurons)! Here, based **on my experience on this specific data in the table**, I started with two hidden layers. First hidden layer (l=2) has 40 neurons ($N^2 = 40$), and the second hidden layer (l=3) has 20 neurons ($N^3 = 20$). What we know so far:

4. The total number of layers in the ANN model is 4 ($L = 4$).

This is how the ANN model looks like:



All the weights ans biases are randomly initialized within [-0.2, 0.2] using `<sodium.h>` library, defined by `initial_range` in the program. Probably you don't have the library installed, you need to install it first then during compile add `-lsodium` to link against the library. A learning rate of 0.005 is set by `Learning_rate`. The number of iterations to do forward and backward propagation is set by `epochs`. `train_split` shows how many samples are used to train the model, let's say a value of `0.003` means 0.3% data will be used to train model.

Currently the process of reading `data.txt` is happening inside the `main.c`. `main.c` is where you only call the functions and the real implementation must be before `int main ()`, the same as functions `random_double`, `sigmoid`, and `ForwardPass()`. We run the code during the class and we will discuss different parts of the code. Mind you that the code is like a mess in terms of structure and other issues, but here we are only focusing on ANN aspects of the program.

Very Important! Just don't forget this was a very basic introduction to ANN. There are many important concepts that we haven't discussed. If you want to start working in this field of study, I strongly suggest you to work with Python, R, or MATLAB, where the user is provided with many toolboxes. At one point, if you need to develop an ANN or any AI model with some features that these toolboxes do not have them, then you may need to write a code from scratch like what we did here.