Programming for Mechatronics - MECHTRON 2MP3

Pedram Pasandide (pasandip@mcmaster.ca)

GitHub

McMaster University.

Fall 2024

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, operating 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.

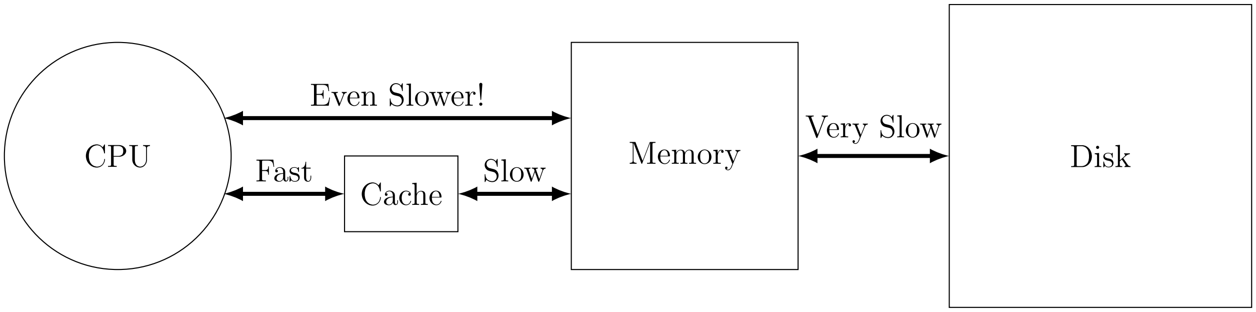
By the end of this section, you'll have a clearer picture of the intricate dance that occurs within the heart of your computer, demystifying the technology that surrounds us daily. So, let's embark on this journey together and peer into the fascinating world of computer components and connections.

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

```
Incldiung 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 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 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 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 short: 16 bits
Signed short range: -32768 to 32767
Unsigned short range: 0 to 65535

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

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:

```c
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:

```c
#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.

```
#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:

```c
#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 2,147,483,64**8**.

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.

```c
#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:

```
#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;
```

```
    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`:

```c
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:

```c
#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:

```c
#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:

```c
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:

```c
#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`:

```c
#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:

```c
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:

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

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

```c
#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.

```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++) {
  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>
```

```c
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:

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

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

```c
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:

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

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

```c
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.

```c
#include <stdio.h>

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:

```c
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:

```c
#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:

```
#include <stdio.h>

void printNumber() {
```

```c
 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:

```c
#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:

```c
#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>
```

```c
#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:

```c
#include <stdio.h>

// Global constant variable
const int MAX_VALUE = 100;

// 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:

```c
#include <stdio.h>

// 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.

2.10 Bit Manipulation

Bit manipulation refers to the act of algorithmically manipulating bits or binary digits, which are the fundamental units of data in computers. In C, bit-level operations are efficient and can be useful for tasks that involve hardware programming, encryption, compression, or optimizing performance.

C provides several bitwise operators that allow programmers to directly work with the bits of data, giving them fine-grained control over binary-level operations. **Why Bit Manipulation?** Bit manipulation is commonly used for:

- Setting or clearing specific bits in a number.
- Toggling bits on and off.
- Checking whether a particular bit is set.
- Shifting bits for efficient multiplication/division by powers of 2.
- Performing low-level optimizations that are often needed in systems programming or embedded systems.

Let's start by understanding the basic bitwise operators in C.

Table 1: Bitwise Operators in C

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR (Exclusive OR)
~	Bitwise NOT (One's Complement)
<<	Left Shift (Shifts bits to the left)
>>	Right Shift (Shifts bits to the right)

The following code shows the a basic example of bitwise operations in C:

```c
#include <stdio.h>

int main() {
 unsigned char a = 5;  // Binary: 00000101
 unsigned char b = 9;  // Binary: 00001001

 printf("a & b = %d\n", a & b);  // Bitwise AND
 printf("a | b = %d\n", a | b);  // Bitwise OR
 printf("a ^ b = %d\n", a ^ b);  // Bitwise XOR
 printf("~a = %d\n", ~a);        // Bitwise NOT
 printf("b << 1 = %d\n", b << 1); // Left Shift
 printf("b >> 1 = %d\n", b >> 1); // Right Shift

 return 0;
}
```

The above code will give us:

```
 a & b = 1
 a | b = 13
 a ^ b = 12
 ~a = -6
 b << 1 = 18
 b >> 1 = 4
```

Let's break down each of these operations:

- & (AND): Only the bits that are 1 in both numbers will be 1 in the result.
- | (OR): If any bit is 1 in either of the numbers, the corresponding bit will be 1 in the result.

- ˆ(XOR): The result is 1 if the bits are different, and 0 if they are the same.
- ˜(NOT): This inverts all the bits, turning 1 into 0 and vice versa.
- << (Left Shift): Shifts the bits to the left, filling with 0s.
- >> (Right Shift): Shifts the bits to the right, either preserving or dropping sign bits.

To check whether a bit at a specific position is set or not, we can use the AND operator & with a mask:

```
#include <stdio.h>

int isBitSet(int num, int pos){
 return (num & (1 << (pos-1))) != 0;
}

int main(){
 int number = 5; // Binary: 0000000000000101

 printf("Bit 2 is %s\n",isBitSet(number, 3)?"set.":"not set.");

 return 0;
}
```

The output:

```
Bit 2 is set.
```

One interesting application of the XOR operation is swapping two numbers without using a temporary variable:

```
#include <stdio.h>

int main() {
 int x = 10, y = 20;

 printf("Before swap: x = %d, y = %d\n", x, y);

 x = x ^ y;
 y = x ^ y;
 x = x ^ y;

 printf("After swap: x = %d, y = %d\n", x, y);

 return 0;
}
```

This will result:

```
Before swap: x = 10, y = 20
After swap: x = 20, y = 10
```

Another common bit manipulation task is counting the number of bits that are set to 1 in an integer. This is also known as the Hamming Weight:

```
#include <stdio.h>

int countSetBits(int num) {
 int count = 0;
 while (num > 0) {
  // Add 1 if the least significant bit is 1
  count += (num & 1);
  // Right shift to process the next bit
  num >>= 1;
 }
 return count;
}

int main() {
 // Binary: 0000000000001101 (3 bits set)
 int number = 13;

 // output: "Number of set bits: 3"
 printf("Number of set bits: %d\n", countSetBits(number));

 return 0;
}
```

Checking whether a number is a power of two is another common bitwise operation. A number that is a power of two has exactly one bit set to 1.

```c
#include <stdio.h>

int isPowerOfTwo(int num) {
 return (num > 0) && ((num & (num - 1)) == 0);
}

int main() {
 int number = 16;

 printf("%d %s a power of two.\n",number, isPowerOfTwo(number)?"
    is":"is not");

 return 0;
}
```

Bit manipulation is a powerful tool in C programming that allows fine control over binary data. Whether you're setting individual bits, swapping values, or performing optimizations, understanding these techniques will give you a strong foundation in low-level programming.

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.

This is the general format of Makefile(s):

```
target: prerquisites
(TAB) command line(s)
```

Where:

- Left side of `:` is the **target** to be built,
- Right side of `:` files (**Dependencies**) which **target** depends on,
- Line(s) start with a TAB if the next line is a command,
- Each line ends with `ENTER`,
- Line(s) started with `#` are Comments.

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

```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;
}
```

Using the following command line in the terminal, I can compile the code and create the executable object file `pedram`:

```
gcc -o pedram factorial.c -Wall -Wextra -std=c99
```

Now we want to do the same with a `Makefile`. Open a terminal and use `nano Makefile`. Copy and paste the following code and save the file.

```
pedram: factorial.c
 gcc -o pedram factorial.c -Wall -Wextra -std=c99
```

Make sure there is a TAB space behind `gcc`, one four `Space bar` on keyboard. You can open the `Makefile` file on VScode, or on a Text Editor. `Makefile`s are better to be opened on Text Editor since they show the difference between four `Space bar` and TAB space. In this Makefile:

- `pedram` is specified as the target,
- `factorial.c` is listed as a prerequisite for `pedram`, meaning `pedram` depends on, `factorial.c`. This ensures that if `factorial.c` changes, `pedram` will be rebuilt,
- The command `gcc -o pedram factorial.c` is used to compile `factorial.c` **directly** into the `pedram` executable without generating an intermediate object file.

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`. `make` searches for a file with name **makefile** in the current directory I If there is no such a file, make searches for a file with name **Makefile**. This is what **I** see in your **terminal**:

```
pedram@pedram-GL553VE:~/MECHTRON2MP3/prac/lec$ make
gcc -o pedram factorial.c -Wall -Wextra -std=c99
```

Check the directory and you will find the object file `pedram`, and you can execute the program the same as always with `./pedram`. If I remove the TAB before the command line in the `Makefile`, like:

```
pedram: factorial.c
gcc -o pedram factorial.c -Wall -Wextra -std=c99
```

Then, in the terminal I see the following error:

```
Makefile:15: *** missing separator.  Stop.
```

If I keep the TAB, but remove the line `pedram:  factorial.c`, then I get the following message:

```
Makefile:15: *** recipe commences before first target.  Stop.
```

Both mean the object file `pedram` is not made yet. Every time I use `make`, file(s) will be created in the current directory. In this case it is only the final executable file named `pedram`. Some times instead of removing them manually, I can use the `clean` command in the directory to remove the files. For example with the 2nd version of `Makefile`:

```
pedram: factorial.c
	gcc -o pedram factorial.c -Wall -Wextra -std=c99

clean:
	rm -f pedram
```

I can type in the terminal `make clean` and it will remove (`rm`) the file ()`-f`) named `pedram` from the current directory.

WARNING! The line `rm -f` will remove any file name given from the directory. For example, if I use `rm -f pedram <other files>`, the other files will be removed!!

In the context of your `Makefile` examples, **macros** (also known as variables in Make) are used to define reusable values that can be referenced throughout the `Makefile`. Several useful macros are:

- `CC`: Represents the compiler (`gcc`).
- `CFLAGS`: Represents the compiler flags (`-Wall -Wextra -std=c99`).
- `EXECUTABLE`: Represents the name of the executable (`pedram`).
- `SRC`: Represents the source file (`factorial.c`).

Why macros are useful? Making the long story short, macros in `Makefiles` improve readability, maintainability, consistency, flexibility, and help avoid repetition, making it easier to manage and modify complex build processes. With Macros I can write the same `Makefile` like:

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99
EXECUTABLE = pedram
SRC = factorial.c

$(EXECUTABLE): $(SRC)
	$(CC) $(CFLAGS) -o $@ $^
```

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

The way you program is **unique** like your **handwriting**, or unique to the way you think. I can create the object file `pedram`, with another type of handwriting:

```
pedram: factorial.o
	gcc -o pedram factorial.o

factorial.o: factorial.c
	gcc -c factorial.c -Wall -Wextra -std=c99

clean:
	rm -f pedram factorial.o
```

`factorial.o` is an object file generated by compiling the `factorial.c` source file. In C and C++ programming, when you compile a source file (`.c` file), the compiler produces an object file (`.o` file) as an intermediate step before creating the final executable. Here's what each line does:

1. `pedram:  factorial.o`: This line indicates that the target `pedram` depends on `factorial.o`. In other words, before creating the `pedram` executable, Make needs to ensure that `factorial.o` is up-to-date.

2. `gcc -o pedram factorial.o`: This is the rule to build the `pedram` executable. It specifies that `pedram` depends on `factorial.o`, and to create `pedram`, the command:

   `gcc -o pedram factorial.o`

   is executed. This command links the `factorial.o` object file to produce the final executable named `pedram`.

3. `factorial.o:  factorial.c`: This line indicates that the `factorial.o` object file depends on `factorial.c`. If `factorial.c` is newer than `factorial.o`, Make will rebuild `factorial.o`.

4. `gcc -c factorial.c`: This is the rule to compile `factorial.c` into `factorial.o`. The `-c` flag tells the compiler to generate the object file without linking, so it produces `factorial.o` instead of an executable.

Why did I add `-Wall -Wextra -std=c99` to `factorial.o:  factorial.c` not the other one? After `make` in the terminal I get:

```
gcc -c factorial.c -Wall -Wextra -std=c99
gcc -o pedram factorial.o
```

If I do `make clean`, then in the terminal I get:

```
rm -f pedram factorial.o
```

Kind of IMPORTANT! After I pass the command `make` in the terminal, I create `pedram` executable object file, and `factorial.o` object file. I can execute the program by `./pedram`, but not with `./factorial.o`. Now let's **remove** the `-c` while doing `factorial.o:  factorial.c`. The this is what I see in the terminal:

```
    pedram@pedram-GL553VE:~/MECHTRON2MP3/prac$ make
    gcc factorial.c -Wall -Wextra -std=c99
    gcc -o pedram factorial.o
    /usr/bin/ld: cannot find factorial.o: No such file or directory
    collect2: error: ld returned 1 exit status
    make: *** [Makefile:2: pedram] Error 1
```

This means that the line 2 (`gcc -o pedram factorial.o`), was not executed, meaning the object file `pedram` was not created. `pedram` is depended on `factorial.o` which tells me potentially the problem is `factorial.o`. By **removing** `-c` the source code `factorial.c` is directly compile into the executable file `factorial.o`. In terminal I can run the program this time with `./factorial.o`.
My point is intentionally introduce errors to Makefiles we have here by removing some syntax. Then try to logically find the issue. Les's say in the next version of `Makefile`, remove `-c` or `$<`.

With Macros I can write the same `Makefile` with the following format:

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99
EXECUTABLE = pedram
SRC = factorial.c
OBJ = factorial.o

$(EXECUTABLE): $(OBJ)
 $(CC) -o $@ $^
```

```
$(OBJ): $(SRC)
	$(CC) -c $(CFLAGS) $< -o $@

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

In this Makefile:

- `CC` is the compiler macro.
- `CFLAGS` contains compiler flags.
- `EXECUTABLE` is the macro for the executable name.
- `SRC` is the source file macro.
- `OBJ` is the object file macro.

In a Makefile, `$@` and `$^` are automatic variables used to represent the target and all prerequisites of a rule, respectively. Also `$<` is an automatic variable that represents the first prerequisite of the target.

- `$@` is replaced with the name of the target, which is `$(EXECUTABLE)`, i.e., `pedram`.
- `$^` is replaced with the list of prerequisites, which is `$(OBJ)`, i.e., `factorial.o`.
- `$<` is replaced with the name of the first prerequisite, which in this case is `$(SRC)`, or `factorial.c`, or the right side.

Using the command `make -p` line, you will find all the symbols we use in the makefile. Not really user-friendly to follow though!

Using macros makes it easier to modify the `Makefile` if necessary, as you only need to change the values of the macros. For example, if you change the source file or add more source files, you only need to update the `SRC` macro. Similarly, changing the compiler or compiler flags can be done by modifying the corresponding macros.

Keep the latest version of `Makefile`, and pass `make` command in the terminal. We will see:

```
gcc -c -Wall -Wextra -std=c99 factorial.c -o factorial.o
gcc -o pedram factorial.
```

If I do it one more time I get:

```
make: 'pedram' is up to date.
```

Pass the command `make clean`. **Then**, execute `make -n`. `make -n` is used to perform a "dry run" of a Makefile. It doesn't actually execute the commands specified in the `Makefile`; instead, it prints the commands it would execute without actually executing them. This is useful for understanding what actions make would take without affecting the filesystem.

Use `cat -v -t -e Makefile` in the terminal. This is what I see in the terminal:

```
CC = gcc$
CFLAGS = -Wall -Wextra -std=c99$
EXECUTABLE = pedram$
SRC = factorial.c$
OBJ = factorial.o$
$
$(EXECUTABLE): $(OBJ)$
^I$(CC) -o $@ $^$
$
$(OBJ): $(SRC)$
^I$(CC) -c $(CFLAGS) $< -o $@$
$
clean:$
^Irm -f $(EXECUTABLE) $(OBJ)$
$
# this is a comment$
```

We know the command `cat`. The flags are:

- `-v`: Display non-printing characters visibly. This option shows non-printing characters in a way that they can be easily identified, such as showing tabs as `Î`.
- `-t`: Display tab characters (`Î`) as `Î`. This option specifically handles tab characters and displays them as `Î`.
- `-e`: Display end-of-line markers (`$`) at the end of each line. This option adds a dollar sign `$` at the end of each line, indicating the end of the line.

Reminder! There is a `TAB` space before the line starting with `$(CC) ...` and `rm -f ...` (commands). 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`. The difference is clear in the textbfText Editor.

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:

```c
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

int square(int num);

#endif
```

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

```c
#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
```

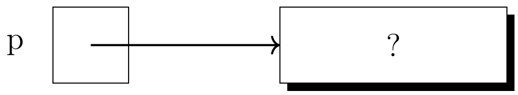
When we declare a pointer variable like `int *p`;, the pointer `p` is capable of storing memory addresses that point to an integer. The unary operator `*` is used to **de-reference** a pointer, which means accessing the value at the memory location that the pointer points to. For example, if `p` points 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:

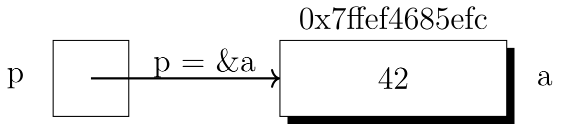
```
#include <stdio.h>

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.

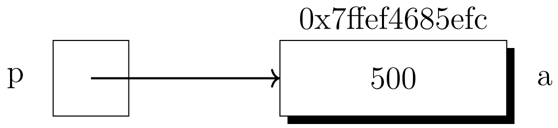


By de-referencing `p` with `*p`, we can access the value stored at the address pointed by `p`, which is the value of `a`. Here is the output in **my computer**. I am saying **my computer**, because the address given to the value `a` to save it, in your computer will be different. Actually, every time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

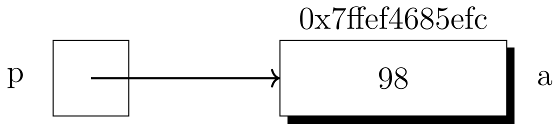
```
Value of 'a': 42
Address of 'a': 0x7ffef4685efc
Value of 'p' (address of 'a'): 0x7ffef4685efc
Value pointed by 'p': 42
```

```
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);
}
```

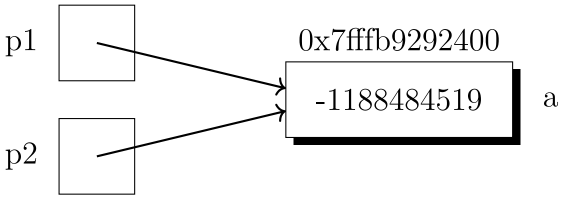
In the first part, p1 and p2 are declared as integer pointers. p1 is set to point to the address of variable a. Then p2 is assigned the value of p1. Now both p1 and p2 point to the address of a, and the value of a becomes 1.

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:

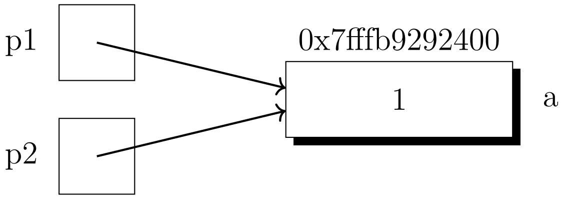
```
The initial values:
a=-1188484519
b=32767
```

```
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
```

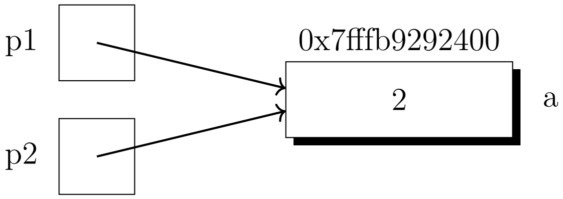
Let's go step by steps through schematic process! At the beginning `a` and `b` integers were declared without initialization. The same as `p1` and `p2` pointers. After `p1=&a`, `p1` will point at the the address of `a`. Using `p1=p2`, `p2` will point at the same address (`&a`) that `p1` is pointing at:



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);
```

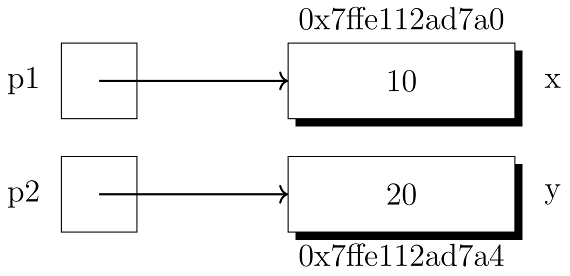
```
    printf("y=%d\n", y);
    printf("Value at address pointed by p1: %d\n", *p1);
    printf("Value at address pointed by p2: %d\n", *p2);

    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.

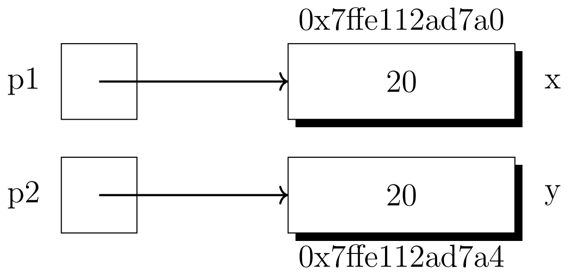
Finally, `*p1` is assigned the value of `*p2`. This means the value stored at the address pointed by `p2` (value of `y`) is **copied** to the address pointed by `p1` (value of `x`). After this step, both `*p1` and `*p2` become 20, and the addresses of `p1` and `p2` remain unchanged.

```
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:

```
#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:

```
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:

```c
#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:

```c
#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:

```c
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 *) 0x7ffffffbef0
(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.

Setting, Clearing, and Toggling Bits by passing pointers to a function are more advanced examples of bit manipulation:

- **Setting a bit**: To set (turn on) a specific bit, we use the OR operator | along with a mask.
- **Clearing a bit**: To clear (turn off) a specific bit, we use the AND operator & with the bitwise NOT of a mask.
- **Toggling a bit**: To flip a specific bit, we use the XOR operator ˆ.

In the following example, we pass the address of variables to a function (another scope), and update the value saved within the given address:

```
#include <stdio.h>

void setBit(int *num, int pos) {
 *num |= (1 << pos);
}

void clearBit(int *num, int pos) {
 *num &= ~(1 << pos);
}

void toggleBit(int *num, int pos) {
 *num ^= (1 << pos);
}

int main() {
 int number = 5;  // Binary: 0000000000000101

 // Set the 2nd bit: 0000000000000101 -> 0000000000001101
 setBit(&number, 2);
 printf("After setting bit 2: %d\n", number); // Output: 13

 // Clear the 2nd bit: 0000000000001101 -> 0000000000000101
 clearBit(&number, 2);
 printf("After clearing bit 2: %d\n", number); // Output: 5

 // Toggle the 1st bit: 0000000000000101 -> 0000000000000111
 toggleBit(&number, 1);
 printf("After toggling bit 1: %d\n", number); // Output: 7

 return 0;
}
```

Here is the output for the above code:

```
After setting bit 2: 5
After clearing bit 2: 1
After toggling bit 1: 3
```

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]);
 }
```

```c
}

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:

```c
#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!

```c
#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
```

```c
// 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**.

```c
#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++;
    }
    // OR we can use `int length = strlen(names[i]);` from `<string.h>`
    // `int length = sizeof(names[i])/sizeof(names[i][0]) - 1; ` will not work why?

    // `length+1` in `names[i][length+1]` to skip `\0`
    printf("Name %d: %s (Length: %d) - After '\\0': '%c'\n", i + 1, names[i],
        length, names[i][length+1]);
  }
}
```

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'`). The output:

```
Name 1: Anne-Marie (Length: 10) - After '\0': 'A'
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': ''
```

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:

```c
#include <stdio.h>
#include <string.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 = strlen(names[i]);

    // `length+1` in `names[i][length+1]` to skill `\0`
    printf("Name %d: %s (Length: %d) - After '\\0': '%c'\n", i + 1, names[i],
        length, names[i][length+1]);
  }
}
```

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					

This time the output is:

```
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’: ’’
```

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. **Use GDB and print out the addresses to see the sequence**.

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’).

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:

```c
#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* × *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 with 1024-byte (1 KB) increments on your machine. You can check `ulimit --help` for more information. In my computer it is `8192` KB (8192 * 1024 bytes or 8192÷1024 MB). 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”);
 printf(”Allocate memory: %0.1lf KB\n”,(double)(size * sizeof(int)/1024));
}

void allocateOnHeap(const int size) {


 int* largeArray = (int*)malloc(size * sizeof(int)); // Allocate the array
     on the heap
 // Check if memory allocation is successful to allocate (size * sizeof(
    int)) bytes
 if (largeArray != NULL) {
  printf(”Heap memory allocation Succeed!\n”);
  printf(”Allocate memory: %0.1lf KB\n\n”,(double)(size * sizeof(int)
     /1024));
  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));

  // Knowing that each integer takes 4 bytes
  // the maximum number of integers I can Save on stack is:
  // 8192 * 1024 (bytes)/ 4 = 2,097,152

  // this show the maximum integer value I can save by type ‘int‘
  printf(”Maximum possible value of ‘int‘: %d\n\n”, INT_MAX);

  // SIZE how many integer we can save?
  int SIZE = 2097152; // much less than 2,147,483,647 (integer overflow)

  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 = 2097152`) 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 possible value of ‘int‘: 2147483647


Attempting to allocate on heap...
Heap memory allocation Succeed!
```

```
Allocate memory: 8192.0 KB

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 it is not! That's why I have defined `int SIZE = 2097152;` 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:

```c
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:

```c
#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
```

```c
  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 have 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:

```c
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:

```
#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

```
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
```

```c
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:

```c
#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.

Sometimes you might see `people->` instead of `people.` when we try to access to the member. Take a look at the following code:

```
#include <stdio.h>

struct Person {
 char name[20];
 int age;
};

```

```c
int main() {
 // Creating a struct instance
 struct Person person1 = {"John", 30};

 // Using the dot operator
 printf("Name: %s, Age: %d\n", person1.name, person1.age);

 // Creating a pointer to a struct
 struct Person *personPtr = &person1;

 // Using the arrow operator
 printf("Name: %s, Age: %d\n", personPtr->name, personPtr->age);

 return 0;
}
```

In this example, `person1` is a `struct` instance, so we use the dot operator to access its members (`person1.name` and `person1.age`). `personPtr` is a pointer to a `struct`, so we use the arrow operator to access its members (`personPtr->name` and `personPtr->age`). Both approaches achieve the same result, but the choice depends on whether the `struct` is accessed through a pointer or directly.

I can also have `struct` within another `struct`. In the following example, we'll create a program that models a library system, where each book has information about its title, author, and publication date, and each library member has information about their name, ID, and the books they have borrowed. Copy and paste the following code in `structure.c`.

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

#define MAX_BB 2       // Maximum Borrowed Books allowed per member
#define MAX_NAME 50
#define MAX_BOOKS 10   // Storage/Memory capacity to keep books
#define MAX_MEMBERS 5 // Storage/Memory capacity to save members

// Struct representing a book
struct Book {
 char title[MAX_NAME];
 char author[MAX_NAME];
 int publicationYear;
};
```

```
// Struct representing a library member
struct LibraryMember {
 char name[MAX_NAME];
 int id;
 struct Book BB[MAX_BB];
 int numBB;
};

// Function to add a book to the library collection
void addBook(struct Book *library, int *numBooks, const char *title, const char *
    author, int publicationYear) {
 if (*numBooks >= MAX_BOOKS) {
  printf("Library is full. Cannot add more books.\n");
  return;
  // or we can use realloc() to increase the size of `library`
 }
 strcpy(library[*numBooks].title, title);
 strcpy(library[*numBooks].author, author);
 library[*numBooks].publicationYear = publicationYear;
 (*numBooks)++;
}

// Function to start a new library membership
void addMember(struct LibraryMember *members, int *numMembers, const char *name,
    int id) {
 if (*numMembers >= MAX_MEMBERS) {
  printf("Membership capacity reached. Cannot add more members.\n");
  return;
  // or we can use realloc() to increase the size of `members`
 }
 strcpy(members[*numMembers].name, name);
 members[*numMembers].id = id;
 members[*numMembers].numBB = 0;
 (*numMembers)++;
}

// Function to allow a member to borrow a book
void borrowBook(struct LibraryMember *member, struct Book book) {
 if (member->numBB >= MAX_BB) {
  printf("%s has reached the maximum limit of borrowed books.\n", member->name);
  return;
 }
 member->BB[member->numBB] = book;
 member->numBB++;
 printf("%s borrowed \"%s\" by %s.\n", member->name, book.title, book.author);
}
```

```
void printBorrowedBooks(struct LibraryMember member) {
 printf("Borrowed books by %s (ID: %d):\n", member.name, member.id);
 if (member.numBB == 0) {
  printf("No books borrowed.\n");
 } else {
  for (int i = 0; i < member.numBB; i++) {
   printf("%d. \"%s\" by %s, published in %d\n",
   i + 1,
   member.BB[i].title,
   member.BB[i].author,
   member.BB[i].publicationYear);
  }
 }
}

int main() {
 // Allocate memory for library books
 struct Book *library = (struct Book *)malloc(MAX_BOOKS * sizeof(struct Book));
 int numBooks = 0;

 // Adding books to the library
 addBook(library, &numBooks, "The Great Gatsby", "F. Scott Fitzgerald", 1925);
 addBook(library, &numBooks, "What Is Man?", "Mark Twain", 1906);
 addBook(library, &numBooks, "To Kill a Mockingbird", "Harper Lee", 1960);
 addBook(library, &numBooks, "1984", "George Orwell", 1949);

 // Allocate memory for library members
 struct LibraryMember *members = (struct LibraryMember *)malloc(MAX_MEMBERS *
     sizeof(struct LibraryMember));
 int numMembers = 0;

 // Adding members
 addMember(members, &numMembers, "John Brown", 1001);
 addMember(members, &numMembers, "Alice Johnson", 1002);

 // Borrowing books
 borrowBook(&members[0], library[0]);  // John Brown borrows "The Great Gatsby"
 borrowBook(&members[1], library[1]);  // Alice Johnson borrows "What Is Man?"
 borrowBook(&members[0], library[2]);  // John Brown borrows "To Kill a
     Mockingbird"
 borrowBook(&members[0], library[3]);  // John Brown tries to borrow "1984" but
     reaches limit

 printBorrowedBooks(members[0]);

 // Free allocated memory
 free(library);
```

```c
  free(members);

  return 0;
}
```

This is what you should see in the terminal.

```
John Brown borrowed "The Great Gatsby" by F. Scott Fitzgerald.
Alice Johnson borrowed "What Is Man?" by Mark Twain.
John Brown borrowed "To Kill a Mockingbird" by Harper Lee.
John Brown has reached the maximum limit of borrowed books.
Borrowed books by John Brown (ID: 1001):
1. "The Great Gatsby" by F. Scott Fitzgerald, published in 1925
2. "To Kill a Mockingbird" by Harper Lee, published in 1960
```

There is an issue with this code. The format is not perfectly readable since we have long code in a single file. It is better to separate the codes into multiple files, based on their purpose, context, etc. I think it might be more readable to have a `structure.c`, `functions.c`, and `functions.h`.

- `structure.c`:

```c
#include "functions.h"
#include <stdlib.h>

int main() {
 // Allocate memory for library books
 struct Book *library = (struct Book *)malloc(MAX_BOOKS *
    sizeof(struct Book));
 int numBooks = 0;

 // Adding books to the library
 addBook(library, &numBooks, "The Great Gatsby", "F. Scott
    Fitzgerald", 1925);
 addBook(library, &numBooks, "What Is Man?", "Mark Twain",
    1906);
 addBook(library, &numBooks, "To Kill a Mockingbird", "
    Harper Lee", 1960);
 addBook(library, &numBooks, "1984", "George Orwell", 1949);

 // Allocate memory for library members
```

```
    struct LibraryMember *members = (struct LibraryMember *)
       malloc(MAX_MEMBERS * sizeof(struct LibraryMember));
    int numMembers = 0;

    // Adding members
    addMember(members, &numMembers, "John Brown", 1001);
    addMember(members, &numMembers, "Alice Johnson", 1002);

    // Borrowing books
    borrowBook(&members[0], library[0]);  // John Brown borrows "The Great Gatsby"
    borrowBook(&members[1], library[1]);  // Alice Johnson borrows "What Is Man?"
    borrowBook(&members[0], library[2]);  // John Brown borrows "To Kill a Mockingbird"
    borrowBook(&members[0], library[3]);  // John Brown tries to borrow "1984" but reaches limit

    printBorrowedBooks(members[0]);

    // Free allocated memory
    free(library);
    free(members);

    return 0;
  }
```

- `functions.c`:

```
// functions.c
#include "functions.h"
#include <stdio.h>
#include <string.h>

// Function to add a book to the library collection
void addBook(struct Book *library, int *numBooks, const char *title, const
    char *author, int publicationYear) {
 if (*numBooks >= MAX_BOOKS) {
  printf("Library is full. Cannot add more books.\n");
  return;
  // or we can use realloc() to increase the size of ‘library‘
 }
```

```
  strcpy(library[*numBooks].title, title);
  strcpy(library[*numBooks].author, author);
  library[*numBooks].publicationYear = publicationYear;
  (*numBooks)++;
}

// Function to start a new library membership
void addMember(struct LibraryMember *members, int *numMembers, const char *
    name, int id) {
  if (*numMembers >= MAX_MEMBERS) {
    printf("Membership capacity reached. Cannot add more members.\n");
    return;
    // or we can use realloc() to increase the size of `members`
  }
  strcpy(members[*numMembers].name, name);
  members[*numMembers].id = id;
  members[*numMembers].numBB = 0;
  (*numMembers)++;
}

// Function to allow a member to borrow a book
void borrowBook(struct LibraryMember *member, struct Book book) {
  if (member->numBB >= MAX_BB) {
    printf("%s has reached the maximum limit of borrowed books.\n", member->
        name);
    return;
  }
  member->BB[member->numBB] = book;
  member->numBB++;
  printf("%s borrowed \"%s\" by %s.\n", member->name, book.title, book.
      author);
}

void printBorrowedBooks(struct LibraryMember member) {
  printf("Borrowed books by %s (ID: %d):\n", member.name, member.id);
  if (member.numBB == 0) {
    printf("No books borrowed.\n");
  } else {
    for (int i = 0; i < member.numBB; i++) {
      printf("%d. \"%s\" by %s, published in %d\n",
      i + 1,
      member.BB[i].title,
      member.BB[i].author,
      member.BB[i].publicationYear);
    }
  }
}
```

- `functions.h`:

```
// functions.h
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#define MAX_BB 2       // Maximum Borrowed Books allowed per member
#define MAX_NAME 50
#define MAX_BOOKS 10   // Storage/Memory capacity to keep books
#define MAX_MEMBERS 5 // Storage/Memory capacity to save members

// Struct representing a book
struct Book {
 char title[MAX_NAME];
 char author[MAX_NAME];
 int publicationYear;
};

// Struct representing a library member
struct LibraryMember {
 char name[MAX_NAME];
 int id;
 struct Book BB[MAX_BB];
 int numBB;
};

// Function prototypes
// Function to add a book to the library collection
void addBook(struct Book *library, int *numBooks, const char *title, const
    char *author, int publicationYear);

// Function to start a new library membership
void addMember(struct LibraryMember *members, int *numMembers, const char *
    name, int id);

// Function to allow a member to borrow a book
void borrowBook(struct LibraryMember *member, struct Book book);

// Print information about members
void printBorrowedBooks(struct LibraryMember member);

#endif // FUNCTIONS_H
```

You can use the following `Makefile` to compile and run the executable object file `library`.

```
library: structure.o functions.o
    gcc -o library structure.o functions.o
# Sometimes you might need to add "functions.h"!
structure.o: structure.c
functions.o: functions.c

clean:
    rm -f structure.o functions.o library
```

We can run the code by `./library`. What about flags `-Wall -Wextra -std=c99`? Updating the Makefile:

```
library: structure.o functions.o
    gcc -o library structure.o functions.o

structure.o: structure.c
    gcc -c structure.c -Wall -Wextra -std=c99
functions.o: functions.c
    gcc -c functions.c -Wall -Wextra -std=c99

clean:
    rm -f structure.o functions.o library
```

Now I get:

```
gcc -c structure.c -Wall -Wextra -std=c99
structure.c: In function ‘main’:
structure.c:7:17: warning: unused variable ‘book3’ [-Wunused-variable]
7 |     struct Book book3 = {"To Kill a Mockingbird", "Harper Lee", 1960};
|                  ^~~~~
gcc -c functions.c -Wall -Wextra -std=c99
functions.c: In function ‘borrowBook’:
functions.c:7:9: warning: unused variable ‘waste’ [-Wunused-variable]
7 |     int waste = 0;
|         ^~~~~
gcc -o library structure.o functions.o
```

If I do `make clean`, then use the command line `make -n` in the terminal, this is what I see:

```
gcc -c structure.c -Wall -Wextra -std=c99
gcc -c functions.c -Wall -Wextra -std=c99
gcc -o library structure.o functions.o
```

This is what happens in order to create the executable file `library`. I can write the same `Makefile` with Macros like:

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

# List of source files
SRCS = structure.c functions.c

# List of object files
OBJS = $(SRCS:.c=.o)

# Rule to build the executable
$(EXECUTABLE): $(OBJS)
    $(CC) -o $@ $^

# Rule to compile source files into object files
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

# Rule to clean the project
clean:
    rm -f $(OBJS) $(EXECUTABLE)
```

Using `make -n`, I have:

```
gcc -Wall -Wextra -std=c99 -c structure.c -o structure.o
gcc -Wall -Wextra -std=c99 -c functions.c -o functions.o
gcc -o library structure.o functions.o
```

5 Effective Code Development Practices

Effective code development practices are essential for writing efficient, maintainable, and reliable software. This section covers a range of topics that contribute to robust programming, including the use of `typedef` for simplifying type declarations, managing memory effectively to avoid leaks, and utilizing compiler optimizations for better performance. Profiling tools such as `gprof` and Intel VTune Profiler help analyze code execution, while GitHub and version control systems enable collaborative development and code management. Finally, proper documentation ensures that code remains understandable and maintainable over time, making these practices vital for both individual and team-based projects.

5.1 `typedef`

In C, `typedef` is a keyword used to create an alias for existing data types. This can make code more readable and easier to manage, especially when dealing with complex types like structures or function pointers. Using `typedef`, you can define new names for existing types, which simplifies declarations and enhances code clarity.

```c
#include <stdio.h>

// Defining a new type alias using typedef
typedef unsigned long ulong;

// Using typedef with a structure
typedef struct {
 int x;
 int y;
} Point;

int main() {
 // Using the new alias for unsigned long
 ulong largeNumber = 1000000;
 printf("Large Number: %lu\n", largeNumber);

 // Using the typedef for a structure
 Point p1;
 p1.x = 10;
 p1.y = 20;
 printf("Point p1: (%d, %d)\n", p1.x, p1.y);
```

```
  return 0;
}
```

In this example:

- `typedef` is used to define `ulong` as an alias for `unsigned long`.
- The `Point` structure is aliased using `typedef`, making the declaration of variables simpler and clearer.

5.2 Memory Leak

A memory leak in C occurs when dynamically allocated memory (using `malloc`, `calloc`, or `realloc`) is not properly deallocated with free. Over time, memory leaks can lead to wasted memory and reduced system performance, especially in long-running applications. Tools like `valgrind` are used to detect memory leaks and other memory-related issues in C programs.

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

int main() {
 // Dynamically allocate memory for an integer
 int *ptr = (int *)malloc(sizeof(int));
 if (ptr == NULL) {
  printf("Memory allocation failed\n");
  return 1;
 }

 // Assign value to the allocated memory
 *ptr = 42;
 printf("Value: %d\n", *ptr);

 // Intentionally forgetting to free the memory causes a memory leak
 // free(ptr);  // Uncommenting this line will prevent the memory leak

 return 0;
}
```

In this example, we allocate memory for an integer using `malloc`, but if we don't call `free(ptr);`, the allocated memory is not returned to the system, resulting in a memory leak.

To detect memory leaks, you can use `valgrind`, a tool for debugging and profiling programs. Here's how you would run the example program through `valgrind`:

```
gcc -o memleak_example memleak_example.c
valgrind --leak-check=full ./memleak_example
```

Valgrind will provide detailed information on memory leaks, including how much memory was leaked and where it was allocated.

5.3 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.

```c
#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`. Save this code as `Matrix_Multiplication.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++)
```

```c
 {
  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:

```c
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 run the code using the following command:

`gcc -O0 -o Matrix_Multiplication Matrix_Multiplication.c`

The CPU time in my computer is around 73 seconds. If I use the following command:

`gcc -O3 -o Matrix_Multiplication Matrix_Multiplication.c`

The time taken by CPU is around 47 second in **my computer**.

5.4 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.

5.4.1 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!

5.4.2 VTune Profiler - From Installation to Case Study - (Optional topic)

IMPORTANT! VTune is a very useful profiling tool. However, VTune only works on Intel processors, and recent semesters, it created issues for some students. I've decided to make this topic an optional section for those of you who are really curious and want to explore it further, beyond what's required for this course.

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:

Installing VTune on Linux:

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.c` we had for $A \times B = C$. But this time I made a change in the code and instead of using `#define N 2000`, I receive the input from user. This is the final version of `Matrix_Multiplication.c`:

```c
// <The libraries are inlcuded>



void matrixMultiplication(double **A, double **B, double **C, int
    N)
{
 // <This part of the code is the same>
}

void freeMatrix(double **matrix, int N)
{
 // <This part of the code is the same>
}

int main(int argc, char *argv[]) {

 // Check if a filename argument is provided
 if (argc != 2) {
  printf("Usage: %s <Matrix Dimension>\n", argv[0]);
  return 1;
 }

 const int N = atoi(argv[1]);

 // <This part of the code is the same>

 // Call the matrix multiplication function
 matrixMultiplication(A, B, C, N);

 // <This part of the code is the same>

 // Free allocated memory
 freeMatrix(A,N);
 freeMatrix(B,N);
 freeMatrix(C,N);
```

```
    return 0;
  }
```

Step 1: Compile the code with no optimization flag:

`gcc -o Matrix_Multiplication Matrix_Multiplication.c`.

This means the executable file's name is `Matrix_Multiplication`, and I can I run the program by:

`./Matrix_Multiplication 2000`,

where `2000` is the value for `N`.

Step 2: Open VTune by `vtune-gui` command in the terminal Figure 1.

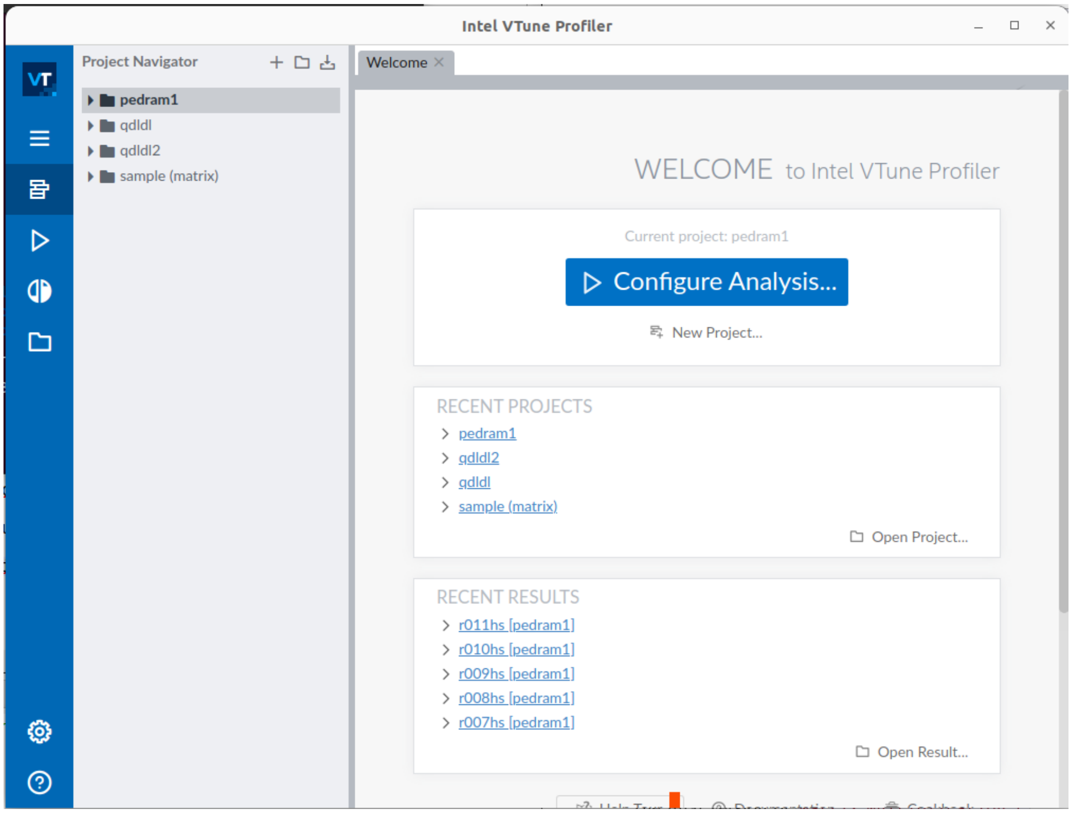
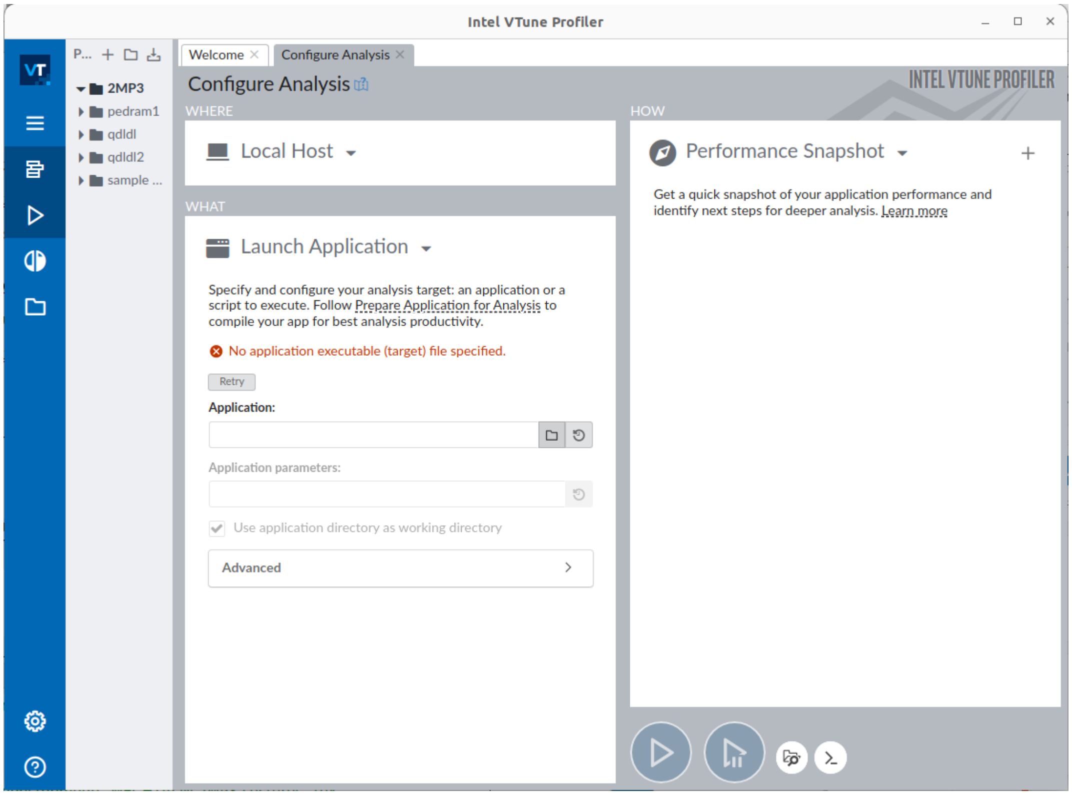


Figure 1: VTune Environment

In at the left bar click section Project Navigator, create a new project with a name you want like "2MP3" in the section "Project name." You don't need to change the "Location" section. Now you have to able to see the following window (Figure 2):



Configure Analysis

Get a quick snapshot of your application performance and identify next steps for deeper analysis. Learn more

Specify and configure your analysis target: an application or a script to execute. Follow Prepare Application for Analysis to compile your app for best analysis productivity.

No application executable (target) file specified.

Application:

Application parameters:

☑ Use application directory as working directory

Figure 2: VTune Project Configuration

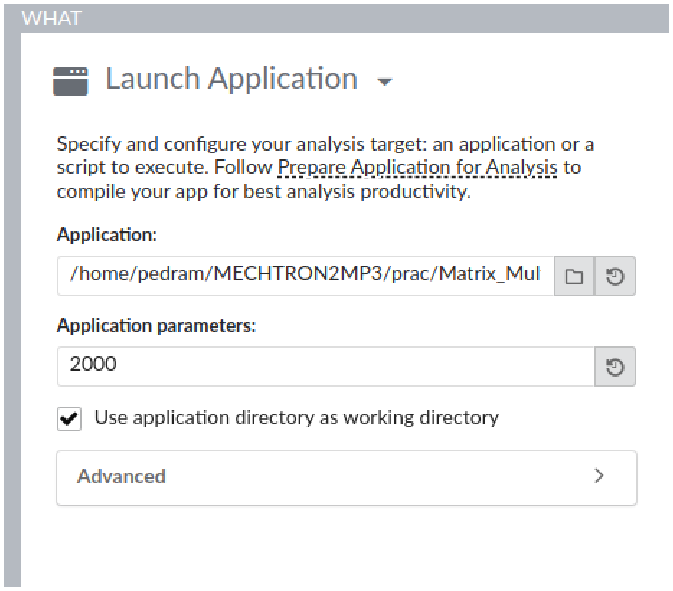
Step 3: Under the section ”Application” choose the path to the executable file that you want to analyze. I this case, it is `Matrix_Multiplication` object file that we create in the **Step 1**. In **my computer** it is:

`/home/pedram/MECHTRON2MP3/prac/Matrix_Multiplication`.

In the section ”Application parameters” you have to give the inputs to run the program. In this case we have only on input and it is the value for N or the dimension of the matrices:

`2000`

Figure 3 is what you should have so far:



Specify and configure your analysis target: an application or a script to execute. Follow Prepare Application for Analysis to compile your app for best analysis productivity.

Application:

/home/pedram/MECHTRON2MP3/prac/Matrix_Mul

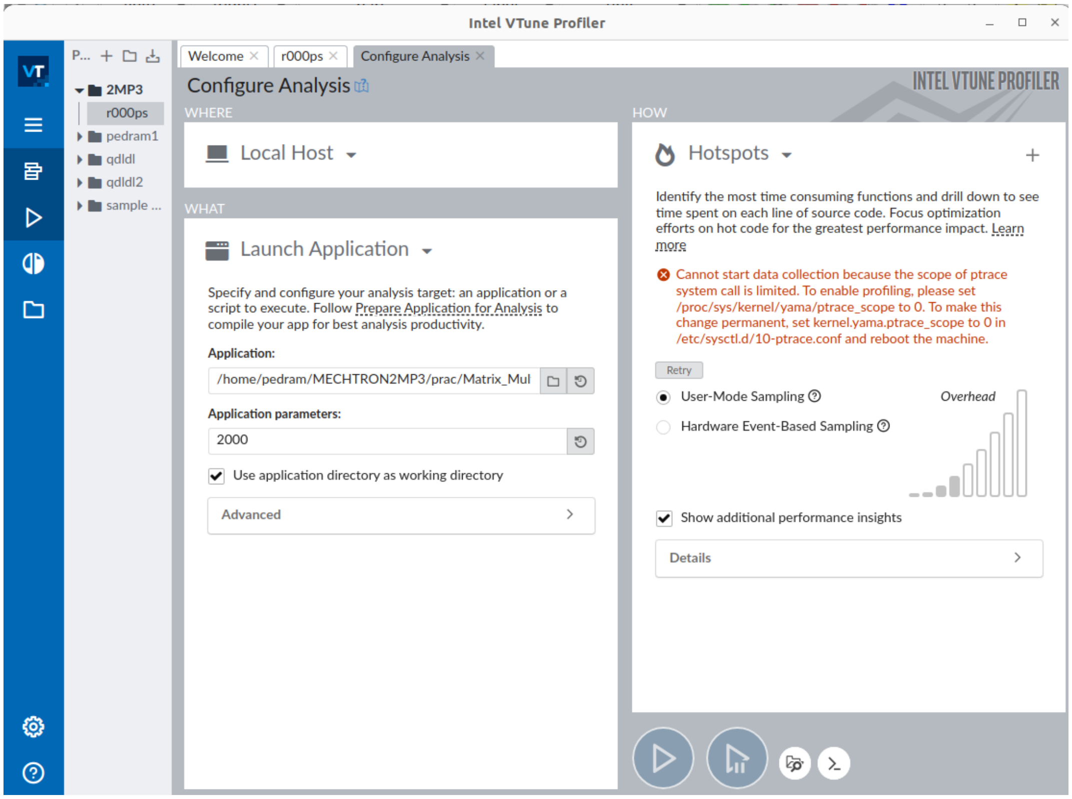
Application parameters:

2000

☑ Use application directory as working directory

Figure 3: VTune Project Configuration

Step 4: Click on the run button to start the analysis. Navigate to the ”Hotspots” section and you will see the following red message in the right side of the window (Figure 4):



Specify and configure your analysis target: an application or a script to execute. Follow Prepare Application for Analysis to compile your app for best analysis productivity.

Application:

/home/pedram/MECHTRON2MP3/prac/Matrix_Mul

Application parameters:

2000

☑ Use application directory as working directory

Identify the most time consuming functions and drill down to see time spent on each line of source code. Focus optimization efforts on hot code for the greatest performance impact. Learn more

Cannot start data collection because the scope of ptrace system call is limited. To enable profiling, please set /proc/sys/kernel/yama/ptrace_scope to 0. To make this change permanent, set kernel.yama.ptrace_scope to 0 in /etc/sysctl.d/10-ptrace.conf and reboot the machine.

● User-Mode Sampling

○ Hardware Event-Based Sampling

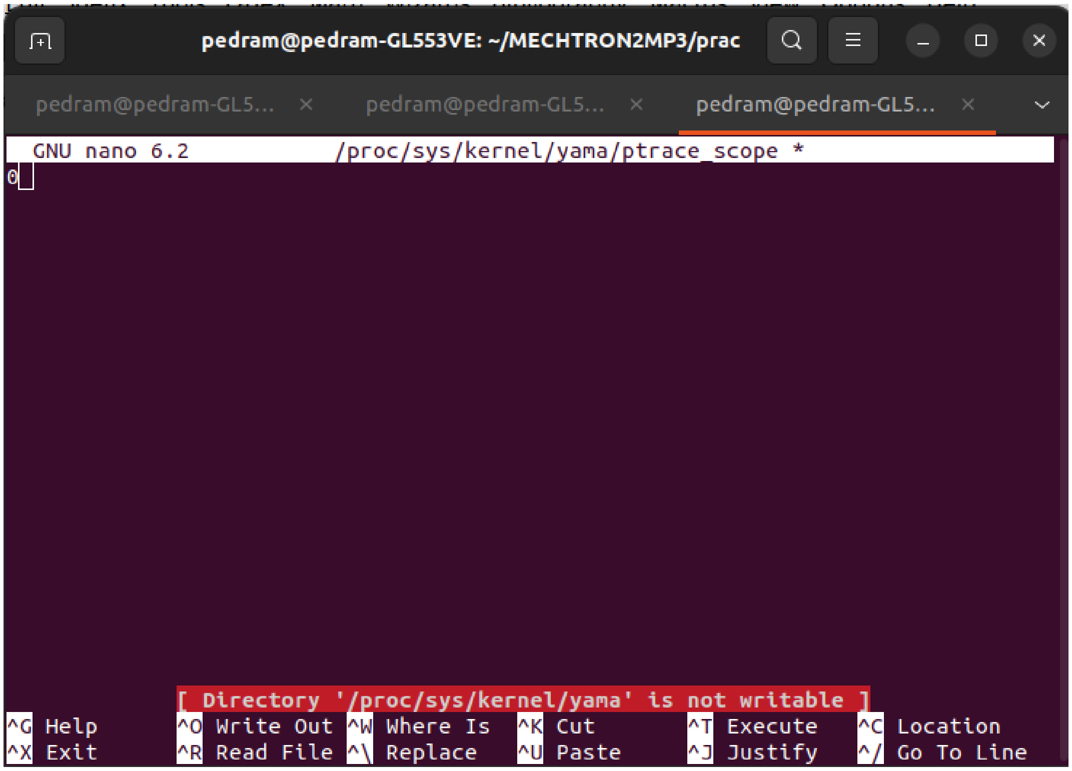
☑ Show additional performance insights

Figure 4: VTune Project Configuration - Error

Open a new terminal an run the following command:

```
sudo nano /proc/sys/kernel/yama/ptrace_scope
```

Change the value from 1 to zero like (Figure 5):

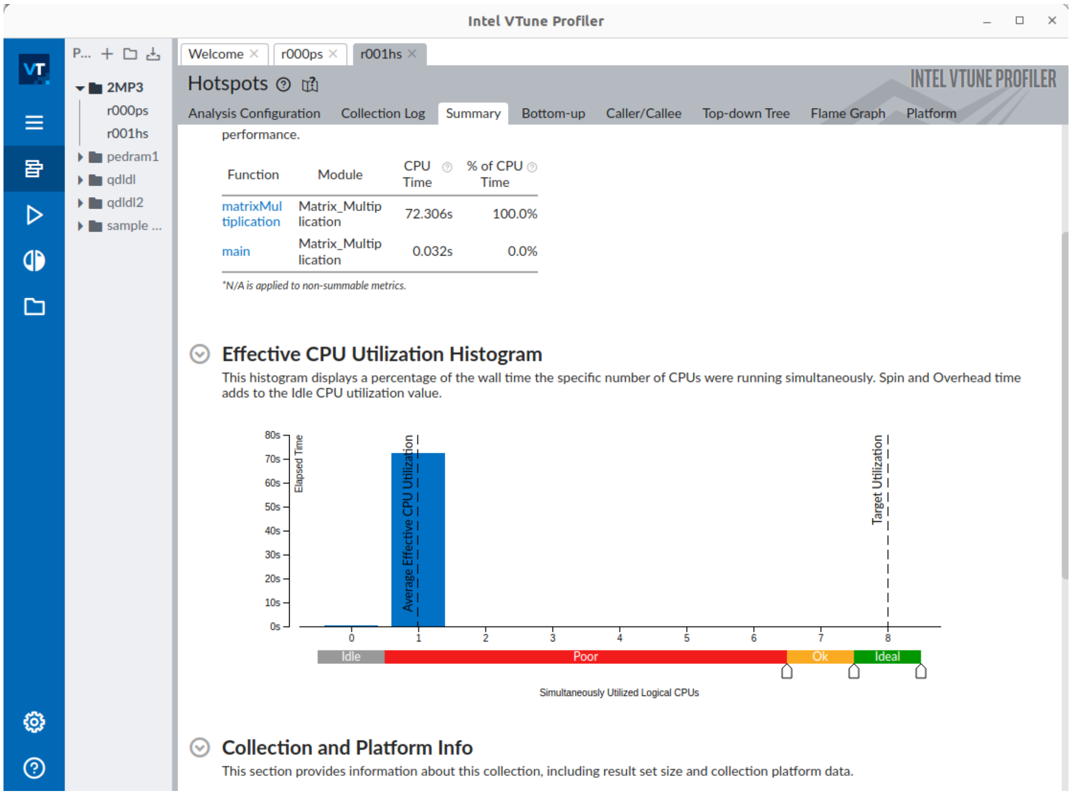


```
  GNU nano 6.2              /proc/sys/kernel/yama/ptrace_scope *
0

                 [ Directory '/proc/sys/kernel/yama' is not writable ]
^G Help       ^O Write Out  ^W Where Is   ^K Cut        ^T Execute    ^C Location
^X Exit       ^R Read File  ^\ Replace    ^U Paste      ^J Justify    ^/ Go To Line
```

Figure 5: VTune Project Configuration - Error - Solve

Go back to VTune environment and in Figure 4 press the button `Retry`. The error should be gone by now. Click the button **Start**. Now the VTune is executing the program and collecting information.

Step 5: Under the window Summary you can see some information like the CPU time or how much your program is efficient (Figure 6). There are a lot more information here but I leave how you read them to you!



performance.

Function	Module	CPU Time	% of CPU Time
matrixMultiplication	Matrix_Multiplication	72.306s	100.0%
main	Matrix_Multiplication	0.032s	0.0%

*N/A is applied to non-summable metrics.

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Figure 6: VTune Results

This time I want to conduct another analysis but with optimization flag `-O3`. Compile the program with:

`gcc -O3 -o Matrix_Multiplication Matrix_Multiplication.c`.

Go back to VTune environment and click Configure Analysis. This time the object file `Matrix_Multiplication` is updated by using only the optimization flag -O3. Run the analysis in the same way. The main difference you see is that the CPU run time is dropped (in **my computer** it dropped from around 72 second to 45 seconds).

Installing VTune on WSL:

To install Intel VTune Profiler on Ubuntu running within the Windows Subsystem for Linux (WSL), you'll need to follow specific steps to set up VTune within this environment. Please note that running performance profiling tools like VTune directly within WSL might have limitations due to differences in the kernel and system behavior between Linux and Windows.

1. Ensure that your Ubuntu installation within WSL is up-to-date by running:

```
sudo apt update
sudo apt upgrade
```

2. Download and Install VTune:

- 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. You must download the Linux version.

3. Install the Downloaded Package:

Navigate to the directory where the downloaded VTune installer is located. You should have a file with `.sh` extension. You can install it by `sudo sh ./`. For example:

```
sudo sh ./l_oneapi_vtune_p_2024.0.0.49503_offline.sh
```

If you have trouble to navigate to the directory where the downloaded file is you can open the file explorer on WSL using `explorer.exe .` which opens the current directory of your terminal.

Follow the installation process in the WSL terminal (not even VScode environment!). Follow the instruction to install VTune. First text you see on WSL terminal is:

```
INSTALLATION LOCATION: /opt/intel/oneapi
SPACE REQUIRED TO INSTALL: 1.8 GB          SPACE REQUIRED TO DOWNLOAD: 0 Bytes
CAN INTEGRATE WITH: Eclipse*.
 Intel Software Installer:  4.3.4.1166



 By continuing with this installation, you accept the terms and conditions of
 Intel End User License Agreement
 Accept & install   Accept & customize   Download Only   Decline & quit
```

Move within the terminal using you keyboard and choose `Accept & install`. Also, you can see the directory that VTune is going to be installed (`/opt/intel/oneapi`).

Next step you will see:

```
 [x] Skip Eclipse* IDE Configuration
 Next   Back   Quit
```

Do the check for option Skip Eclipse like `[x]`. Again you have to move here with you keyboard and then press `Next`.

Next step check the following option. I mean I don't think we would have any alternative option! Then press `Begin Installation`.

```
[x] I consent to the collection of my information
[ ] I do NOT consent to the collection of my information
Begin Installation  Back  Quit
```

At last, you will see the following information. You can just `Close` it.

```
Installation Complete | Intel VTune(TM) Profiler
______________________________________________________________


The following tools have been installed successfully:
Intel VTune(TM) Profiler
Installation location: /opt/intel/oneapi



Go to Installed Products  Close
Intel Software Installer Installed Products
```

4. Ensuring that VTune Profiler is installed and properly configured within the Windows Subsystem for Linux (WSL) environment involves checking the installation and verifying its functionality. You can simply type the `vtune` command line in the WSL terminal. Probably you will see an error that

```
Command not found.
```

If you're receiving a "command not found" error when attempting to run vtune within your WSL Ubuntu environment, it's likely that the VTune Profiler is not added to the system's PATH variable, preventing the shell from recognizing the command.

Setting Up PATH for VTune:

1. Identify the directory where VTune Profiler is installed. You can use:

   `sudo find / -name vtune`

   And you will see:

   ```
   /opt/intel/oneapi/vtune
   /opt/intel/oneapi/vtune/2024.0/etc/vtune
   /opt/intel/oneapi/vtune/2024.0/bin64/vtune
   and so on!
   ```

   If you go to the the following directory using WSL terminal:

   ```
   /opt/intel/oneapi/vtune/2024.0/bin64/vtune
   ```

and type the following command line:

```
ls -l vtune
```

you should be able to see an executable vtune file.

2. **Add to PATH (Temporary):** Temporarily add the VTune installation directory to the PATH in the current terminal session:

   ```
   export PATH=$PATH:/path/to/vtune/installation/directory
   ```

   Replace `/path/to/vtune/installation/directory` with the actual path where VTune is installed. In our case it is:

   ```
   export PATH=$PATH:/opt/intel/oneapi/vtune/2024.0/bin64
   ```

3. Verify PATH Update: To verify that the directory has been added to PATH, you can echo the PATH variable:

   ```
   echo $PATH
   ```

   This command will display the current value of the PATH variable including the newly added directory.

Warning! The changes made to PATH using export will only persist for the current terminal session. Once you close the terminal or start a new session, the changes will not be retained.

Add to PATH (Permanently):

To make the PATH changes permanent, you can edit your `.bashrc` file and add the export command there.

(a) Here's how you can access and edit the `.bashrc` file. Open Terminal in Ubuntu (WSL). Navigate to Home Directory using `cd ~`. Check if the `.bashrc` file exists using the `ls -a` command. The `-a` flag shows all files, including hidden files like `.bashrc`.

(b) Use a text editor like `nano` to edit your `.bashrc` file using `nano ~/.bashrc`

(c) Add the following line at the end of the file:

```
export PATH=$PATH:/opt/intel/oneapi/vtune/2024.0/bin64
```

Then, Save and Exit.

(d) To apply the changes without restarting the terminal, run:

```
source ~/.bashrc
```

Now finally we have VTune on WSL! If you run `vtune` command line on WSL terminal you should be able to see a list of different options.

Simply I can run VTune for the `Matrix_Multiplication.c`. The same way we did previously, first you have to compile and make the executable object file.

`gcc -o Matrix_Multiplication Matrix_Multiplication.c`

Now we have the object file called `Matrix_Multiplication`.

On the WSL terminal I can run:

`vtune -collect hotspots ./Matrix_Multiplication 2000`

where 2000 is the input size give by user. This is what I see in the other computer I have (that's why the results are slightly different):

```
vtune: Executing actions 75 % Generating a report       Elapsed Time: 79.897s

CPU Time: 79.890s
 Effective Time: 79.890s
 Spin Time: 0s
 Overhead Time: 0s
 Total Thread Count: 1
 Paused Time: 0s

 Top Hotspots

 Function               Module                 CPU Time   % of CPU Time(%)
 ---------------------  ---------------------  ---------  ----------------
 matrixMultiplication   Matrix_Multiplication    79.850s             99.9%
 main                   Matrix_Multiplication     0.030s              0.0%
 __GI___libc_malloc     libc.so.6                 0.010s              0.0%
```

5.4.3 Code Coverage

`gcov` is a profiling tool used in the GNU Compiler Collection (GCC) suite for code coverage analysis. It provides information about how much of your code is executed during testing. When compiled with the –coverage flag in GCC, the compiler adds instrumentation to the code, enabling `gcov` to gather data about which lines of code were executed and how many times.

Once you compile your code with the –coverage flag and run your program, it generates `.gcda` (coverage data) and `.gcno` (coverage notes) files. These files contain coverage data that gcov uses to generate human-readable reports detailing the coverage statistics of your code.

`gcov` provides detailed information about:

- Which lines of code were executed.
- How many times each line was executed.
- Percentage of code coverage for functions, lines, and branches.
- Unexecuted lines of code or functions that were never called during testing.

This information is valuable for identifying untested or poorly tested sections of code, ensuring comprehensive test coverage, and improving the quality of your software.

to use `gcov`, you'll need to have the GNU Compiler Collection (GCC) installed on your system, as `gcov` is a part of the GCC suite. In most cases, when you install GCC, `gcov` should be included by default.

To check if `gcov` is available on your system, you can open a terminal and type `gcov --version`.

If `gcov` is installed, it will display the version information. If it's not installed, you may need to install GCC or specifically look for a package that includes gcov. The method to install GCC and related tools can vary depending on your operating system. For instance, on Debian-based systems like Ubuntu, you can install GCC and related tools using the following command:

```
sudo apt-get install build-essential
```

This command installs essential development tools, including GCC and `gcov`, on your system. The exact package names and installation method might differ based on your specific operating system and package manager.

To use `gcov`, follow these steps:

1. **Compile with Coverage Flags**: Compile your C/C++ code with the `-fprofile-arcs` and `-ftest-coverage` flags to enable code coverage instrumentation. For example:

   ```
   gcc -fprofile-arcs -ftest-coverage -o my_program my_code.c
   ```

2. **Run Your Program**: Execute your program (`./my_program`). Ensure that all the code paths are covered during the execution. This step is necessary to collect coverage information.

3. **Generate Coverage Data**: After running your program, `gcov` generates coverage data files. Execute `gcov` on your source file to generate these files. For example:

   ```
   gcov my_code.c
   ```

   This command generates a `.gcov` file for each source file, containing information about code coverage for each line of code.

4. **View the Coverage Report**: Open the `.gcov` file generated by `gcov` in a text editor or use the less command to view the coverage information:

   ```
   less my_code.c.gcov
   ```

   This file provides details about the number of times each line was executed during program execution.

Using `gcov` helps you identify which parts of your code have been executed during testing, enabling you to analyze code coverage and assess the effectiveness of your tests.

Warning! In the **Step 3** after using `gcov my_code.c` you might see the following error:

```
my_program.gcno:version 'B23*', prefer 'B14*'
Segmentation fault (core dumped)
```

Or something like:

```
my_code.gcno:cannot open notes file
my_code.gcda:cannot open data file, assuming not executed
No executable lines
```

This might happen if the `gcov` tool version doesn't match the version used when compiling the code, leading to compatibility issues. The `.gcno` file is generated during the compilation and contains information necessary for `gcov` to produce coverage reports.
To resolve this issue, ensure that the `gcov` version you're using aligns with the version of the compiler used to generate the `.gcno` file. If possible, try to use the same version of `gcov` as the one used for compilation.
The second solution is to use another version of GCC. To use a different version of GCC on your system, you need to have multiple versions installed. You can see all the installed versions of GCC on your Linux system by using the following command in the terminal:
`ls /usr/bin/gcc*`
Here are the versions of GCC installed on my OS:

```
/usr/bin/gcc     /usr/bin/gcc-ar     /usr/bin/gcc-nm     /usr/bin/gcc-ranlib
/usr/bin/gcc-11  /usr/bin/gcc-ar-11  /usr/bin/gcc-nm-11  /usr/bin/gcc-ranlib-11
/usr/bin/gcc-12  /usr/bin/gcc-ar-12  /usr/bin/gcc-nm-12  /usr/bin/gcc-ranlib-12
/usr/bin/gcc-9   /usr/bin/gcc-ar-9   /usr/bin/gcc-nm-9   /usr/bin/gcc-ranlib-9
```

In this case, to solve the version problem I compile my source code with another version like:
`gcc-9 -fprofile-arcs -ftest-coverage -o my_program my_code.c`
Additionally, it might help to clean the directory of any old `.gcno`, `.gcda`, and `.gcov` files before rerunning the process. You can remove these files with:
`rm *.gcno *.gcda *.gcov`
Then, recompile your code with coverage flags and rerun the program to generate fresh coverage data. If the issue persists, you may need to check for version mismatches between your compiler and `gcov`.

Consider a C file named `my_code.c`:

```
#include <stdio.h>

int main() {
 int i, sum = 0;
 for (i = 1; i <= 10; ++i) {
  sum += i;
 }
 printf("Sum: %d\n", sum);
 return 0;
}
```

Follow the steps I mentioned above. This is the result I see in the terminal after `gcov my_code.c`:

```
File 'my_code.c'
Lines executed:100.00% of 6
Creating 'my_code.c.gcov'

Lines executed:100.00% of 6
```

In `my_code.c.gcov` file generated by I have:

```
   -:    0:Source:pedram0.c
   -:    0:Graph:pedram0.gcno
   -:    0:Data:pedram0.gcda
   -:    0:Runs:1
   -:    1:#include <stdio.h>
   -:    2:
   1:    3:int main() {
   1:    4:    int i, sum = 0;
 11:    5:    for (i = 1; i <= 10; ++i) {
 10:    6:        sum += i;
   -:    7:    }
   1:    8:    printf("Sum: %d\n", sum);
   1:    9:    return 0;
   -:   10:}
```

The information displayed by `gcov` after running the code provides details about the coverage analysis of the source code:

Source: Indicates the name of the source file being analyzed (in this case, `my_code.c`).

Graph: Refers to the associated `.gcno` file, which contains information for the program's control flow graph.

Data: Refers to the associated `.gcda` file, which contains coverage data gathered from the execution of the program.

Runs: Indicates the number of times the program has been executed, in this case, once.

Code Lines: Displays line-by-line coverage details for the source code:

- The hyphen - indicates lines that don't contain executable code.
- Line numbers along with the number of times each line was executed.

The specific coverage information illustrates which lines of the code were executed during the program run. In this case:

- Line 3 (`int main() {}`) was executed only one time.
- Lines 5 and 6 (within the `for` loop) were executed multiple times (as indicated by the execution counts).
- Line 8 (`printf("Sum:  %d\n", sum);`) was executed once.
- Line 9 (`return 0;`) was executed once.

This output provides an overview of which lines in the source code were executed during the program's run, aiding in understanding the code's coverage and potentially identifying areas that were not executed during testing.

5.5 GitHub and Version Control

Git is a distributed version control system (DVCS) designed to handle everything from small to very large projects with speed and efficiency. It was created by Linus Torvalds in 2005 to manage the development of the Linux kernel.

GitHub is a web-based platform built on top of Git, providing additional features and services to enhance collaboration and project management.

Overall, Git and GitHub have revolutionized the way software is developed by providing powerful tools for version control, collaboration, and project management. Whether you're working on a personal project or collaborating with a team, mastering Git and GitHub can significantly improve your workflow and productivity.

To check if Git is installed on your system, you can run `git --version` command in your terminal. If Git is installed, this command will display the version of Git that is installed on your system. If Git is not installed, you will likely see an error message indicating that the git command is not found.

To install Git on Unix-based systems such as Linux or macOS, you can use the package manager specific to your distribution. On **Debian/Ubuntu-based** systems, you can use `apt install git`. On **macOS**, you can install Git using Homebrew `brew install git`.

Here are some **basic** examples of using **Git** on a Linux machine:

Initializing a Repository: To start using Git in a directory, you need to initialize a Git repository. From terminal go to the directory you want to initialize like `cd /path/to/your/project`. Mine is `~/git_practice`. Once you're in the directory, use `ls -a` command to list all files and directories, including hidden ones. If a `.git` exists, it means the folder is initialized as a Git repository. Switch to another folder that you don't need it and git is not initialized, or remove the git in the directory by `rm -rf .git`. Then use `git init` to initialize Git:

```
hint: Using ’master’ as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:  git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of ’master’ are ’main’, ’trunk’ and
hint: ’development’. The just-created branch can be renamed via this command:
hint:
hint:  git branch -m <name>
Initialized empty Git repository in /home/pedram/git_practice/.git/
```

Checking Status: You can check the status of your repository to see which files have been modified, added, or deleted using `git status`. The following output in terminal says on branch `master`, there are no **commits** and few untracked files.

```
On branch master

No commits yet

Untracked files:
(use "git add <file>..." to include in what will be committed)
.vscode/
```

```
Hello.c
Makefile
a.out
factorial.c
functions.c
functions.h
functions.o
main
main.c
main.o
main1.c
make/
pedram.c
pedram2.c

nothing added to commit but untracked files present (use "git add" to track)
```

Adding Files: to the **staging** area **before committing** them by `git add <file1> <file2>`, or `git add .` command to add **all** the contents of the current directory. Then, use `git status` again, and you should be able to see:

```
On branch master

No commits yet

Changes to be committed:
(use "git rm --cached <file>..." to unstage)
new file:   .vscode/settings.json
new file:   Hello.c
new file:   Makefile
new file:   a.out
new file:   factorial.c
new file:   functions.c
new file:   functions.h
new file:   functions.o
new file:   main
new file:   main.c
new file:   maingit remote add origin https://github.com/pedrampa/git_practice.git.o
new file:   main1.c
new file:   make/Makefile
new file:   pedram.c
new file:   pedram2.c
```

Committing Changes: Commit your changes to the repository along with a descriptive message, like `git commit -m "Your descriptive message here"`, and we get:

```
[master (root-commit) 72a1d0e] Initial commit
15 files changed, 420 insertions(+)
<and a bunch of other information!>
```

Then use `git status` again:

```
On branch master
nothing to commit, working tree clean
```

Viewing Commit History: View a log of all commits made in the repository by `git log`:

```
Author: pedrampa <pedram.pasandide.edu1@gmail.com>
Date:   Fri Mar 22 15:44:05 2024 -0400

Initial commit
```

Let’s play with it. While I am in the same directory that I `git init`, I use `nano new_file.c` to create `new_file.c` as a new file. Then, I do `git status`, and I’ll see:

```
On branch master
Untracked files:
(use "git add <file>..." to include in what will be committed)
new_file.c

nothing added to commit but untracked files present (use "git add" to track)
```

Using `cat new_file.c`, I can see the contents inside the file (the stuff I wrote):

```
// I set fire to the rainnnn (a message from me!)
```

I can use `git commit -m "didn’t work (a message from me)"` to save the changes. Let’s open the file `nano new_file.c`, and remove the previous content and paste the following code:

```c
// an update within the file.
// the previous comment was removed

#include <stdio.h>

int main(){printf("I can set fire to the rain!\n");}
```

Compile the code with `gcc new_file.c`, so you can run the code with `./a.out`. Now use `git status`:

```
On branch master
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
modified:   a.out
modified:   new_file.c

no changes added to commit (use "git add" and/or "git commit -a")
```

I can use `git diff new_file.c` to see the changes line by line:

```
diff --git a/new_file.c b/new_file.c
index 7d8df95..bbff458 100644
--- a/new_file.c
+++ b/new_file.c
@@ -1 +1,6 @@
-// I set fire to the rainnnn (a message from me!)
+// an update within the file.
+// the previous comment was removed
+
+#include <stdio.h>
+
+int main(){printf("I can set fire to the rain!\n");}
```

Let's check the branches we have with `git branch`:

```
* master
```

Instead of `* master`, you might see `* main`. `git branch <branch_name>` creates a new branch. Let's say we create two new branches named `Pedram` and `Yiqi`. Then, `git branch`:

```
Pedram
Yiqi
* master
```

If I switch the branch using `git checkout Pedram`, I get:

```
error: Your local changes to the following files would be overwritten by checkout:
new_file.c
Please commit your changes or stash them before you switch branches.
Aborting
```

Above message say there some changes that are not committed yet. I add the files using `git add new_file.c a.out` and commit the changes with `git commit -m "the code"`. Now the command `git checkout Pedram` gives me:

```
Switched to branch 'Pedram'
```

Now if I use `ls` in the terminal I will see the same file were in `master` branch. I can see the same files in the window open showing the contents of the directory. **The same directory but files can be different**. Let's try. To **remove all the files in the current branch** you can use `git rm -r .`. If we use `ls` or go to the window (`open .`), we will see nothing is there. We haven't commit any changes yet (double check with `git status`).

The command `git checkout .` is typically used to discard changes in the working directory for tracked files. However, since you have already staged the removal of files using `git rm -r .`, those changes are already staged, not present in your working directory.

To restore the files that you have staged for removal (i.e., to unstage the removal), you can use `git restore --staged .`. This command will unstage the removal of files, effectively bringing them back to the staging area. After that, you can use `git checkout .` to discard the changes in the working directory for any tracked files that were modified:

I remove all the files again, but this, time I will `git commit -m "remove all the files"`. Now I want to go back to a specific previous version right before the one I removed all the files. Use `git log` to view the commit history and identify the commit hash of the last commit before the removal:

```
commit bd27fa4c9e6fa267a9493f9411cdbb65153057a1 (HEAD -> Pedram)
Author: pedrampa <pedram.pasandide.edu1@gmail.com>
Date:   Fri Mar 22 19:05:31 2024 -0400

remove all the files

commit 073c5c5a6cc0ba373a1002e6dbb81c0f1d381106 (master, Yiqi)
Author: pedrampa <pedram.pasandide.edu1@gmail.com>
Date:   Fri Mar 22 18:32:28 2024 -0400

the code
```

If after `git log`, the terminal was stuck in showing the logs, just press the key `Q` on your keyboard. Note the commit hash of the commit just before the `remove all the files` commit. I want to go back to the version with comment `the code` at time `18:32:28`. To reset the branch to a specific previous commit version, we can use `git reset --hard <commit_hash>`. In my case, I have to use `git reset --hard 073c5c5a6cc0ba373a1002e6dbb81c0f1d381106`.

After resetting the branch, the files will be restored to their state at the specified commit. You can now use `git checkout -- .` to bring back the files from the previous commit. Use `ls` to see the files. Again, I remove all the files except `new_file.c` and `a.out` by the command shell:

```
git ls-files | grep -vE ’^(new_file.c|a.out)$’ | xargs git rm
```

But this time, I commit the changes for real for real by `git commit -m "Remove 2"`. I change `nano new_file.c` with the contents `cat new_file.c`:

```c
#include <stdio.h>

int main(){printf("Helloooo from Pedram branch!\n");}
```

Compile the code and now `a.out` is updated too. Then, execute:

1. `git add .`, and
2. `git commit -m "Pedram branch latest version"`.

Check the files within the directory. Using `git checkout master`, we can switch back to `master` branch, and we have totally different files (you can check it with `ls` in the terminal or in the window with `open .`).

Merging Branches: Merge changes from one branch into another using:

1. `git checkout <branch_to_merge_into>`
2. `git merge <branch_to_merge_from>`

For example, using `git checkout Yiqi`, I switch the branch to `Yiqi`. Here I have all the same files which I have in `master` branch, since `Yiqi` was create as a copy from `master`. Check the current branch using `git branch`:

```
Pedram
* Yiqi
master
```

While I am in `Yiqi`, I use `git merge Pedram` to merge everything from `Pedram` into `Yiqi`. Now check the files in the current directory within the branch `Yiqi`. You'll see there are only the files we had (better to say we still have) in `Pedram`.

There is another way to revert the changes that you might see other people do. Mind you that controlling the versions is the most important feature of git. I made some changes in the `Yiqi` branch by updating `new_file.c` to:

```c
// this is a comment from Yiqi, Git is really useful. Don't run away!
#include <stdio.h>

int main(){printf("Helloooo from Pedram branch!\n");}
```

Use `git add .`, then `git commit -m "Updates from Yiqi"`. I can use `git log --oneline` to see a summary of updates:

```
627361f (HEAD -> Yiqi) Updates from Yiqi
1b3cd94 (Pedram) Pedram branch latest version
c67fd32 Remove 2
073c5c5 (master) the code
2f5622d didn't work (a message from me)
72a1d0e Initial commit
```

The first columns are the same hash (first 7 characters). The latest update was made from current branch, `Yiqi`, with the comment ”Updates from Yiqi”, and a hash of `627361f`. I want to go back to the previous version, I just need to revert the latest change with hash `627361f`. I can use `git revert 627361f` giving:

```
Revert "Updates from Yiqi"

This reverts commit 627361fec114e6a595ebfb8fae94924b6c1f7785.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch Yiqi
# Changes to be committed:
#       modified:   new_file.c
#
```

Save the file and exit. Just to double check I can execute the command `cat new_file.c`, and it will show me the following output, which is the initial version merged from `Pedram` branch.

```
#include <stdio.h>

int main(){printf("Helloooo from Pedram branch!\n");}
```

Let's move forward to **GitHub**. Now I want to make a copy of what I have on a server like GitHub. Create an account, then a new repository, named `git_practice` (In my machine the name of folder is also the same). The first page will give us some information about **Quick setup - if you've done this kind of thing before** and a HTTPS link, which in my case it is:

`https://github.com/pedrampa/git_practice.git`

Go to the terminal and `cd /path/to/your/project` where we initialized `git`. In my machine it was `˜/git_practice`. Then, add the GitHub repository as a remote:

`git remote add origin https://github.com/pedrampa/git_practice.git`

To push all branches from your local repository to the GitHub repository, including `Pedram`, `Yiqi`, and `master`, use the `--all` flag with the `git push --all origin`. This command will ask you to enter the username and password you have for your Github account. But this will give you the following error:

```
Username for 'https://github.com': pedrampa
Password for 'https://pedrampa@github.com':
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see <here it shows you a link>
fatal: Authentication failed for 'https://github.com/pedrampa/git_practice.git/'
```

The error message indicates that the authentication with GitHub failed. This typically happens when you're using a username and password for authentication, but GitHub no longer supports password authentication for Git operations over HTTPS.

To resolve this, you can use one of the following methods for authentication:

1. SSH Key Authentication: Set up SSH key authentication between your local machine and GitHub. This involves generating an SSH key pair, adding the public key to your GitHub account, and configuring Git to use SSH for authentication. We will not cover it here. But it is really useful. For more information take a look at Connecting to GitHub with SSH.

2. Personal Access Token (PAT):

Generate a Personal Access Token on GitHub and use it as a replacement for your GitHub password. Here's how you can do it:

- Go to your GitHub account settings.
- Click on ”Developer settings” > ”Personal access tokens” > ”Generate new token”.
- Give your token a descriptive name, select the scopes (permissions) you need, and click ”Generate token”.
- Copy the generated token.
- When prompted for your password during the git push operation, use your GitHub username as the username and the generated token as the password.

Using the token as the password I get:

```
Enumerating objects: 36, done.
Counting objects: 100% (36/36), done.
Delta compression using up to 8 threads
Compressing objects: 100% (32/32), done.
Writing objects: 100% (36/36), 15.37 KiB | 3.07 MiB/s, done.
Total 36 (delta 6), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (6/6), done.
To https://github.com/pedrampa/git_practice.git
* [new branch]      Pedram -> Pedram
* [new branch]      Yiqi -> Yiqi
* [new branch]      master -> master
```

Now I can see the same files on GitHub under the repository named `git_practice` with 3 different branches that I can switch between them.

Let’s remove all the files from your local machine (not GitHub). Technically, the whole folder `git_practice` from my local machine. Let’s say it happened accidentally, and now we have a server online which has the backup. We want to get the files from Github. Go to the repository in the GitHub account, click on button called `<> Code` ,and find ”HTTPS” address under the section ”clone”. In my GitHub account, the HTTPS is:

`https://github.com/pedrampa/git_practice.git`

Then, in the terminal with `pwd` where I want to have the folder `git_practice`, I execute:

`git clone https://github.com/pedrampa/git_practice.git`

This will give me:

```
Cloning into 'git_practice'...
remote: Enumerating objects: 36, done.
remote: Counting objects: 100% (36/36), done.
remote: Compressing objects: 100% (26/26), done.
remote: Total 36 (delta 6), reused 36 (delta 6), pack-reused 0
Receiving objects: 100% (36/36), 15.37 KiB | 1.92 MiB/s, done.
```

Use `cd git_practice/`, then `git branch`, I can only see:

```
* Pedram
```

But, when you I use `git branch -a`, I have:

```
* Pedram
remotes/origin/HEAD -> origin/Pedram
remotes/origin/Pedram
remotes/origin/Yiqi
remotes/origin/master
```

Use `git checkout Yiqi` and `git checkout master` to have a copy of them also **on your local machine**. The branch `Pedram` was cloned first since it is the **default branch** on GitHub. Let's remove `git_practice` from your local machine. Then go back to the `Personal access tokens` setting on your GitHub account and click on the button ”Revoke all” or ”Delete” the specific token we initially create. Deleting this token means the password to sign into the GitHub and clone the files is not gonna work anymore. Go back to the terminal and try do download the files again by `git clone https://github.com/pedrampa/git_practice.git`. If **you do it right away**, then, you see in the terminal:

```
Cloning into 'git_practice'...
remote: Invalid username or password.
fatal: Authentication failed for 'https://github.com/pedrampa/git_practice.git/'
```

Which drives me crazy, because there is not explanation. But if we wait a little bit and then doing it again, we'll see:

```
git clone https://github.com/pedrampa/git_practice.git
Cloning into 'git_practice'...
Username for 'https://github.com': pedrampa
Password for 'https://pedrampa@github.com':
```

Here after I enter my username and password (I have to generate **a new token** for the password), I get the cloned files.

NOTE! Let's stop here. We can keep going on GitHub and there is a lot to mention. There many more important and useful command and features on `git`. I think when you start a long term project with your friends or coworkers, you'll get your hands on it. Here I just tried to show you an example of a project that potentially three people, including `Pedram`, `Yiqi`, and `master`, work on developing or supervising the progresses. Once, I remember, in one of the projects I was working on with my coworker Yiqi and our supervisor, I messed some pushes and we lost a couple weeks of progress. It happens! Anyway, don't forget that here we just touched the surface.

5.6 Documentation

Documentation is essential for software development because it serves as a comprehensive reference guide that enables developers, stakeholders, and users to understand, utilize, and maintain the software effectively. It provides insights into the design, functionality, usage, and implementation details of the codebase, facilitating collaboration among team members, easing the onboarding process for new developers, and ensuring consistency and accuracy in development practices. Documentation also acts as a roadmap for troubleshooting, debugging, and extending the software, enabling faster resolution of issues and smoother integration of new features. Ultimately, well-maintained documentation enhances the overall quality, reliability, and longevity of software projects, contributing to their success in meeting the needs of users and stakeholders

We have many documentation generators, and determining which one is ”better” depends on various factors such as the specific requirements of your project, the programming languages you're using, your team's preferences, and the features you prioritize. Each documentation generator has its own strengths and weaknesses.

One of a widely used documentation generator tool for software projects is Doxygen. It is designed to extract documentation from source code comments, typically written in languages like C, C++, Java, Python, and others, and generate various forms of documentation from them. These forms can include HTML, PDF, LaTeX, RTF, and more.

Developers use Doxygen to document their code by adding special comment blocks to their source code. These comments are then processed by Doxygen to generate documentation in a structured and easily navigable format. This documentation can include information such as class and function descriptions, parameters, return values, usage examples, and more.

Doxygen also supports features such as cross-referencing between different parts of the documentation, inheritance diagrams, collaboration diagrams, and dependency graphs, which can help users understand the structure and relationships within a codebase.

Overall, Doxygen helps developers maintain well-documented and easily understandable codebases, which is crucial for collaboration, maintenance, and future development of software projects.

Note! I was thinking how to write this section and I found these two videos on YouTube:

- How to install Doxygen on Unix?
- Doxygen from installation to a case study.

Check the results in the second video and see how much it is interesting and professional to make a document like that. The most difficult part for you might be the installation. I suggest even if you don't want to use just practice the installation on Unix and ask your TAs to help you out if there is any issue.

Anyway, that would be pointless if I wanted to write the same stuff here. I guess they are explaining the details well enough to start with documentation.

6 Basics of Data Structures

A data structure is a way to organize, manage, and store data so that it can be accessed and modified efficiently. Data structures are fundamental in computer science and provide the foundation for building efficient algorithms. They allow developers to handle large amounts of data in a systematic way, enhancing performance in terms of time and space. Common operations like searching, insertion, deletion, and sorting are optimized using various data structures. Choosing the right data structure can greatly improve the efficiency of a program. Some important types of data structures include arrays, linked lists, stacks, queues, trees, heaps, and graphs.

6.1 Hashing and HashMap

Hashing is a technique used to map data of arbitrary size to data of a fixed size. In simple terms, hashing converts an input (or key) into an integer, known as the **hash code**, which is used as an index in an array (known as a hash table) to store the corresponding value. The main advantage of hashing is that it allows for fast data retrieval. Ideally, it provides constant time complexity, $\mathbf{O(1)}$, for lookups, insertions, and deletions.

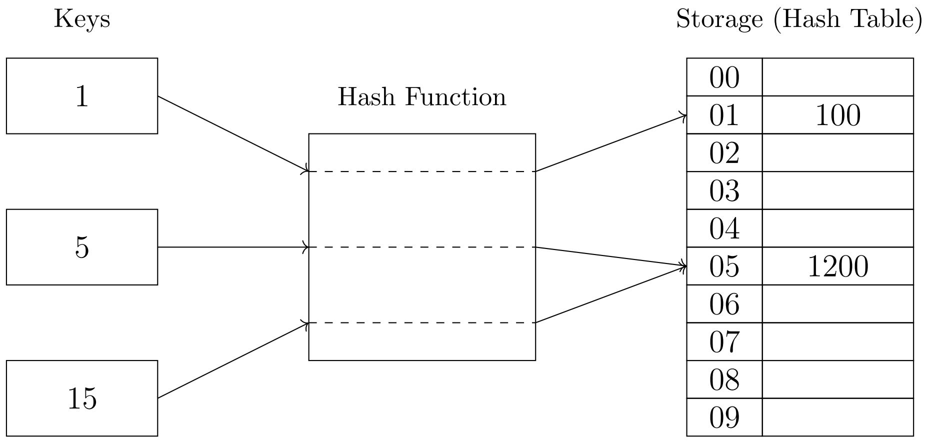
A **HashMap** is a data structure that implements an associative array, or map, where key-value pairs are stored. Each key is unique, and values can be accessed by their keys efficiently using hashing. Some applications of Hashing and HashMap:

- Databases: Used for indexing to quickly retrieve data based on a key.
- Caching: Hashing is used to store results of expensive computations and retrieve them quickly.
- Password Management: Hash functions are used to store password hashes instead of plain text passwords.
- Symbol Tables in Compilers: Used to store variable names and their associated data in programming languages.

Figure 7 is an example of a basic hash table without collision handling. In this case, we assume that the keys are unique, so we don't need to handle any collisions.

We can program the above hash table as:

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



00	
01	100
02	
03	
04	
05	1200
06	
07	
08	
09	

Figure 7: Visualization of Hash Function and Storage in Hash Table with Collision (200 was replaced by 1200)

```c
#define TABLE_SIZE 10

typedef struct {
 int key;
 int value;
} HashEntry;

HashEntry* hashTable[TABLE_SIZE];

// Simple hash function
int hashFunction(int key) {
 return key % TABLE_SIZE;
}

void insert(int key, int value) {
 int index = hashFunction(key);
 HashEntry *entry = (HashEntry*)malloc(sizeof(HashEntry));
 entry->key = key;
 entry->value = value;
 hashTable[index] = entry;
}

int search(int key) {
 int index = hashFunction(key);
 if (hashTable[index] != NULL && hashTable[index]->key == key) {
  return hashTable[index]->value;
 }
 return -1;  // Key not found
}

int main() {
 insert(1, 100);
 insert(5, 200);
 insert(15, 1200);  // Collision would occur here without handling

 printf("Value for key 1: %d\n", search(1));
 printf("Value for key 5: %d\n", search(2));
 printf("Value for key 15: %d\n", search(15));

 return 0;
}
```

The output for the above code is:

```
Value for key 1: 100
Value for key 5: -1
Value for key 15: 1200
```

In real-world scenarios, hash collisions are common, so we need to handle them. One common technique is **separate chaining**, where each array index contains a linked list of entries that hash to the same index (Figure 8).

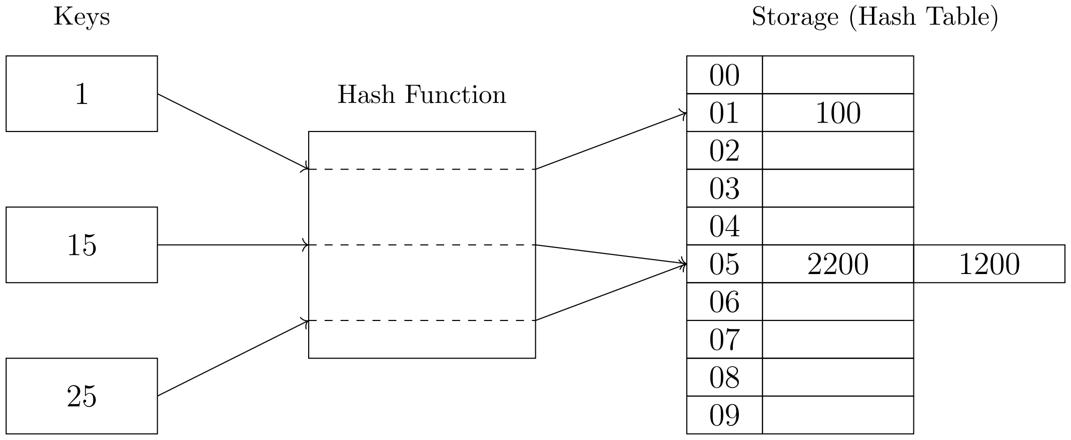


Figure 8: Handling Collision Using Linked Lists

We can program the above procedure as:

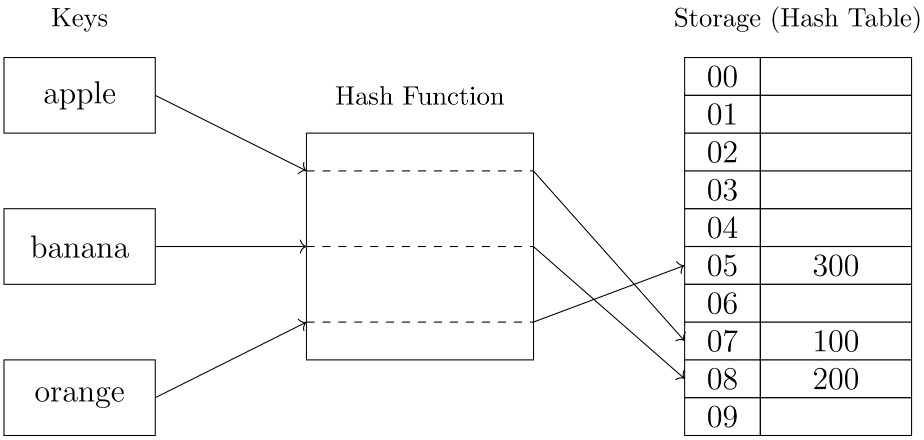
```c
#include <stdio.h>
#include <stdlib.h>
```

```c
#define TABLE_SIZE 10

typedef struct HashEntry {
 int key;
 int value;
 struct HashEntry* next;
} HashEntry;

HashEntry* hashTable[TABLE_SIZE];

// Simple hash function
int hashFunction(int key) {
 return key % TABLE_SIZE;
}

void insert(int key, int value) {
 int index = hashFunction(key);
 HashEntry *newEntry = (HashEntry*)malloc(sizeof(HashEntry));
 newEntry->key = key;
 newEntry->value = value;
 newEntry->next = NULL;

 // Insert at the beginning of the list for simplicity
 if (hashTable[index] == NULL) {
  hashTable[index] = newEntry;
 } else {
  newEntry->next = hashTable[index];
  hashTable[index] = newEntry;
 }
}

int search(int key) {
 int index = hashFunction(key);
 HashEntry* entry = hashTable[index];
 while (entry != NULL) {
  if (entry->key == key) {
   return entry->value;
  }
  entry = entry->next;
 }
 return -1;  // Key not found
}
```

```
int main() {
 insert(1, 100);
 insert(15, 1200);  // Collision
 insert(25, 2200);  // Collision

 printf("Value for key 15: %d\n", search(15));
 printf("Value for key 25: %d\n", search(25));

 return 0;
}
```

In this example, keys 15 and 25 hash to the same index, but collisions are handled by maintaining a linked list at that index.

Hashing is often used with string keys (Figure 9), especially in dictionary-like structures. A more advanced example is using strings as keys in a hash map. We need a **string hashing function** like the **djb2** hash function.



00	
01	
02	
03	
04	
05	300
06	
07	100
08	200
09	

Figure 9: Hash Function and Storage in Hash Table in Strings

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

#define TABLE_SIZE 10

typedef struct HashEntry {
 char* key;
 int value;
 struct HashEntry* next;
} HashEntry;
```

```
HashEntry* hashTable[TABLE_SIZE];

// Hash function for strings (djb2 algorithm)
unsigned long hashFunction(char* str) {
 unsigned long hash = 5381;
 int c;
 while ((c = *str++)) {
  hash = ((hash << 5) + hash) + c;
 }
 return hash % TABLE_SIZE;
}

void insert(char* key, int value) {
 unsigned long index = hashFunction(key);
 HashEntry* newEntry = (HashEntry*)malloc(sizeof(HashEntry));
 newEntry->key = strdup(key);  // Duplicate string
 newEntry->value = value;
 newEntry->next = hashTable[index]; // collision is handled
 hashTable[index] = newEntry;
}

int search(char* key) {
 unsigned long index = hashFunction(key);
 HashEntry* entry = hashTable[index];
 while (entry != NULL) {
  if (strcmp(entry->key, key) == 0) {
   return entry->value;
  }
  entry = entry->next;
 }
 return -1;  // Key not found
}

int main() {
 insert("apple", 100);
 insert("banana", 200);
 insert("orange", 300);

 printf("Value for 'apple': %d\n", search("apple"));
 printf("Value for 'banana': %d\n", search("banana"));
 printf("Value for 'cherry': %d\n", search("cherry"));
```

```
  return 0;
}
```

In this example, the **djb2** hash function processes strings, and the hash map stores key-value pairs where the keys are strings. Collisions are handled using linked lists, as before.

Additional Considerations:

- Load Factor: The performance of a hash table is dependent on the load factor, which is the ratio of the number of entries to the table size. High load factors can lead to more collisions and performance degradation.
- Open Addressing: Another method to resolve collisions is open addressing, where instead of using linked lists, we probe the hash table for the next available slot.

6.2 Linked List

A Linked List is a linear data structure where elements are stored in nodes. Each node contains two parts:

1. **Data**: The actual value or information stored in the node.
2. **Pointer (next)**: A reference (or pointer) to the next node in the list.

Unlike arrays, linked lists are not stored in contiguous memory locations. Instead, each element points to the next, allowing for dynamic memory allocation. This flexibility makes linked lists useful for scenarios where you do not know the size of the dataset in advance or need frequent insertions and deletions.

Linked lists are fundamental in many computer science applications, and they serve as the foundation for more complex data structures like stacks, queues, and graphs.

Types of Linked Lists are (Figure 10):

- Singly Linked List: Each node contains a single pointer pointing to the next node.
- Doubly Linked List: Each node contains two pointers, one pointing to the next node and another pointing to the previous node.
- Circular Linked List: The last node points back to the first node, forming a circular chain.

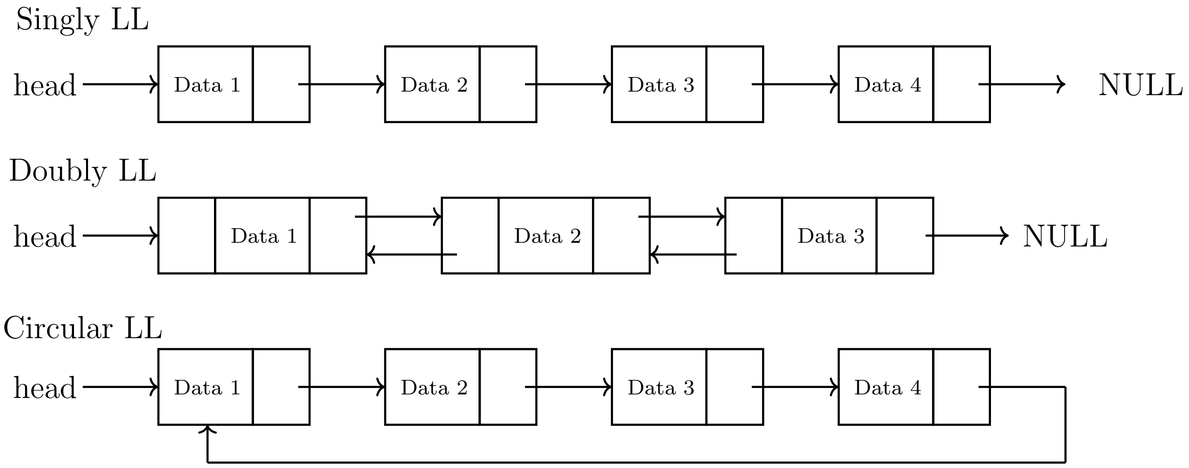


Figure 10: Linked List (LL) Data Structure

Advantages of Linked Lists:

- Dynamic Size: Linked lists can grow and shrink as needed.
- Efficient Insertions/Deletions: Inserting or deleting an element at any position (especially in the middle) is faster compared to arrays, as no shifting is required.

Disadvantages of Linked Lists:

- Random Access: Linked lists don't allow random access. To access a node, you must traverse the list from the beginning.
- Memory Overhead: Each node requires extra memory for the pointer to the next node.

Here is a basic example of **Singly** Linked List Implementation in C:

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

// Node structure
typedef struct Node {
 int data;
 struct Node* next;
}Node;

// Function to create a new node
Node* createNode(int data) {
 struct Node* newNode = (Node*)malloc(sizeof(Node));
 newNode->data = data;
 newNode->next = NULL;
 return newNode;
}

// Function to print the linked list
void printList(Node* head) {
 Node* temp = head;
 while (temp != NULL) {
  printf("%d -> ", temp->data);
  temp = temp->next;
 }
 printf("NULL\n");
}

// Function to insert a new node at the end
void appendNode(Node** head, int data) {
 Node* newNode = createNode(data);
 if (*head == NULL) {
  *head = newNode;
  return;
 }
 Node* temp = *head;
 while (temp->next != NULL) {
  temp = temp->next;
 }
 temp->next = newNode;
}

// Main function
int main() {
 Node* head = NULL;

 // Inserting nodes
 appendNode(&head, 10);
 appendNode(&head, 20);
 appendNode(&head, 30);
 // Printing the list
 printList(head);

 return 0;
}
```

This code defines a singly linked list. It demonstrates how to create a new node and append it to the end of the list. The `printList` function traverses the list from the head to the last node:

```
10 -> 20 -> 30 -> NULL
```

Sometimes it is necessary to insert a `Node` at the beginning of list:

```
void insertAtBeginning(Node** head, int data) {
 Node* newNode = createNode(data);
 newNode->next = *head;
 *head = newNode;
}
```

Here, the new node is inserted at the beginning of the list. The new node's next pointer points to the current head, and the head is updated to the new node.

Reversing a singly linked list is a common operation where you reverse the direction of the pointers in the list:

```
Node* reverseList(Node* head) {
 Node* prev = NULL;
 Node* current = head;
 Node* next = NULL;

 while (current != NULL) {
  next = current->next;   // Store the next node
  current->next = prev;   // Reverse the current node's pointer
  prev = current;         // Move pointers one position ahead
  current = next;
 }
 return prev;
}
```

The above function has a time Complexity of $O(n)$, where n is the number of nodes, and a space complexity of $O(1)$, since we only use a few extra pointers. You can add the following code into the `main()` to test your function:

```
Node* ReversedHead = reverseList(head);
printList(ReversedHead);
```

This will give you:

```
 30 -> 20 -> 10 -> NULL
```

Merging two sorted linked lists into one sorted list is another common problem:

```
Node* mergeSortedLists(Node* l1, Node* l2) {
 if (!l1) return l2;
 if (!l2) return l1;

 if (l1->data < l2->data) {
  l1->next = mergeSortedLists(l1->next, l2);
  return l1;
 } else {
  l2->next = mergeSortedLists(l1, l2->next);
  return l2;
 }
}
```

The function recursively merges two sorted lists. If `l1`'s data is smaller, the function moves to the next node in `l1`, otherwise, it moves to the next node in `l2`. You can test the above function by creating two lists, let's say `head1` and `head2`, pass two lists to function, and saved the returned linked list into another list, say `MergedList`, then print the results with `printList`.

Another commonly used operation on linked list is removing Nth node from end of list:

```
Node* removeNthFromEnd(Node* head, int n) {
 Node* dummy = createNode(0);
 dummy->next = head;
 Node *first = dummy, *second = dummy;

 for (int i = 0; i <= n; i++) {
  first = first->next;
 }
 while (first != NULL) {
  first = first->next;
  second = second->next;
 }

 Node* toDelete = second->next;
 second->next = second->next->next;
 free(toDelete);

 return dummy->next;
}
```

This algorithm uses two pointers. The first pointer is moved n nodes ahead. Then both pointers move together until the first pointer reaches the end. This way, the second pointer ends up at the node right before the node to be removed. Test the above function by adding the following code into `main()`:

```
Node* RemovedNth = removeNthFromEnd(head,1);
printList(RemovedNth);
```

This will give you:

```
10 -> 20 -> NULL
```

Why is head changed even though you return a new pointer? Probably you have noticed after calling a function like `removeNthFromEnd()` the original `head` list is changed (the same as `RemovedNth`). This happens because C passes pointers by value, meaning the function receives a copy of the pointer `head`, but the copy still points to the same memory as the original. Any changes made to the list via this pointer will affect the original list, including when nodes are removed or restructured.

Linked lists are a fundamental data structure that offer great flexibility in managing dynamically sized data. Their ability to efficiently insert and delete elements, combined with their simplicity, makes them essential in many real-world applications. Mastering the different operations on linked lists (insertion, deletion, traversal, and searching) is crucial for understanding more complex data structures and algorithms. Obviously, there is not enough time to cover all those function in this course. Please feel free to reach out if you are looking for some online resource to practice. Meanwhile you can start with the following exercises:

- Implement a function that removes duplicates from a sorted linked list.

- Implement a function that checks if a linked list has a cycle.

These tasks build the foundational skills necessary to move on to more advanced data structures like doubly linked lists and circular linked lists.

6.3 Stack

A stack is a linear data structure that follows the **Last In**, **First Out** (**LIFO**) principle, meaning the last element added to the stack is the first one to be removed. Think of a stack like a pile of books: you can only remove or add a book to the top of the stack. This structure is widely used in computer science for tasks like function call management, expression evaluation, undo mechanisms in text editors, depth-first search in graphs, and more.

Key operations in a stack (Figure 11) are:

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes the element from the top of the stack.
- **Peek/Top**: Returns the top element without removing it.
- **isEmpty**: Checks if the stack is empty.

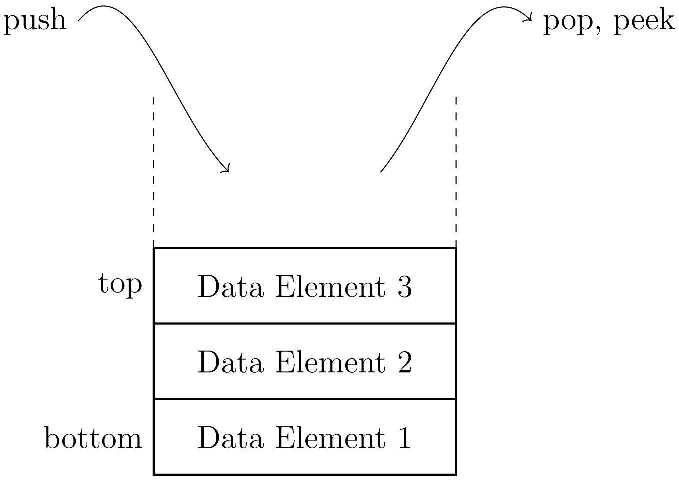


Figure 11: Stack Data Structure

Stacks are used in:

- Expression Evaluation: Converting and evaluating expressions like infix, postfix, or prefix.

- Recursion: Function calls are managed using a call stack.
- Undo Mechanisms: Maintaining a history of operations in applications like text editors.
- Backtracking Algorithms: Such as depth-first search (DFS) in graph traversal.
- Memory Management: Keeping track of memory during function calls (stack memory).

A stack can be implemented in two ways:

1. Array-based Implementation (Fixed size)
2. Linked List-based Implementation (Dynamic size)

We'll start with a basic array-based implementation and gradually move to more advanced topics. Here's a simple example of how to implement a **stack using arrays** in C:

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

#define MAX 5  // Define maximum size for the stack

typedef struct Stack {
 int arr[MAX];
 int top;
}Stack;

// Initialize stack
void initStack(Stack *stack) {
 stack->top = -1;
}

// Check if stack is full
int isFull(Stack *stack) {
 return stack->top == MAX - 1;
}

// Check if stack is empty
int isEmpty(Stack *stack) {
 return stack->top == -1;
}
```

```c
// Push an element onto the stack
void push(Stack *stack, int value) {
 if (isFull(stack)) {
  printf("Stack Overflow\n");
  return;
 }
 stack->arr[++stack->top] = value;
 printf("Pushed %d onto the stack\n", value);
}

// Pop an element from the stack
int pop(Stack *stack) {
 if (isEmpty(stack)) {
  printf("Stack Underflow\n");
  return -1;
 }
 return stack->arr[stack->top--];
}

// Peek at the top element
int peek(Stack *stack) {
 if (isEmpty(stack)) {
  printf("Stack is empty\n");
  return -1;
 }
 return stack->arr[stack->top];
}

int main() {
 Stack stack;
 initStack(&stack);

 push(&stack, 10);
 push(&stack, 20);
 push(&stack, 30);

 printf("Top element is %d\n", peek(&stack));
 printf("Popped element is %d\n", pop(&stack));
 printf("Popped element is %d\n", pop(&stack));

 return 0;
}
```

In this basic implementation:

- We define a stack with a maximum size using an array.
- The stack operations (push, pop, peek, isFull, isEmpty) are defined as functions that modify the stack.
- Overflow and underflow conditions are handled, ensuring that we can't add more elements than the defined maximum, and we can't remove elements from an empty stack.

A **linked list-based implementation of a stack** provides flexibility as it allows dynamic size. Each node in the stack is dynamically allocated memory, and we can easily push or pop elements without worrying about the predefined size limit.

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

typedef struct Node{
 int data;
 struct Node *next;
} Node;

// Check if the stack is empty
int isEmpty(Node *top)
{
 return top == NULL;
}

// Push an element onto the stack
void push(Node **top, int value)
{
 Node *newNode = (Node *)malloc(sizeof(Node));
 if (!newNode)
 {
  printf("Heap Overflow\n");
  exit(1);
 }
 newNode->data = value;
 newNode->next = *top;
 *top = newNode;
 printf("Pushed %d onto the stack\n", value);
}

// Pop an element from the stack
int pop(Node **top)
{
 if (isEmpty(*top))
 {
  printf("Stack Underflow\n");
  return -1;
 }
 Node *temp = *top;
 int poppedValue = temp->data;
 *top = (*top)->next;
 free(temp);
 return poppedValue;
}

// Peek at the top element
int peek(Node *top)
{
 if (isEmpty(top))
 {
  printf("Stack is empty\n");
  return -1;
 }
 return top->data;
}

int main()
{
 Node *stack = NULL;
```

```
    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);

    printf("Top element is %d\n", peek(stack));

    printf("Popped element is %d\n", pop(&stack));
    printf("Popped element is %d\n", pop(&stack));

    return 0;
}
```

In this implementation:

- Each element (node) of the stack contains data and a pointer to the next element.
- We can dynamically allocate and deallocate memory as elements are pushed or popped from the stack.
- This implementation avoids the fixed size limitation of array-based stacks.

Depending on the required flexibility, stacks can have different designs. Let's solve Generate Parentheses problem. Given `n` pairs of parentheses, write a function to generate all combinations of well-formed parentheses. For example when `n = 3` then the output is:

```
["((()))","(()())","(())()","()(())","()()()"]
```

While the example of generating valid parentheses combinations doesn't directly involve an explicit stack data structure in the solution, the underlying concept of managing balanced parentheses does relate to stack-like behavior. Here's why:

1. Balanced Parentheses Problem:
    - The idea of generating or validating balanced parentheses is a classic use case of a stack. When you add an opening parenthesis `(`, it's like ”pushing” it onto a stack, and when you add a closing parenthesis `)`, you ”pop” an opening parenthesis from the stack to match it.
    - If you were asked to validate whether a given string of parentheses is well-formed, the best approach would be using a stack, where you push `(` and pop when encountering `)`. If the stack is empty at the end, the string is valid.

2. Implicit Stack in Recursion:

   - Recursion itself uses the call stack, where each recursive call pushes the current state onto the stack and returns (or pops) when done. In this sense, the recursive function in your solution is behaving similarly to a stack.
   - Each recursive call in your solution corresponds to adding or removing parentheses, which mimics the "push" and "pop" behavior of a stack.

You can implement an iterative approach using a stack instead of recursion. In this case, you'd maintain a stack to manage the state at each level of decision-making. Here's how an iterative solution using an explicit stack might look:

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

// Define a structure to hold the current state of parentheses
   generation
typedef struct {
 char *str;
 int open;
 int close;
 int index;
} StackState;

// Function to generate parentheses iteratively using a stack
char** generateParenthesis(int n, int* returnSize) {
 // Maximum possible size for the result array
 int capacity = 128;
 char** result = (char**)malloc(capacity * sizeof(char*));
 *returnSize = 0;

 // Allocate stack
 StackState* stack = (StackState*)malloc((2 * n + 1) * sizeof(
    StackState));
 int top = 0;

 // Initial state: empty string, no open or close parentheses
    used
 stack[top].str = (char*)malloc((2 * n + 1) * sizeof(char));
```

```c
  stack[top].str[0] = '\0';
  stack[top].open = 0;
  stack[top].close = 0;
  stack[top].index = 0;
  top++;

  while (top > 0) {
   // Pop the current state from the stack
   top--;
   StackState curr = stack[top];

   // If the current combination is complete, add it to the result
   if (curr.index == 2 * n) {
    if (*returnSize >= capacity) {
     capacity *= 2;
     result = realloc(result, capacity * sizeof(char*));
    }
    result[*returnSize] = (char*)malloc((2 * n + 1) * sizeof(char)
      );
    strcpy(result[*returnSize], curr.str);
    (*returnSize)++;
    continue;
   }

   // If we can add an opening parenthesis
   if (curr.open < n) {
    stack[top].str = (char*)malloc((2 * n + 1) * sizeof(char));
    strcpy(stack[top].str, curr.str);
    stack[top].str[curr.index] = '(';
    stack[top].str[curr.index + 1] = '\0';
    stack[top].open = curr.open + 1;
    stack[top].close = curr.close;
    stack[top].index = curr.index + 1;
    top++;
   }

   // If we can add a closing parenthesis
   if (curr.close < curr.open) {
    stack[top].str = (char*)malloc((2 * n + 1) * sizeof(char));
    strcpy(stack[top].str, curr.str);
    stack[top].str[curr.index] = ')';
    stack[top].str[curr.index + 1] = '\0';
    stack[top].open = curr.open;
    stack[top].close = curr.close + 1;
    stack[top].index = curr.index + 1;
    top++;
   }

   free(curr.str);  // Free the current string after using it
  }

  free(stack);  // Free the stack memory
  return result;
 }

 int main() {
  int n = 3;
  int returnSize;

  char** result = generateParenthesis(n, &returnSize);

  printf("Generated %d combinations:\n", returnSize);
  for (int i = 0; i < returnSize; i++) {
   printf("%s\n", result[i]);
   free(result[i]);  // Free each combination
  }

  free(result);  // Free the result array

  return 0;
 }
```

We could use a backtracking algorithm combined with a stack-like approach. For a given number of n pairs of parentheses, the total number of characters we need to place is `2 * n` (since each pair consists of one opening `(` and one closing `)` parenthesis). The goal is to generate valid combinations where each opening parenthesis has a corresponding closing parenthesis, following the rule that at no point can we have more closing parentheses than opening parentheses in the current sequence. Here is the coding using backtracking approach:

```
// Helper function to perform the backtracking
void generateParenthesisUtil(int n, int open, int close, char*
   str, int index, char*** result, int* count, int* capacity) {
 // If the current combination is complete, add it to the result
    array
 if (index == 2 * n) {
  str[index] = ’\0’;  // Null-terminate the string

   // If we’ve reached the capacity of result array, resize it
  if (*count >= *capacity) {
   *capacity *= 2;  // Double the capacity
   *result = realloc(*result, *capacity * sizeof(char*));
   if (*result == NULL) {
    perror("Realloc failed");
    exit(EXIT_FAILURE);
   }
  }

  (*result)[*count] = (char*)malloc((2 * n + 1) * sizeof(char));
  strcpy((*result)[*count], str);
  (*count)++;
  return;
 }

 // If we can add an opening parenthesis, add it and recurse
 if (open < n) {
  str[index] = ’(’;
  generateParenthesisUtil(n, open + 1, close, str, index + 1,
     result, count, capacity);
 }

 // If we can add a closing parenthesis, add it and recurse
 if (close < open) {
  str[index] = ’)’;
  generateParenthesisUtil(n, open, close + 1, str, index + 1,
     result, count, capacity);
 }
}
```

```c
// Main function to generate all valid combinations of parentheses
char** generateParenthesis(int n, int* returnSize) {
 int capacity = 128;  // Initial capacity
 char** result = (char**)malloc(capacity * sizeof(char*));
 char* str = (char*)malloc((2 * n + 1) * sizeof(char));  // String to hold the current combination
 int count = 0;

 // Check if allocation was successful
 if (result == NULL || str == NULL) {
  perror("Malloc failed");
  exit(EXIT_FAILURE);
 }

 generateParenthesisUtil(n, 0, 0, str, 0, &result, &count, &capacity);

 free(str);  // Free the temporary string used for combinations
 *returnSize = count;
 return result;
}
```

Stacks are crucial in many programming tasks, from memory management to expression evaluation and recursive function handling. In this section, we covered the basics of the Stack data structure, including both array-based and linked list-based implementations.

6.4 Queues

A queue is a linear data structure that follows the **First In First Out (FIFO)** principle. This means that the first element added to the queue will be the first one to be removed. Queues are commonly used in scenarios where order matters, such as in scheduling tasks, managing resources, and handling requests in a concurrent system.

In a queue, elements are added at the **rear** (or tail) and removed from the **front** (or head). This behavior can be visualized similarly to a line of people waiting for service: the first person in line is the first to be served. Some of key characteristics of Queues are:

- **FIFO Structure**: Elements are processed in the order they were added.
- **Operations**: The primary operations of a queue include:
- **Enqueue**: Add an element to the rear of the queue.
- **Dequeue**: Remove an element from the front of the queue.
- **Peek**: Retrieve the front element without removing it.
- **IsEmpty**: Check if the queue is empty.

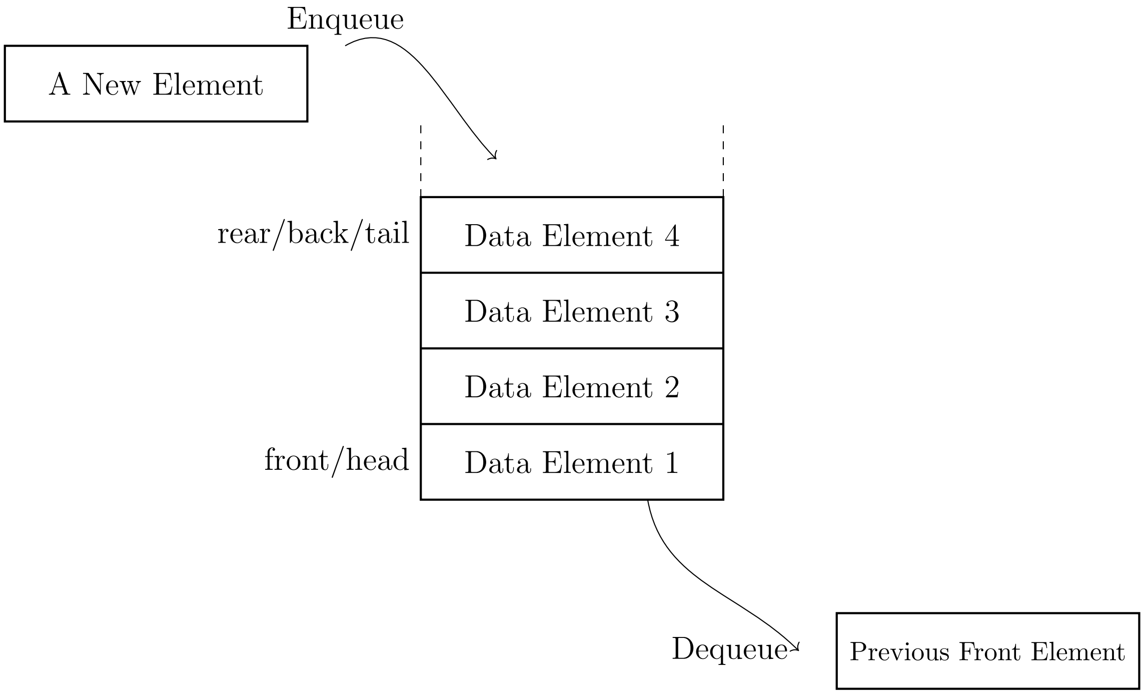


Figure 12: Queue Data Structure

Queues are widely used in various applications, including:

- Task Scheduling: Operating systems use queues to manage processes and tasks.
- Print Spooling: Print jobs are queued to ensure they are processed in order.
- Breadth-First Search (BFS): Queues are essential in graph traversal algorithms.
- Buffer Management: Queues are used in networking and data streaming to manage packets.

Let's start with basic operations on queues. A queue can be implemented using an array or a linked list. Here's how to implement a queue using a linked list in C:

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

typedef struct Node {
 int data;
 struct Node* next;
} Node;

typedef struct Queue {
 Node* front;
 Node* rear;
} Queue;

// Create a new queue
Queue* createQueue() {
 Queue* q = (Queue*)malloc(sizeof(Queue));
 q->front = q->rear = NULL;
 return q;
}

// Enqueue operation
void enqueue(Queue* q, int value) {
 Node* newNode = (Node*)malloc(sizeof(Node));
 newNode->data = value;
 newNode->next = NULL;

 if (q->rear == NULL) {
  q->front = q->rear = newNode; // First element
  return;
 }
 q->rear->next = newNode;
 q->rear = newNode;
}

// Dequeue operation
int dequeue(Queue* q) {
 if (q->front == NULL) {
  printf("Queue is empty\n");
  return -1; // Indicate queue is empty
 }
 Node* temp = q->front;
 int value = temp->data;
 q->front = q->front->next;
 if (q->front == NULL) {
  q->rear = NULL; // Queue is empty
 }
 free(temp);
 return value;
}

// Peek operation
int peek(Queue* q) {
 if (q->front == NULL) {
  printf("Queue is empty\n");
  return -1; // Indicate queue is empty
 }
 return q->front->data;
}

// Check if queue is empty
int isEmpty(Queue* q) {
 return q->front == NULL;
}
```

A circular queue is an advanced version of a linear queue where the last position is connected back to the first position, forming a circle. This eliminates the waste of space that can occur in a linear queue.

```
#define SIZE 5

typedef struct CircularQueue {
 int items[SIZE];
 int front;
 int rear;
} CircularQueue;

// Create a new circular queue
CircularQueue* createCircularQueue() {
 CircularQueue* cq = (CircularQueue*)malloc(sizeof(CircularQueue)
    );
 cq->front = -1;
 cq->rear = -1;
 return cq;
}

// Enqueue operation for circular queue
void enqueueCircular(CircularQueue* cq, int value) {
 if ((cq->front == 0 && cq->rear == SIZE - 1) || (cq->front == cq
    ->rear + 1)) {
  printf("Circular Queue is full\n");
  return; // Queue is full
 }

 if (cq->front == -1) {
  cq->front = 0; // First element
 }
 cq->rear = (cq->rear + 1) % SIZE; // Move rear to next position
 cq->items[cq->rear] = value;
}

// Dequeue operation for circular queue
int dequeueCircular(CircularQueue* cq) {
 if (cq->front == -1) {
  printf("Circular Queue is empty\n");
  return -1; // Indicate queue is empty
 }

 int value = cq->items[cq->front];
 if (cq->front == cq->rear) {
  cq->front = cq->rear = -1; // Queue is empty
 } else {
  cq->front = (cq->front + 1) % SIZE; // Move front to next
     position
 }
 return value;
}
```

A priority queue is an abstract data type that operates similarly to a regular queue but allows elements to be dequeued based on priority rather than order. Higher priority elements are removed before lower priority ones.

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

typedef struct MinHeap {
 int* arr;
 int capacity;
 int size;
} MinHeap;

MinHeap* createMinHeap(int capacity) {
 MinHeap* minHeap = (MinHeap*)malloc(sizeof(MinHeap));
 minHeap->capacity = capacity;
 minHeap->size = 0;
 minHeap->arr = (int*)malloc(capacity * sizeof(int));
 return minHeap;
}

void swap(int* a, int* b) {
 int temp = *a;
 *a = *b;
 *b = temp;
}

void insertMinHeap(MinHeap* minHeap, int value) {
 if (minHeap->size == minHeap->capacity) {
  printf("Heap is full\n");
  return;
 }
 minHeap->size++;
 int i = minHeap->size - 1;
 minHeap->arr[i] = value;

 while (i != 0 && minHeap->arr[(i - 1) / 2] > minHeap->arr[i]) {
  swap(&minHeap->arr[i], &minHeap->arr[(i - 1) / 2]);
  i = (i - 1) / 2;
 }
}

int removeMin(MinHeap* minHeap) {
 if (minHeap->size <= 0) {
  return -1; // Heap is empty
 }
 if (minHeap->size == 1) {
  minHeap->size--;
  return minHeap->arr[0];
 }

 int root = minHeap->arr[0];
 minHeap->arr[0] = minHeap->arr[minHeap->size - 1];
 minHeap->size--;
 // Heapify down
 int i = 0;
 while (i < minHeap->size / 2) {
  int smallest = i;
  if (2 * i + 1 < minHeap->size && minHeap->arr[2 * i + 1] <
     minHeap->arr[smallest]) {
   smallest = 2 * i + 1;
  }
  if (2 * i + 2 < minHeap->size && minHeap->arr[2 * i + 2] <
     minHeap->arr[smallest]) {
   smallest = 2 * i + 2;
  }
  if (smallest == i) break;
  swap(&minHeap->arr[i], &minHeap->arr[smallest]);
  i = smallest;
 }
 return root;
}
```

Using two stacks, you can implement a queue. The idea is to use one stack for enqueue operations and another for dequeue operations.

```c
typedef struct QueueUsingStacks {
 Stack* stack1; // For enqueue
 Stack* stack2; // For dequeue
} QueueUsingStacks;
```

```c
void enqueueUsingStacks(QueueUsingStacks* q, int value) {
 push(q->stack1, value); // Push to stack1
}

int dequeueUsingStacks(QueueUsingStacks* q) {
 if (isEmpty(q->stack2)) {
  while (!isEmpty(q->stack1)) {
   push(q->stack2, pop(q->stack1)); // Transfer elements from stack1 to stack2
  }
 }
 return pop(q->stack2); // Pop from stack2
}
```

A circular buffer can be implemented using a queue, allowing for efficient use of space and fixed-size storage. Implementing a circular buffer using a queue is an effective way to manage a fixed-size buffer. A circular buffer allows for efficient data management, especially in scenarios where data is continuously written and read, such as in streaming data applications. Below is a simple implementation of a circular buffer in C. For more practice you can write the code for a circular buffer.

Queues are a fundamental data structure with a variety of applications, including scheduling, resource management, and task processing. Understanding how to implement and utilize different types of queues, such as circular queues and priority queues, equips students with valuable skills for solving real-world problems. Additionally, implementing queues using other data structures can deepen their understanding of data management and manipulation.

6.5 Trees

A tree is a hierarchical data structure that simulates a parent-child relationship between elements. Unlike linear data structures like arrays, linked lists, stacks, or queues, trees are non-linear and can efficiently represent relationships in a more complex manner.

Trees are used in various real-world applications such as database indexing, hierarchical structures (e.g., file systems), network routing algorithms, and expression parsing in compilers. Key terminology of trees:

- **Node**: Basic unit containing data and references to other nodes.

- **Root**: The top node of the tree, from which all other nodes descend.
- **Parent**: A node that has children.
- **Child**: A node that is a descendant of another node.
- **Leaf**: A node that does not have any children.
- **Subtree**: A tree formed by a node and its descendants.
- **Height**: The length of the longest path from the root to a leaf.
- **Depth**: The number of edges from the root to a node.
- **Diameter** The longest path between two nodes in the tree, by counting the number of edges in between.

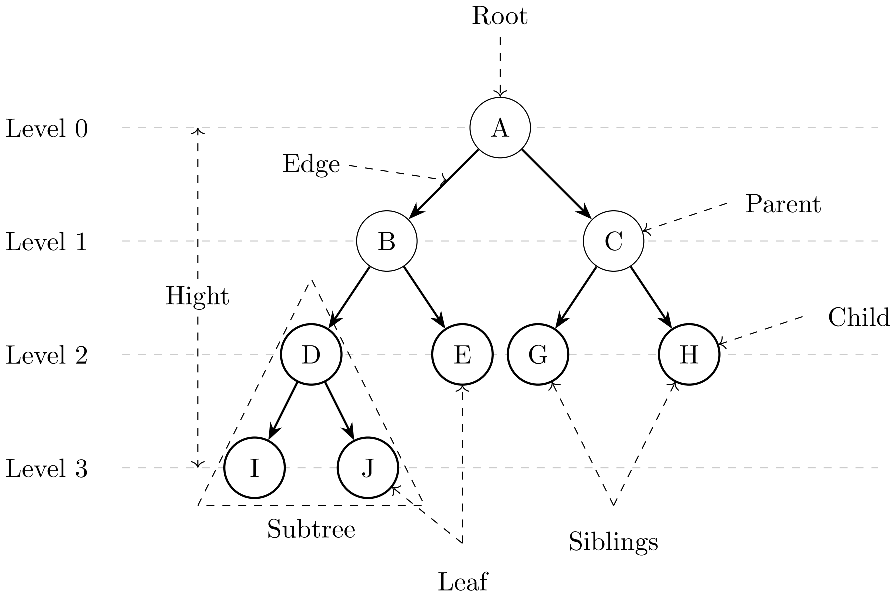


Figure 13: Tree Data Structure

Let's start with fundamental operations like insertion, deletion, and traversal. Inserting into a binary tree typically involves finding an appropriate spot where the new node maintains the tree's properties.

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

typedef struct Node {
 int data;
 struct Node* left;
 struct Node* right;
}Node;

Node* createNode(int data) {
 Node* newNode = (Node*)malloc(sizeof(Node));
 newNode->data = data;
 newNode->left = newNode->right = NULL;
 return newNode;
}

void insert(Node** root, int data) {
 if (*root == NULL) {
  *root = createNode(data);
  return;
 }
 if (data < (*root)->data)
 insert(&(*root)->left, data);
 else
 insert(&(*root)->right, data);
}

void inorderTraversal(Node* root) {
 if (root != NULL) {
  inorderTraversal(root->left);
  printf("%d ", root->data);
  inorderTraversal(root->right);
 }
}

int main() {
 Node* root = NULL;
 insert(&root, 50);
 insert(&root, 30);
 insert(&root, 20);
 insert(&root, 40);
 insert(&root, 70);
 insert(&root, 60);
 insert(&root, 80);
 printf("Inorder traversal: ");
 inorderTraversal(root);
 return 0;
}
```

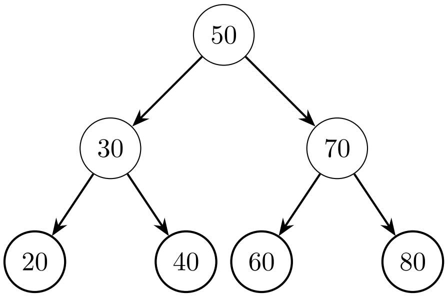


Figure 14: Binary Search Tree constructed with the values 50, 30, 20, 40, 70, 60, and 80.

Printing the value in traversal order we have:

```
 Inorder traversal: 20 30 40 50 60 70 80
```

Inverting a binary tree means swapping the left and right children of every node. It's a simple yet fundamental operation. For example if we have the following tree:

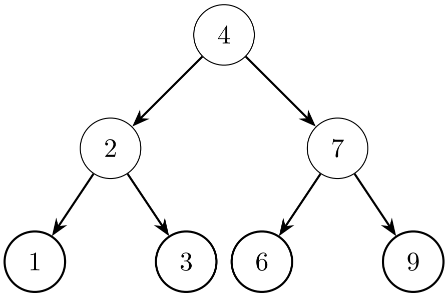


Figure 15: Binary Tree Before Invert Function

The inverted tree becomes:

The C code to invert a Binary Tree:

```
Node* invertTree(Node* root) {
 if (root == NULL) return NULL;
 struct Node* temp = root->left;
 root->left = root->right;
 root->right = temp;
```

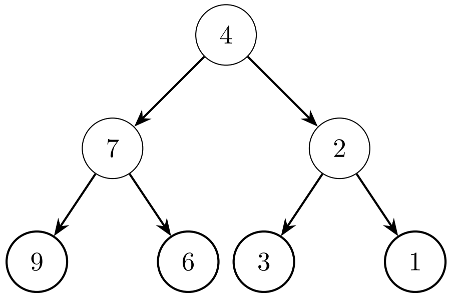


Figure 16: Binary Tree After Invert Function

```
 invertTree(root->left);
 invertTree(root->right);
 return root;
}
```

The depth of a binary tree is the longest path from the root to a leaf node. It can be calculated recursively by checking the height of the left and right subtrees and returning the greater of the two.

```
int maxDepth(Node* root) {
 if (root == NULL) return 0;
 int leftDepth = maxDepth(root->left);
 int rightDepth = maxDepth(root->right);
 return (leftDepth > rightDepth ? leftDepth : rightDepth) + 1;
}
```

The diameter of a binary tree is the length of the longest path between two nodes in a tree. This path may or may not pass through the root.

```
int diameter(Node* root, int* height) {
 if (root == NULL) {
  *height = 0;
  return 0;
 }

 int leftHeight = 0, rightHeight = 0;
 int leftDiameter = diameter(root->left, &leftHeight);
 int rightDiameter = diameter(root->right, &rightHeight);

 *height = (leftHeight > rightHeight ? leftHeight : rightHeight)
    + 1;
```

```
   return fmax(leftHeight + rightHeight + 1, fmax(leftDiameter,
      rightDiameter));
  }
```

A Binary Search Tree is a binary tree where each node follows the rule:

- The value of the left child is less than the value of the parent.
- The value of the right child is greater than the value of the parent.

BSTs are useful for searching and sorting operations, offering average time complexity of $O(logn)$ for search, insert, and delete operations.

A balanced binary tree is one in which the height difference between the left and right subtrees of any node is at most one. This ensures optimal time complexity for insertion, deletion, and search operations. To check if a tree is balanced:

```
  int isBalanced(Node* root) {
   if (root == NULL) return 1;

   int leftHeight = maxDepth(root->left);
   int rightHeight = maxDepth(root->right);

   return abs(leftHeight - rightHeight) <= 1 && isBalanced(root->
      left) && isBalanced(root->right);
  }
```

Trees are a versatile and foundational data structure in computer science. From basic operations like insertion and traversal to more advanced operations like tree inversion and depth calculation, they are used in various real-world applications such as databases, file systems, and algorithms. By mastering binary trees and their specialized forms like Binary Search Trees (BST) and Balanced Trees, students can efficiently solve many computational problems.

6.6 Tries

A Trie, often referred to as a prefix tree, is a specialized tree data structure that is particularly well-suited for managing a dynamic set of strings, where the keys are usually strings. The term ”Trie” is derived from the word ”retrieval.” Tries are designed to facilitate fast lookups, insertions, and deletions of keys while providing efficient storage. Some key characteristics of tries:

- **Structure**: A Trie consists of nodes, where each node represents a character of a string. The path from the root to a particular node represents a prefix of some strings stored in the Trie.
- **Root Node**: The root node is usually empty and serves as the starting point for all strings.
- **Children Nodes**: Each node can have multiple children corresponding to different characters.
- **End of Word**: A boolean flag is typically associated with nodes to indicate whether a particular node marks the end of a valid word.

(A figure in progress)

Applications of Tries are:

- **Autocomplete Systems**: Tries are widely used in search engines and text editors to provide suggestions as users type.
- **Spell Checkers**: They help quickly verify whether a word exists in a dictionary.
- **IP Routing**: Tries can be used to store and search routing prefixes efficiently.
- **Longest Common Prefix**: They can be used to find the longest common prefix among a set of strings.

Let's start we basic implementations. To insert a string into a Trie, start from the root and traverse down the tree, creating nodes as necessary for each character in the string. Once the last character is inserted, mark that node as the end of a word.

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

#define ALPHABET_SIZE 26

typedef struct TrieNode {
 struct TrieNode *children[ALPHABET_SIZE];
 bool isEndOfWord;
} TrieNode;

TrieNode* getNode() {
 TrieNode* node = (TrieNode*)malloc(sizeof(TrieNode));
 node->isEndOfWord = false;
 for (int i = 0; i < ALPHABET_SIZE; i++)
 node->children[i] = NULL;
 return node;
}

void insert(TrieNode *root, const char *key) {
 TrieNode *pCrawl = root;
 for (int level = 0; key[level] != ’\0’; level++) {
  int index = key[level] - ’a’; // Assuming lowercase letters
  if (!pCrawl->children[index])
  pCrawl->children[index] = getNode();
  pCrawl = pCrawl->children[index];
 }
 pCrawl->isEndOfWord = true; // Mark end of word
}
```

To search for a string in a Trie, follow the path defined by the characters of the string. If you reach a node that signifies the end of the string and the end of the word flag is true, then the string exists in the Trie.

```
bool search(TrieNode *root, const char *key) {
 TrieNode *pCrawl = root;
 for (int level = 0; key[level] != ’\0’; level++) {
  int index = key[level] - ’a’;
  if (!pCrawl->children[index])
  return false;
  pCrawl = pCrawl->children[index];
 }
 return (pCrawl != NULL && pCrawl->isEndOfWord);
}
```

Deletion is slightly more complex. To delete a string, you need to traverse down the tree as you would for searching. If the end of the word is reached, unmark the end of the word flag. If the node has no children, you can remove the node and backtrack.

```
bool deleteHelper(TrieNode *root, const char *key, int depth) {
 if (!root)
 return false;
 if (depth == strlen(key)) {
  if (root->isEndOfWord) {
   root->isEndOfWord = false; // Unmark end of word
   return (root->children[0] == NULL); // Return true if no
      children
  }
  return false; // Key doesn't exist
 }

 int index = key[depth] - 'a';
 if (deleteHelper(root->children[index], key, depth + 1)) {
  free(root->children[index]); // Delete child node
  root->children[index] = NULL;
  return (root->isEndOfWord == false && root->children[0] == NULL
     ); // Check if node can be deleted
 }
 return false;
}

void deleteKey(TrieNode *root, const char *key) {
 deleteHelper(root, key, 0);
}
```

Using a Trie for autocomplete functionality involves finding all the words that share a common prefix.

```
void printSuggestions(TrieNode *root, char *prefix) {
 if (root->isEndOfWord)
 printf("%s\n", prefix);

 for (int i = 0; i < ALPHABET_SIZE; i++) {
  if (root->children[i]) {
   char newPrefix[100];
   snprintf(newPrefix, sizeof(newPrefix), "%s%c", prefix, 'a' + i
      );
   printSuggestions(root->children[i], newPrefix);
  }
 }
}
```

```
void autocomplete(TrieNode *root, const char *prefix) {
 TrieNode *pCrawl = root;
 for (int level = 0; prefix[level] != ’\0’; level++) {
  int index = prefix[level] - ’a’;
  if (!pCrawl->children[index])
  return; // No suggestions
  pCrawl = pCrawl->children[index];
 }
 printSuggestions(pCrawl, (char *)prefix);
}
```

To find the longest common prefix among an array of strings using a Trie, you can insert all the strings into the Trie and then traverse the Trie until you hit a node with multiple children or the end of a word.

```
char* longestCommonPrefix(TrieNode *root) {
 TrieNode *pCrawl = root;
 char *prefix = (char *)malloc(100);
 int index = 0;

 while (pCrawl && !pCrawl->isEndOfWord) {
  int childCount = 0;
  int childIndex = -1;
  for (int i = 0; i < ALPHABET_SIZE; i++) {
   if (pCrawl->children[i]) {
    childCount++;
    childIndex = i;
   }
  }
  if (childCount != 1)
  break; // More than one child, stop
  prefix[index++] = ’a’ + childIndex;
  pCrawl = pCrawl->children[childIndex];
 }
 prefix[index] = ’\0’;
 return prefix;
}
```

Tries are a powerful data structure for managing strings and implementing efficient search functionalities. Understanding how to insert, search, delete, and utilize tries for advanced operations such as autocomplete and finding the longest common prefix will equip students with valuable skills for handling string-related challenges in programming. Tries can outperform other data structures in certain scenarios, especially when dealing with large sets of strings or when prefix-based queries are common.

6.7 Heap

A heap is a specialized binary tree-based data structure that satisfies the heap property. Heaps are useful for implementing priority queues and for efficient sorting algorithms such as Heapsort. There are two types of heaps:

- Max-Heap: In a max-heap, the key of a parent node is always greater than or equal to the keys of its children.

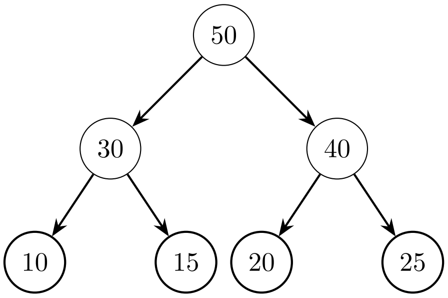


Figure 17: Max Heap

- Min-Heap: In a min-heap, the key of a parent node is always less than or equal to the keys of its children.

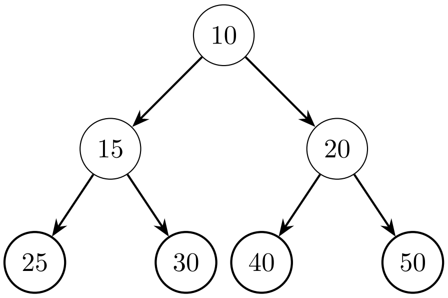


Figure 18: Min Heap

The main advantage of using a heap is its ability to maintain the heap property during insertion and deletion operations in logarithmic time, making it suitable for tasks like scheduling and event simulation. Some of heap applications are:

- **Priority Queues**: Heaps are often used to implement priority queues, where the highest (or lowest) priority element is retrieved first.
- **Heapsort**: Heaps provide an efficient way to sort an array in-place with a time complexity of $O(nlogn)$.
- **Graph Algorithms**: Heaps are used in algorithms like Dijkstra's shortest path algorithm and Prim's minimum spanning tree algorithm.
- **Memory Management**: Heaps are used to manage memory allocation dynamically.

Before we start programming heaps, it is necessary to understand the basic concepts, including:

1. **Complete Binary Tree**: A heap is a complete binary tree, meaning all levels are completely filled except possibly for the last level, which is filled from left to right.
2. **Heap Property**: For a max-heap, the key at each node must be greater than or equal to the keys of its children, and for a min-heap, the key at each node must be less than or equal to the keys of its children.
3. **Heap Representation**: Heaps are often implemented using arrays rather than linked structures. The parent-child relationships are managed using simple arithmetic:
    - Parent of node at index i: $Parent(i) = \frac{i-1}{2}$
    - Left child of node at index i: $Left(i) = 2i + 1$
    - Right child of node at index i: $Right(i) = 2i + 1$

Heapify is the process used to restore the heap property after insertion or deletion. The two main operations are:

1. **Heapify up**: Restores the heap property after insertion.
2. **Heapify down**: Restores the heap property after deletion of the root.

Inserting an element into a max-heap requires placing the new element at the end of the heap (at the next available leaf position). Then, the element is moved up the tree until the heap property is restored, a process known as **heapifying up**. Here is a basic implementation of insertion in a max-heap:

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

#define MAX 100

typedef struct {
 int array[MAX];
 int size;
} MaxHeap;

MaxHeap* createHeap() {
 MaxHeap* heap = (MaxHeap*)malloc(sizeof(MaxHeap));
 heap->size = 0;
 return heap;
}

void heapifyUp(MaxHeap* heap, int index) {
 int parentIndex = (index - 1) / 2;
 if (index > 0 && heap->array[index] > heap->array[parentIndex])
    {
  // Swap with parent
  int temp = heap->array[index];
  heap->array[index] = heap->array[parentIndex];
  heap->array[parentIndex] = temp;
  heapifyUp(heap, parentIndex);
 }
}

void insert(MaxHeap* heap, int value) {
 if (heap->size == MAX) {
  printf("Heap is full!\n");
  return;
 }
 // Insert at the end
 heap->array[heap->size] = value;
 heap->size++;
 // Heapify up
 heapifyUp(heap, heap->size - 1);
}

void printHeap(MaxHeap* heap) {
 for (int i = 0; i < heap->size; i++) {
  printf("%d ", heap->array[i]);
 }
 printf("\n");
}

int main() {
 MaxHeap* heap = createHeap();
 insert(heap, 10);
 insert(heap, 20);
 insert(heap, 5);
 insert(heap, 30);
 insert(heap, 15);

 printHeap(heap); // Output: 30 20 5 10 15
 free(heap);
 return 0;
}
```

Deletion in a max-heap usually involves removing the root node (the largest element in the max-heap). After removal, the last element in the heap replaces the root, and the heap is **heapified down** to maintain the heap property.

```c
void heapifyDown(MaxHeap* heap, int index) {
 int left = 2 * index + 1;
 int right = 2 * index + 2;
 int largest = index;

 if (left < heap->size && heap->array[left] > heap->array[largest
    ]) {
  largest = left;
 }
 if (right < heap->size && heap->array[right] > heap->array[
    largest]) {
  largest = right;
 }
 if (largest != index) {
  // Swap
  int temp = heap->array[index];
  heap->array[index] = heap->array[largest];
  heap->array[largest] = temp;
  heapifyDown(heap, largest);
 }
}

int deleteRoot(MaxHeap* heap) {
 if (heap->size == 0) {
  printf("Heap is empty!\n");
  return -1;
 }

 int rootValue = heap->array[0];
 heap->array[0] = heap->array[heap->size - 1];
 heap->size--;
 heapifyDown(heap, 0);

 return rootValue;
}
```

Heapsort is an efficient, comparison-based sorting algorithm that leverages the heap data structure. The algorithm has a time complexity of $O(nlogn)$ and works by building a max-heap from the input data, then repeatedly extracting the maximum element and rebuilding the heap.

```
void heapSort(int array[], int size) {
 MaxHeap* heap = createHeap();

 // Build the heap
 for (int i = 0; i < size; i++) {
  insert(heap, array[i]);
 }

 // Extract elements one by one
 for (int i = size - 1; i >= 0; i--) {
  array[i] = deleteRoot(heap);
 }
 free(heap);
}
```

The problem **Kth Largest Element in a Stream** requires maintaining a running stream of numbers and being able to efficiently retrieve the Kth largest element after every new insertion. This problem can be efficiently solved using a min-heap (also known as a priority queue). We use a **min-heap** of size k, where the smallest element in the heap is the Kth largest element. The approach we need to take is:

1. We maintain a min-heap of size k.
2. Each time a new number is added:
   - If the heap has fewer than k elements, we insert the number.
   - If the heap already contains k elements, we compare the new number with the root (smallest element in the heap). If the new number is larger, we replace the root with the new number and reheapify.
3. The root of the heap will always contain the Kth largest element.

Here's how we can implement this approach in C:

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

#define MAX 100
\codebox{
typedef struct {
 int* array;
 int size;
 int capacity;
} MinHeap;

MinHeap* createMinHeap(int capacity) {
 MinHeap* heap = (MinHeap*)malloc(sizeof(MinHeap));
 heap->array = (int*)malloc(sizeof(int) * capacity);
 heap->size = 0;
 heap->capacity = capacity;
 return heap;
}

void swap(int* a, int* b) {
 int temp = *a;
 *a = *b;
 *b = temp;
}

void heapifyDown(MinHeap* heap, int index) {
 int left = 2 * index + 1;
 int right = 2 * index + 2;
 int smallest = index;\codebox{

 if (left < heap->size && heap->array[left] < heap->array[
    smallest]) {
  smallest = left;
 }
 if (right < heap->size && heap->array[right] < heap->array[
    smallest]) {
  smallest = right;
 }
 if (smallest != index) {
  swap(&heap->array[index], &heap->array[smallest]);
  heapifyDown(heap, smallest);
 }
}

void heapifyUp(MinHeap* heap, int index) {
 int parent = (index - 1) / 2;
 if (index > 0 && heap->array[index] < heap->array[parent]) {
  swap(&heap->array[index], &heap->array[parent]);
  heapifyUp(heap, parent);
 }
}

void insert(MinHeap* heap, int value) {
 if (heap->size < heap->capacity) {
  heap->array[heap->size] = value;
  heap->size++;
  heapifyUp(heap, heap->size - 1);
 } else if (value > heap->array[0]) {
  heap->array[0] = value;
  heapifyDown(heap, 0);
 }
}

int getKthLargest(MinHeap* heap) {
 return heap->array[0];  // The root of the min-heap is the Kth
    largest element.
}

int main() {
 int k = 3;
 MinHeap* heap = createMinHeap(k);

 int stream[] = {4, 5, 8, 2, 6, 9};
 int n = sizeof(stream) / sizeof(stream[0]);

 for (int i = 0; i < n; i++) {
  insert(heap, stream[i]);
  if (heap->size == k) {
   printf("After inserting %d, the %d-th largest element is: %d\n
      ", stream[i], k, getKthLargest(heap));
  }
 }

 free(heap->array);
 free(heap);
 return 0;
}
```

We maintain a heap of size k, where every insertion ensures that the Kth largest element is always at the root. For example, if $k = 3$, after inserting numbers 4, 5, 8, 2, 6, and 9 into the stream, the program will print the 3rd largest number after each insertion where the heap reaches size k.

The problem **Merge k Sorted Lists** requires merging k sorted linked lists into a single sorted linked list. A min-heap can help solve this problem efficiently. The basic idea is to repeatedly extract the smallest element from all the lists until all the lists are empty. First, we push the first element of each list into a min-heap. Then, we extract the smallest element from the heap, append it to the merged list, and then insert the next element from the same list into the heap. Repeat this process until all lists are exhausted. Here is the implementation in C:

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

// Definition for singly-linked list.
struct ListNode {
 int val;
 struct ListNode* next;
};

typedef struct ListNode ListNode;

typedef struct {
 ListNode** array;
 int size;
} MinHeap;

// Create a new linked list node
ListNode* createNode(int val) {
 ListNode* newNode = (ListNode*)malloc(sizeof(ListNode));
 newNode->val = val;
 newNode->next = NULL;
 return newNode;
}

// Swap two list nodes in heap
void swap(ListNode** a, ListNode** b) {
 ListNode* temp = *a;
 *a = *b;
 *b = temp;
}

// Heapify down
void heapifyDown(MinHeap* heap, int i) {
 int left = 2 * i + 1;
 int right = 2 * i + 2;
 int smallest = i;

 if (left < heap->size && heap->array[left]->val < heap->array[
    smallest]->val) {
  smallest = left;
 }
 if (right < heap->size && heap->array[right]->val < heap->array[
    smallest]->val) {
  smallest = right;
 }
 if (smallest != i) {
  swap(&heap->array[i], &heap->array[smallest]);
  heapifyDown(heap, smallest);
 }
}

// Insert into the heap
void insertHeap(MinHeap* heap, ListNode* node) {
 heap->array[heap->size] = node;
 heap->size++;

 // Heapify up
 int i = heap->size - 1;
 int parent = (i - 1) / 2;
 while (i > 0 && heap->array[i]->val < heap->array[parent]->val)
    {
  swap(&heap->array[i], &heap->array[parent]);
  i = parent;
  parent = (i - 1) / 2;
 }
}

// Extract min from heap
ListNode* extractMin(MinHeap* heap) {
 if (heap->size == 0) {
  return NULL;
 }
 ListNode* root = heap->array[0];
 heap->array[0] = heap->array[heap->size - 1];
 heap->size--;
 heapifyDown(heap, 0);
 return root;
```

```c
}

// Function to merge k sorted lists
ListNode* mergeKLists(ListNode** lists, int k) {
 MinHeap heap;
 heap.array = (ListNode**)malloc(sizeof(ListNode*) * k);
 heap.size = 0;

 // Insert first node of each list into the heap
 for (int i = 0; i < k; i++) {
  if (lists[i] != NULL) {
   insertHeap(&heap, lists[i]);
  }
 }

 // Create a dummy node for the merged list
 ListNode* dummy = creat\codebox{eNode(0);
 ListNode* tail = dummy;

 // Extract min and insert next node from the list
 while (heap.size > 0) {
  ListNode* minNode = extractMin(&heap);
  tail->next = minNode;
  tail = minNode;
  if (minNode->next != NULL) {
   insertHeap(&heap, minNode->next);
  }
 }

 ListNode* result = dummy->next;
 free(dummy);
 free(heap.array);
 return result;
}

// Helper function to print the list
void printList(ListNode* head) {
 while (head != NULL) {
  printf("%d -> ", head->val);
  head = head->next;
 }
 printf("NULL\n");
}

int main() {
 // Create test cases
 ListNode* l1 = createNode(1);
 l1->next = createNode(4);
 l1->next->next = createNode(5);

 ListNode* l2 = createNode(1);
 l2->next = createNode(3);
 l2->next->next = createNode(4);

 ListNode* l3 = createNode(2);
 l3->next = createNode(6);

 ListNode* lists[] = {l1, l2, l3};

 // Merge k sorted lists
 ListNode* mergedList = mergeKLists(lists, 3);
 printList(mergedList);  // Output: 1 -> 1 -> 2 -> 3 -> 4 -> 4 ->
     5 -> 6 -> NULL

 return 0;
}
```

For more practice, find a problem that uses max heap. Heaps are an essential data structure for priority management and efficient sorting. Their use extends across algorithms that require fast retrieval of the minimum or maximum value, with applications in priority queues, graph algorithms, and even memory management systems.

6.8 Graphs

A graph is a powerful and versatile data structure used to represent relationships between different entities. A graph is defined as a set of **vertices** (also called nodes) connected by **edges**. The vertices represent entities, and the edges represent the relationships or connections between them.

Graphs are used extensively in computer science for modeling real-world systems such as networks (social networks, computer networks), maps (GPS systems), and many optimization problems (e.g., shortest path).

Graphs are classified into several types based on their properties:

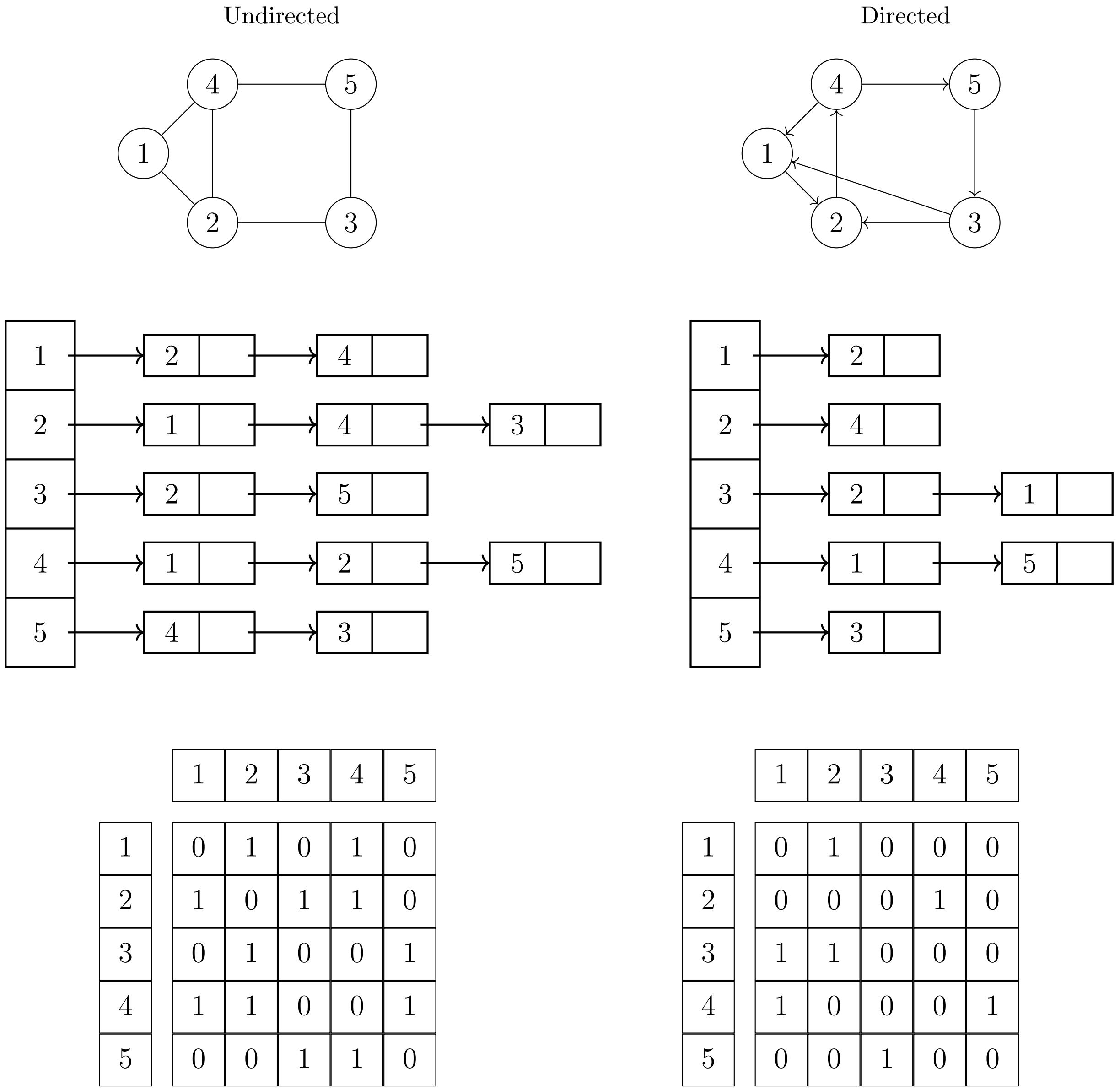
- **Undirected Graph**: The edges have no direction. For example, if there is an edge between vertex A and vertex B, you can traverse it in both directions.
- **Directed Graph (Digraph)**: The edges have directions. An edge from vertex A to vertex B does not imply an edge from B to A.
- **Weighted Graph**: The edges have weights or costs associated with them. Weighted graphs are often used in problems that require finding the shortest path or the minimum cost.

Graphs can be represented in two primary ways:

1. **Adjacency Matrix**: A 2D array is used where `matrix[i][j] = 1` (or the weight of the edge) if there is an edge from vertex `i` to vertex `j`.
2. **Adjacency List**: Each vertex has a list of adjacent vertices (i.e., vertices that are directly connected to it by an edge). This representation is more space-efficient for sparse graphs.

Let's try an example using Adjacency List in C:

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

// Structure to represent a node in the adjacency list
struct Node {
 int vertex;
 struct Node* next;
};

// Structure to represent an adjacency list
struct Graph {
 int numVertices;
 struct Node** adjLists;
};

// Create a node
```



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

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

Figure 19: Undirected and Directed Graphs with Adjacency List and Matrix Representation

```
struct Node* createNode(int v) {
 struct Node* newNode = malloc(sizeof(struct Node));
 newNode->vertex = v;
 newNode->next = NULL;
 return newNode;
}

// Create a graph
struct Graph* createGraph(int vertices) {
 struct Graph* graph = malloc(sizeof(struct Graph));
 graph->numVertices = vertices;
 graph->adjLists = malloc(vertices * sizeof(struct Node*));

 for (int i = 0; i < vertices; i++)
 graph->adjLists[i] = NULL;

 return graph;
}

// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
 struct Node* newNode = createNode(dest);
 newNode->next = graph->adjLists[src];
 graph->adjLists[src] = newNode;

 // Since it's an undirected graph, add an edge from dest to src
    also
 newNode = createNode(src);
 newNode->next = graph->adjLists[dest];
 graph->adjLists[dest] = newNode;
}

// Print the graph
void printGraph(struct Graph* graph) {
 for (int v = 0; v < graph->numVertices; v++) {
  struct Node* temp = graph->adjLists[v];
  printf("Vertex %d\n: ", v);
  while (temp) {
   printf("%d -> ", temp->vertex);
   temp = temp->next;
  }
  printf("\n");
 }
}

int main() {
 int vertices = 5;
 struct Graph* graph = createGraph(vertices);
```

```c
 addEdge(graph, 0, 1);
 addEdge(graph, 0, 4);
 addEdge(graph, 1, 4);
 addEdge(graph, 1, 3);
 addEdge(graph, 1, 2);
 addEdge(graph, 2, 3);
 addEdge(graph, 3, 4);

 printGraph(graph);

 return 0;
}
```

Graph traversal is the process of visiting all the vertices in a graph. There are two primary algorithms for traversing graphs:

1. **Depth-First Search (DFS)**: This algorithm starts at a vertex and explores as far as possible along each branch before backtracking. It uses a stack (or recursion). The code below here is an example of implementing DFS in C:

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

   #define MAX 100

   int visited[MAX] = {0};

   // DFS function
   void DFS(int vertex, int adjMatrix[][MAX], int numVertices) {
    visited[vertex] = 1;
    printf("Visited %d\n", vertex);

    for (int i = 0; i < numVertices; i++) {
     if (adjMatrix[vertex][i] == 1 && !visited[i]) {
      DFS(i, adjMatrix, numVertices);
     }
    }
   }
   ```

```c
int main() {
 int numVertices = 5;
 int adjMatrix[MAX][MAX] = { {0, 1, 0, 1, 0},
  {1, 0, 1, 0, 1},
  {0, 1, 0, 1, 0},
  {1, 0, 1, 0, 1},
  {0, 1, 0, 1, 0} };

 DFS(0, adjMatrix, numVertices);

 return 0;
}
```

2. **Breadth-First Search (BFS)**: This algorithm starts at a vertex and explores all of its neighbors before moving to the next level of neighbors. It uses a queue. Here is a C implementation of BFS in C:

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

#define MAX 100

int queue[MAX], front = -1, rear = -1;
int visited[MAX] = {0};

// Enqueue function
void enqueue(int vertex) {
 if (rear == MAX - 1)
 return;
 if (front == -1)
 front = 0;
 rear++;
 queue[rear] = vertex;
}

// Dequeue function
int dequeue() {
 if (front == -1)
 return -1;
 int vertex = queue[front];
 if (front == rear)
 front = rear = -1;
 else
 front++;
 return vertex;
}

// BFS function
void BFS(int startVertex, int adjMatrix[][MAX], int
   numVertices) {
 visited[startVertex] = 1;
 enqueue(startVertex);

 while (front != -1) {
  int currentVertex = dequeue();
  printf("Visited %d\n", currentVertex);

  for (int i = 0; i < numVertices; i++) {
   if (adjMatrix[currentVertex][i] == 1 && !visited[i]) {
    enqueue(i);
    visited[i] = 1;
   }
  }
 }
}

int main() {
 int numVertices = 5;
 int adjMatrix[MAX][MAX] = { {0, 1, 0, 1, 0},
  {1, 0, 1, 0, 1},
  {0, 1, 0, 1, 0},
  {1, 0, 1, 0, 1},
  {0, 1, 0, 1, 0} };

 BFS(0, adjMatrix, numVertices);

 return 0;
}
```

Graphs have a wide range of applications, and as students become more familiar with basic concepts, they can explore more advanced topics. Below are some crucial advanced topics that students need to understand:

1. **Topological Sorting**: This algorithm is used to order the vertices of a **Directed Acyclic Graph (DAG)** in a linear sequence where for every directed edge u → v, vertex u comes before vertex v in the ordering. Topological sorting is commonly used in scenarios like task scheduling, where certain tasks must be performed before others.

2. **Dijkstra's Algorithm**: This is a **single-source shortest path algorithm** for finding the shortest paths from a source vertex to all other vertices in a weighted graph, where all the edge weights are non-negative. It uses a greedy approach and is widely applied in routing and pathfinding problems, like in GPS systems.

3. **Minimum Spanning Tree (MST)**: An MST is a subset of the edges of a graph that connects all the vertices with the minimum possible total edge weight. Two well-known algorithms to find MSTs are Kruskal's Algorithm and Prim's Algorithm. This problem is useful in network design, where the goal is to minimize the total length of cables required to connect all computers in a network.

4. **Floyd-Warshall Algorithm**: This algorithm is used for finding the shortest paths between all pairs of vertices in a graph. Unlike Dijkstra's, which only finds the shortest path from a single source, Floyd-Warshall calculates the shortest path between every pair of nodes. It works even for graphs with negative edge weights (but no negative cycles).

5. **Bellman-Ford Algorithm**: Like Dijkstra's algorithm, Bellman-Ford is also used for finding the shortest paths from a single source. The major difference is that Bellman-Ford works even when some of the edge weights are negative. However, it is slower than Dijkstra's and is commonly used when negative edge weights are present.

6. **Bipartite Graphs**: A **bipartite** graph is a graph whose vertices can be divided into two disjoint sets such that no two vertices within the same set are adjacent. This concept is widely used in matching problems, where you need to match elements from two different sets, such as job applicants to jobs, or students to projects.

7. **Graph Coloring**: Graph coloring is the process of assigning colors to the vertices of a graph such that no two adjacent vertices share the same color. It is a well-known problem in scheduling and register allocation in compilers.

Graphs are an essential topic in computer science, offering numerous applications in both theoretical and practical domains. Mastering the various algorithms associated with graphs can open up new problem-solving techniques for a wide range of computational challenges. Understanding graph representations, traversal techniques, and advanced algorithms will equip you with the skills to handle complex problems efficiently.

7 A Quick Overview of C++ Language

C++ is largely compatible with C, meaning that most valid C code can be compiled with a C++ compiler. However, there are some differences in syntax and semantics between the two languages. Essentially, teaching C++ after C should be a quick overview of syntax differences for the same examples we had in C. Then, we'll have some examples for C++ features which C doesn't have those.

Here's a simple C++ code that prints "Hello, McMaster!" to the console:

```
#include <iostream>

int main() {
 // Print "Hello, McMaster!" to the console
 std::cout << "Hello, McMaster!" << std::endl;
 return 0;
}
```

To compile and run this code:

1. Save the code in a file with a `.cpp` extension, for example, `hello.cpp`.
2. Navigate to the directory where you saved the file.
3. Compile the code using a C++ compiler. For example, if you have `g++`, you can compile it with `g++ hello.cpp -o hello`.
4. Run the executable with `./hello`.

There are several compilers available for C++. Depending on your specific needs and platform, you may choose one of these compilers or explore other options. Here we only use **GNU** Compiler Collection. `gcc` and `g++` are compiler-drivers of the GNU. Using `g++ --version` gives me the version of `g++` currently is working on my OS. If there is nothing in the terminal, then you need to install it using `sudo apt install g++` and `sudo apt install build-essential` if needed.

`<iostream>` in C++, the same as `<stdio.h` in C, provides functionality for input from the standard input device (usually the keyboard) and output to the standard output device (usually the console). It contains declarations for objects like `std::cout`, `std::cin`, `std::endl`, etc., which are part of the C++ Standard Library's `iostream namespace`.

In C++, input and output operations are encapsulated within the `iostream namespace`. In C, the functions are not encapsulated within a `namespace`.

C++ introduces the concept of **objects** like `std::cout` and `std::cin` for output and input respectively. C, on the other hand, primarily relies on functions like `printf` and `scanf`.

C++ `iostream` offers **type-safety** due to its use of overloaded **operators**, like `<<`. C's `stdio.h` relies on format specifiers in functions like `printf` and `scanf`, which can lead to undefined behavior if used incorrectly.

C and C++ are related programming languages with some key differences. Here's a summary of the main distinctions between the two and also commonly features used:

1. **Classes and Objects:**
   - C does not support classes or objects. It relies on functions and data structures for organizing code.
   - C++ introduces the concept of classes and objects. Classes are user-defined data types that encapsulate data and functions that operate on the data. Objects are instances of classes.
2. **Function Overloading and Default Arguments:**
   - C does not support function overloading or default arguments.
   - C++ supports function overloading, which allows multiple functions with the same name but different parameter lists. It also supports default arguments, allowing function parameters to have default values.
3. **Templates and Generics:**
   - C does not support templates or generics.
   - C++ supports templates, which allow generic programming. Templates enable the creation of functions and classes that operate on different data types without code duplication.
4. **Memory Management:**
   - In C, memory management is done manually using functions like `malloc()` and `free()`.
   - C++ introduces the concept of constructors and destructors for automatic memory management using the `new` and `delete` operators. Additionally, C++ provides smart pointers (`std::unique_ptr`, `std::shared_ptr`, etc.) for safer and more efficient memory management.

In summary, while C and C++ share some similarities, C++ is a more feature-rich and complex language, designed to support a wider range of programming paradigms and provide greater abstraction and flexibility.

7.1 The Same C Codes on C++

The inclusion of `<limits.h>` is traditionally associated with C programming, but it's also valid in C++. Look at the following example:

```cpp
#include <iostream>
#include <limits.h>

int main() {

  std::cout << "Size of char: " << 8 * sizeof(char) << " bits\n";
  std::cout << "Signed char range: " << SCHAR_MIN << " to " << SCHAR_MAX << "\n";
  std::cout << "Unsigned char range: " << 0 << " to " << UCHAR_MAX << "\n\n";

  std::cout << "Size of int: " << 8 * sizeof(int) << " bits\n";
  std::cout << "Signed int range: " << INT_MIN << " to " << INT_MAX << "\n";
  std::cout << "Unsigned int range: " << 0 << " to " << UINT_MAX << "\n\n";

  std::cout << "Size of short: " << 8 * sizeof(short) << " bits\n";
  std::cout << "Signed short range: " << SHRT_MIN << " to " << SHRT_MAX << "\n";
  std::cout << "Unsigned short range: " << 0 << " to " << USHRT_MAX << "\n\n";

  std::cout << "Size of long: " << 8 * sizeof(long) << " bits\n";
  std::cout << "Signed long range: " << LONG_MIN << " to " << LONG_MAX << "\n";
  std::cout << "Unsigned long range: " << 0 << " to " << ULONG_MAX << "\n\n";

  std::cout << "Size of long long: " << 8 * sizeof(long long) << " bits\n";
  std::cout << "Signed long long range: " << LLONG_MIN << " to " << LLONG_MAX << "
      \n";
  std::cout << "Unsigned long long range: " << 0 << " to " << ULLONG_MAX << "\n";

  return 0;
}
```

In C++, the equivalent header is `<climits>`. `<climits>` is the C++ version of `<limits.h>`, and it defines the same constants as `<limits.h>`, but within the `std namespace` (e.g., `std::INT_MIN`, s`td::INT_MAX`, etc.).

However, `<limits.h>` is still valid in C++, and many C++ compilers provide it for compatibility with C code. When included in a C++ program, it exposes the same constants as in C. I can write the code with `<climits>`:

```cpp
#include <iostream>
#include <climits>

using namespace std;
```

```
int main() {

  cout << "Size of char: " << CHAR_BIT << " bits\n";
  cout << "Signed char range: " << SCHAR_MIN << " to " << SCHAR_MAX << "\n";
  cout << "Unsigned char range: " << 0 << " to " << UCHAR_MAX << "\n\n";

  cout << "Size of int: " << sizeof(int) * CHAR_BIT << " bits\n";
  cout << "Signed int range: " << INT_MIN << " to " << INT_MAX << "\n";
  cout << "Unsigned int range: " << 0 << " to " << UINT_MAX << "\n\n";

  cout << "Size of short: " << sizeof(short) * CHAR_BIT << " bits\n";
  cout << "Signed short range: " << SHRT_MIN << " to " << SHRT_MAX << "\n";
  cout << "Unsigned short range: " << 0 << " to " << USHRT_MAX << "\n\n";

  cout << "Size of long: " << sizeof(long) * CHAR_BIT << " bits\n";
  cout << "Signed long range: " << LONG_MIN << " to " << LONG_MAX << "\n";
  cout << "Unsigned long range: " << 0 << " to " << ULONG_MAX << "\n\n";

  cout << "Size of long long: " << sizeof(long long) * CHAR_BIT << " bits\n";
  cout << "Signed long long range: " << LLONG_MIN << " to " << LLONG_MAX << "\n";
  cout << "Unsigned long long range: " << 0 << " to " << ULLONG_MAX << "\n";

  return 0;
}
```

In this version, we use `#include <cstdint>` to include the C++ standard library header that provides fixed-width integer types (`int8_t`, `uint8_t`, etc.). We also directly use the constants provided by the `<cstdint>` header for the ranges.

```
#include <iostream>
#include <cstdint>

using namespace std;

int main() {
  cout << "Size of int8_t: " << 8 * sizeof(int8_t) << " bits\n";
  cout << "Signed int8_t range: " << INT8_MIN << " to " << INT8_MAX << "\n\n";

  cout << "Size of uint8_t: " << 8 * sizeof(uint8_t) << " bits\n";
  cout << "uint8_t range: " << 0 << " to " << UINT8_MAX << "\n\n";

  cout << "Size of int16_t: " << 8 * sizeof(int16_t) << " bits\n";
  cout << "Signed int16_t range: " << INT16_MIN << " to " << INT16_MAX << "\n\n";

  cout << "Size of uint16_t: " << 8 * sizeof(uint16_t) << " bits\n";
  cout << "uint16_t range: " << 0 << " to " << UINT16_MAX << "\n\n";
```

```
    cout << "Size of int32_t: " << 8 * sizeof(int32_t) << " bits\n";
    cout << "Signed int32_t range: " << INT32_MIN << " to " << INT32_MAX << "\n\n";

    cout << "Size of uint32_t: " << 8 * sizeof(uint32_t) << " bits\n";
    cout << "uint32_t range: " << 0 << " to " << UINT32_MAX << "\n\n";

    cout << "Size of int64_t: " << 8 * sizeof(int64_t) << " bits\n";
    cout << "Signed int64_t range: " << INT64_MIN << " to " << INT64_MAX << "\n\n";

    cout << "Size of uint64_t: " << 8 * sizeof(uint64_t) << " bits\n";
    cout << "uint64_t range: " << 0 << " to " << UINT64_MAX << "\n";

    return 0;
}
```

In this version, we should replace `<float.h>` with `<limits>`, which is part of the C++ standard library. We use `std::numeric_limits` to get information about the precision and minimum/maximum values of floating-point types. Additionally, `std::scientific` manipulator is used to format the output in scientific notation.

```
#include <iostream>
#include <limits>

using namespace std;

int main() {
  cout << "Precision:\n";
  cout << "Float: " << numeric_limits<float>::digits10 << " digits\n";
  cout << "Double: " << numeric_limits<double>::digits10 << " digits\n";
  cout << "Long Double: " << numeric_limits<long double>::digits10 << " digits\n\n";

  cout << "Minimum and Maximum Values:\n";
  cout << "Float: Minimum: " << scientific << numeric_limits<float>::min()
  << ", Maximum: " << scientific << numeric_limits<float>::max() << "\n";
  cout << "Double: Minimum: " << scientific << numeric_limits<double>::min()
  << ", Maximum: " << scientific << numeric_limits<double>::max() << "\n";
  cout << "Long Double: Minimum: " << scientific << numeric_limits<long double>::min()
  << ", Maximum: " << scientific << numeric_limits<long double>::max() << "\n";

  return 0;
}
```

`std::numeric_limits<float>::digits10` is a constant defined in the C++ standard library that represents the number of base-10 digits that can be accurately represented by the `float` type.

More specifically, `digits10` gives you the number of decimal digits that can be represented without loss of precision when converting from decimal to binary floating-point format and back. It essentially provides an indication of the precision of the floating-point type.

For example, if `std::numeric_limits<float>::digits10` returns 6, it means that `float` can accurately represent approximately 6 decimal digits. Any number that requires more than 6 digits of precision might lose some precision when stored in a `float`.

Let's create a factorial function and split the code into separate `.cpp` and `.hpp` files:

- `main.cpp`:

```
#include <iostream>
#include "factorial.hpp"

using namespace std;

int main() {
 unsigned int n;
 cout << "Enter a non-negative integer: ";
 cin >> n;

 unsigned long long result = factorial(n);

 cout << "Factorial of " << n << " is: " << result << endl;

 return 0;
}
```

- `factorial.hpp`:

```
#ifndef FACTORIAL_HPP
#define FACTORIAL_HPP

unsigned long long factorial(unsigned int n);

#endif
```

- `factorial.cpp`

```
#include "factorial.hpp"

unsigned long long factorial(unsigned int n) {
 if (n == 0 || n == 1) {
  return 1;
 }
 return n * factorial(n - 1);
}
```

And compile the code with `g++ -o factorial factorial.cpp main.cpp`.

Here’s a simple example demonstrating the usage of pointers in C++:

```
#include <iostream>

using namespace std;

int main() {
 // Declare an integer variable
 int number = 10;
 // Declare a pointer variable and assign the address of ’number’
 int* ptr = &number;

 // Print the value of ’number’
 cout << "Value of number: " << number << endl;
 // Print the address of ’number’
 cout << "Address of number: " << &number << endl;
 // Print the value of ’ptr’ (address of ’number’)
 cout << "Value of ptr: " << ptr << endl;
 // Print the value pointed to by ’ptr’
 cout << "Value pointed by ptr: " << *ptr << endl;

 // Modify the value of ’number’ indirectly through ’ptr’
 *ptr = 20;
 cout << "Modified value of number: " << number << endl;

 return 0;
}
```

And it gives me:

```
Value of number: 10
Address of number: 0x7ffe89eacebc
Value of ptr: 0x7ffe89eacebc
Value pointed by ptr: 10
Modified value of number: 20
```

About dynamic memory allocation in C++, there are a lot of things we can discuss and it will take a lot of time. The best place to start with memory allocation in C++ is here. Here's an example demonstrating dynamic memory allocation for an array in C++:

```
#include <iostream>

int main() {
  int size;
  std::cout << "Enter the size of the array: ";
  std::cin >> size;

  // Dynamically allocate memory for an integer array of 'size' elements
  int* arr = new int[size];

  // Initialize the array elements
  for (int i = 0; i < size; ++i) {
    arr[i] = i * 10;  // Assigning a value based on index for demonstration
  }

  // Print the array elements
  std::cout << "Array elements: ";
  for (int i = 0; i < size; ++i) {
    std::cout << arr[i] << " ";
  }
  std::cout << std::endl;

  // Deallocate the dynamically allocated memory
  delete[] arr;

  return 0;
}
```

It's important to note that when using dynamic memory allocation, you should always deallocate the memory using `delete[]` to avoid memory leaks. Failure to do so can lead to memory exhaustion and unexpected behavior not just in your program but also in your machine. We can do the same for a 2D array:

```
#include <iostream>

int main() {
  int numRows = 3;
  int numCols = 4;

  // Step 1: Declare a pointer to a pointer
  int** array;

  // Step 2: Allocate memory for rows
```

```
 array = new int*[numRows];

 // Step 3: Allocate memory for columns in each row
 for (int i = 0; i < numRows; ++i) {
  array[i] = new int[numCols];
 }

 // Step 4: Assign values to the array
 for (int i = 0; i < numRows; ++i) {
  for (int j = 0; j < numCols; ++j) {
   array[i][j] = i * numCols + j; // Example value assignment
  }
 }

 // Step 4: Print the array
 for (int i = 0; i < numRows; ++i) {
  for (int j = 0; j < numCols; ++j) {
   std::cout << array[i][j] << " ";
  }
  std::cout << std::endl;
 }

 // Step 5: Deallocate the memory
 for (int i = 0; i < numRows; ++i) {
  delete[] array[i];
 }
 delete[] array;

 return 0;
}
```

7.2 Function Overloading

Function overloading allows you to define multiple functions with the same name but different parameter lists. Here's an example demonstrating function overloading in C++:

```
#include <iostream>

// Function to calculate the area of a rectangle
int area(int length, int width) {
 return length * width;
}

// Overloaded function to calculate the area of a circle
double area(double radius) {
 return 3.14 * radius * radius;
}

int main() {
 int length = 5;
 int width = 3;
 double radius = 2.5;

 // Call the area function for rectangle
 std::cout << "Area of rectangle: " << area(length, width) << std::endl;

 // Call the area function for circle
 std::cout << "Area of circle: " << area(radius) << std::endl;

 return 0;
}
```

7.3 `constexpr`

In C++, `constexpr` is a keyword that declares that an object or function can be evaluated at compile time. This allows for performance improvements and enables certain use cases where compile-time evaluation is desirable. Here's a simple example demonstrating the usage of `constexpr`:

```
#include <iostream>

using namespace std;

// Define a constexpr function to compute the factorial of a non-negative integer
constexpr unsigned long long factorial(unsigned int n) {
 return n <= 1 ? 1 : n * factorial(n - 1);
}

int main() {
 constexpr unsigned int num = 5;

 // Call the constexpr function at compile time
 constexpr unsigned long long result = factorial(num);

 // Print the result at compile time
 cout << num << "! (computed at compile time): " << result << endl;

 return 0;
}
```

7.4 Range-based for Loops

Range-based for loops provide a more concise syntax for iterating over elements of a container or a range. They eliminate the need for explicit iterators and simplify the loop syntax. They work with arrays, initializer lists, and user-defined types that provide the necessary interface. Here's an example demonstrating the use of range-based for loops and references:

```cpp
#include <iostream>

using namespace std;

int main() {
  // Example 1: Range-based for loop with array
  int arr[] = {1, 2, 3, 4, 5};

  cout << "Example 1: Range-based for loop with array:\n";
  for (int& num : arr) {
   num *= 2; // Multiply each element of the array by 2
   cout << num << " ";
  }
  cout << endl;

  // Example 2: Range-based for loop with initializer list
  cout << "\nExample 2: Range-based for loop with initializer list:\n";
  for (const auto& elem : {10, 20, 30, 40, 50}) {
   cout << elem << " ";
  }
  cout << endl;

  return 0;
}
```

In this code, `auto` is used in conjunction with the range-based for loop. The `elem` variable is declared using `auto`, which means its type will be automatically deduced by the compiler based on the type of the elements in the range `{10, 20, 30, 40, 50}`.

7.5 Vectors

In C++, vectors are a type of dynamically resizable array provided by the Standard Template Library (STL). They provide similar functionality to arrays but with additional features such as automatic resizing, bounds checking, and various member functions for easy manipulation of elements. Here's an example demonstrating the usage of vectors in C++:

```cpp
#include <iostream>
```

```cpp
#include <vector>

using namespace std;

int main() {
 // Declare and initialize a vector of integers
 vector<int> numbers = {1, 2, 3, 4, 5};

 // Add elements to the vector using push_back
 numbers.push_back(6);
 numbers.push_back(7);

 // Access elements of the vector using indexing
 cout << "Vector elements:";
 for (size_t i = 0; i < numbers.size(); ++i) {
  cout << " " << numbers[i];
 }
 cout << endl;

 // Modify an element of the vector
 numbers[2] = 10;

 // Iterate over the vector using range-based for loop
 cout << "Modified vector elements:";
 for (const auto& num : numbers) {
  cout << " " << num;
 }
 cout << endl;

 // Access the first and last element of the vector
 cout << "First element: " << numbers.front() << endl;
 cout << "Last element: " << numbers.back() << endl;

 // Check if the vector is empty
 if (numbers.empty()) {
  cout << "Vector is empty" << endl;
 } else {
  cout << "Vector is not empty" << endl;
 }

 // Clear the vector
 numbers.clear();

 // Check if the vector is empty after clearing
 if (numbers.empty()) {
  cout << "Vector is empty after clearing" << endl;
 } else {
      cout << "Vector is not empty after clearing" << endl;
    }

    return 0;
  }
```

We include the `<vector>` header to use vectors, and declare a vector named `numbers` and initialize it with some values. We use `push_back` to add elements to the end of the vector. Then, we access elements of the vector using indexing (`[]`), and we modify an element using indexing as well. We iterate over the vector using a range-based for loop. We use `front()` and `back()` member functions to access the first and last elements of the vector. We use `empty()` member function to check if the vector is empty, and `clear()` member function to remove all elements from the vector. This is what we get:

```
Vector elements: 1 2 3 4 5 6 7
Modified vector elements: 1 2 10 4 5 6 7
First element: 1
Last element: 7
Vector is not empty
Vector is empty after clearing
```

Do you think the `vector` named `numbers` is saved on stack or heap??

7.6 Classes

In C++, a class is a user-defined type that serves as a blueprint for creating objects. It encapsulates data for the object (called member variables) and functions to manipulate that data (called member functions). Classes provide a way to model real-world entities and define their behavior and properties. Here's a general format of a class in C++:

```
class ClassName {
 private:
 // Private member variables
 DataType memberVariable1;
 DataType memberVariable2;
 // ...

 public:
 // Constructors
 ClassName(); // Default constructor
 ClassName(DataType arg1, DataType arg2); // Parameterized constructor
 // Member functions
 ReturnType memberFunction1(DataType arg1);
 ReturnType memberFunction2(DataType arg2);
 // ...
};
```

Key components of a class in C++:

- **Private data members**: These are the member variables of the class that are accessible only within the class itself. They encapsulate the internal state of the object and provide data hiding to ensure encapsulation. In another word, we don't trust the user giving use the correct format of inputs and we don't let the user to directly have access to it. We will have examples of all these features.
- **Public member functions**: These are the member functions of the class that are accessible outside the class. They define the behavior of the object and provide an interface for interacting with the object's data. Public member functions can be called by code outside the class.
- **Constructors**: These are special member functions that are called when an object of the class is created. Constructors initialize the object's data members and perform any necessary setup. They have the same name as the class and do not have a return type.
- **Member functions**: These are functions defined within the class that operate on the object's data. They can access the object's private data members and perform operations on them. Member functions provide the behavior and functionality of the class.

Essentially, we can say `struct` and `class` work the same way. However, there is one main difference between structs and classes in C++: By default, member variables and member functions in structs are public, while in classes they are private.

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

// Define a struct to represent a point in 2D space
struct Point {
 double x;
 double y;
};
```

```
int main() {
 // Create an instance of the Point struct
 Point p1;
 p1.x = 3.0, p1.y = 4.0;
 // p1 = {3.0,4.0}; // Or I can use this one

 // Print the coordinates of the point
 cout << "struct: (" << p1.x << ", " << p1.y << ")" << endl;
}
```

Now, let's rewrite the same code using a class:

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

class Point
{
 public:
 // Member functions to print x and y coordinates
 void print() const
 {
  cout << "(" << x << ", " << y << ")" << endl;
 }

 double x = 0, y = 0;
};

int main()
{

 Point p1 = {3.0, 4.0};
 p1.print(); // prints (3, 4)

 p1.x=12.7;
 p1.y=-3.2;
 p1.print(); // prints (12.7, -3.2)
}
```

7.6.1 Private and Public Member

If `double x, y;` are kept as public members, it means they can be accessed and modified directly from outside the class. This breaks the principle of encapsulation, which aims to hide the internal state of an object and restrict access to it. For example, let's define the previous `Point` class as:

```
class Point
{
 public:
 // Member functions to print x and y coordinates
 void print() const
 {
  cout << "(" << x << ", " << y << ")" << endl;
 }

 private:
 double x = 0, y = 0;
};
```

In this case, changing the values in members using `Point p1 = 3.0, 4.0`, `p1.x` or `p1.y` will give us error. Members of the `Point` class are `private`, which means they cannot be directly accessed or modified from outside the class. However, you can provide member functions within the class to initialize or modify these private members. Here's how you can modify the Point class to include setter functions for `x` and `y`:

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

class Point
{
 public:
 void print() const
 {
  cout << "(" << x << ", " << y << ")" << endl;
 }

 void setX(double newX)
 {
  x = newX;
 }
 void setY(double newY)
 {
  y = newY;
 }

 private:
 double x = 0, y = 0;
};

int main()
{
 Point p1;
 p1.print(); // prints (0, 0)

 p1.setX(3.5);
 p1.setY(4.2);

 p1.print(); // prints (3.5, 4.2)

}
```

(in progress) (more examples)

7.6.2 Constructor

In C++, a constructor is a special member function of a class that is automatically called when an object of the class is created. It is used to initialize the object's data members and perform any necessary setup. Constructors have the same name as the class and do not have a return type, not even void.

Constructors can be overloaded, meaning a class can have multiple constructors with different parameter lists. This allows objects to be initialized in different ways depending on the arguments passed to the constructor.

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

class Point
{
 public:
```

```
 // Default constructor
 Point() : x(0), y(0) {}

 // Constructor with 1 arg
 Point(double xyVal) : x(xyVal), y(xyVal) {}
 // Constructor with 2 args
 Point(double xVal, double yVal) : x(xVal), y(yVal) {}

 void print() const
 {
  cout << "(" << x << ", " << y << ")" << endl;
 }

 private:
 double x=0, y=0;
};

int main()
{

 Point p1; //calls default constructor
 p1.print(); // prints (0, 0)

 Point p2(3.5); // calls Constructor with 1 arg
 p2.print(); // prints (3.5, 3.5)

 Point p3(1.5, 2.5);
 p3.print(); // prints (1.5, 2.5)

 Point p4 = {-7.5, -8.8};
 p4.print(); // prints (-7.5, -8.8)
}
```

Mind you that `p4 = -7.5, -8.8;` is calling the constructors with 2 input arguments. If we remove the constructor, will get the error `<brace-enclosed initializer list>`. Since `x` and `y` are also private members they cannot be accessed with `p4.x` or `p4.y` to ensure the encapsulation.

But why we need to protect the data? By making `x` and `y` private, you prevent direct access to these members from outside the class, which helps maintain the integrity of the class's data and prevents unintended modifications. Let's say `x` and `y` are the length of something. By enforcing validation checks in the constructors to ensure that x and y are not negative, you can guarantee that objects of the Point class are always initialized with valid data. This prevents the creation of invalid objects and helps ensure the correctness of the program's behavior. Here's the modified code with `x` and `y`as private members and validation checks in the constructors:

```cpp
#include <iostream>
#include <stdexcept> // for std::invalid_argument
using namespace std;

class Point {
 private:
 double x, y;

 public:
 // Default constructor
 Point() : x(0), y(0) {}

 // Constructor with 1 arg
 Point(double xyVal) {
  if (xyVal < 0) {
   throw std::invalid_argument("Coordinates cannot be negative");
  }
  x = xyVal;
  y = xyVal;
 }

 // Constructor with 2 args
 Point(double xVal, double yVal) {
  if (xVal < 0 || yVal < 0) {
   throw std::invalid_argument("Coordinates cannot be negative");
  }
  x = xVal;
  y = yVal;
 }

 // Member function to print x and y coordinates
 void print() const {
  cout << "(" << x << ", " << y << ")" << endl;
 }
};

int main() {
 try {
  Point p1; // Default constructor, valid
  p1.print(); // prints (0, 0)

  Point p2(3.5); // Constructor with 1 arg, valid
  p2.print(); // prints (3.5, 3.5)

  Point p3(1.5, 2.5); // Constructor with 2 args, valid
  p3.print(); // prints (1.5, 2.5)

  Point p4 = {7.5, 8.8}; // uniform init (2 args), valid
  p4.print(); // prints (7.5, 8.8)

  // Uncomment the following lines to test with negatives
  // Point p5(-2.0); // Constructor with 1 arg, invalid
  // p5.print();
 }
 catch (const std::invalid_argument& e) {
  std::cerr << "Error: " << e.what() << std::endl;
 }
}
```

We can separating member functions from the class to improve code readability by reducing the size and complexity of the class declaration. With the member function definitions moved outside the class, the class declaration becomes more concise and focused, making it easier to read and understand. Here's an example of separating member functions from the class in C++:

```
#include <iostream>

class Calculator
{
 public:
 // Constructor
 Calculator(double initialValue) : result(initialValue) {}

 void add(double value);

 void subtract(double value);
 void multiply(double value);

 void divide(double value);

 void printResult() const;

 private:
 double result;
};

void Calculator::add(double value)
{
 result += value;
}

void Calculator::subtract(double value)
{
 result -= value;
}

void Calculator::multiply(double value)
{
 result *= value;
}

void Calculator::divide(double value)
{
 if (value != 0)
 {
  result /= value;
 }
 else
 {
  std::cerr << "Error: Division by zero" << std::endl;
 }
}

void Calculator::printResult() const
{
 std::cout << "Result: " << result << std::endl;
}

int main()
{
 Calculator calc(0);

 calc.add(5);        // Add 5 to the result
 calc.printResult(); // prints ”Result: 5”

 calc.subtract(2);   // Subtract 2 from the result
 calc.printResult(); // prints ”Result: 3”

 calc.multiply(4);   // Multiply the result by 4
 calc.printResult(); // prints ”Result: 12”

 calc.divide(3);     // Divide the result by 3
 calc.printResult(); // prints ”Result: 4”

 calc.divide(0); // Attempt to divide by zero
 // Error message ”Error: Division by zero” will be printed to stderr
}
```

7.7 try-throw-catch

`try`, `throw`, and `catch` are keywords used for exception handling. Exception handling is a mechanism for dealing with runtime errors or exceptional conditions that occur during program execution. It allows you to separate error handling logic from normal program flow, improving code clarity and maintainability. Here's a simple example demonstrating the use of try, throw, and catch:

```
#include <iostream>
#include <stdexcept> // for std::runtime_error
using namespace std;

// Function to divide two numbers
double divide(double dividend, double divisor) {
 if (divisor == 0) {
  throw std::runtime_error("Division by zero error");
 }
 return dividend / divisor;
}

int main() {
 double result;
 double dividend = 10.0;
 double divisor = 0.0;

 try {
  // Attempt to divide two numbers
  result = divide(dividend, divisor);
  cout << "Result of division: " << result << endl;
 } catch (const std::runtime_error& e) {
  // Handle division by zero error
  cerr << "Error: " << e.what() << endl;
 }
}
```

The `divide()` function attempts to divide two numbers (`dividend` by `divisor`). If the divisor is 0, it throws a `std::runtime_error` with the message `"Division by zero error"`.

In the `main()` function, we use a try block to enclose the call to the `divide()` function. If an exception is thrown within the try block, the program will attempt to find a matching catch block to handle the exception. We have a catch block that catches `std::runtime_error` exceptions and prints an error message to `cerr` (standard error stream). This block handles the division by zero error gracefully.

7.8 Templates

Templates in C++ are a powerful feature that allows functions and classes to operate with generic types. This enables code reuse and type safety while maintaining performance. Templates are a cornerstone of generic programming in C++ and are used extensively in the Standard Template Library (STL).

7.8.1 Function Templates

Function templates allow you to create a single function definition that works with any data type.

- Example: Swap Function

```
#include <iostream>

template <typename T>
void swap(T& a, T& b) {
 T temp = a;
 a = b;
 b = temp;
}
```

```
int main() {
 int x = 5, y = 10;
 swap(x, y);
 std::cout << "Swapped values: x = " << x << ", y = " << y << std
    ::endl;

 double a = 1.1, b = 2.2;
 swap(a, b);
 std::cout << "Swapped values: a = " << a << ", b = " << b << std
    ::endl;

 return 0;
}
```

7.8.2 Class Templates

Class templates allow you to create a class that can handle any data type.

- Example: Simple Generic Stack

```
#include <iostream>

template <typename T>
class Stack {
 private:
 T* arr;
 int top;
 int capacity;

 public:
 Stack(int size = 100) {
  arr = new T[size];
  capacity = size;
  top = -1;
 }

 ~Stack() {
  delete[] arr;
 }
 void push(T value) {
  if (top == capacity - 1) {
   std::cout << "Stack Overflow\n";
   return;
  }
  arr[++top] = value;
 }

 T pop() {
  if (top == -1) {
   std::cout << "Stack Underflow\n";
   exit(EXIT_FAILURE);
  }
  return arr[top--];
 }

 T peek() {
  if (top == -1) {
   std::cout << "Stack is empty\n";
   exit(EXIT_FAILURE);
  }
  return arr[top];
 }

 int size() {
  return top + 1;
 }

 bool isEmpty() {
  return top == -1;
 }
};

int main() {
 Stack<int> intStack;

 intStack.push(1);
 intStack.push(2);
 intStack.push(3);

 std::cout << "Top element is " << intStack.peek() << std::endl;

 intStack.pop();
 intStack.pop();
 intStack.pop();

 std::cout << "Stack size is " << intStack.size() << std::endl;

 Stack<std::string> stringStack;

 stringStack.push("Hello");
 stringStack.push("World");

 std::cout << "Top element is " << stringStack.peek() << std::
    endl;

 return 0;
}
```

7.8.3 Template Specialization

Sometimes, you need to handle specific types differently. Template specialization allows you to provide a specialized implementation for a specific data type.

- Example: Template Specialization for `char*`

```
#include <iostream>
#include <cstring>

template <typename T>
class Compare {
 public:
 bool isEqual(T a, T b) {
  return a == b;
 }
};
```

```cpp
template <>
class Compare<char*> {
 public:
 bool isEqual(char* a, char* b) {
  return strcmp(a, b) == 0;
 }
};

int main() {
 Compare<int> intCompare;
 std::cout << intCompare.isEqual(1, 1) << std::endl; // 1 (true)
 std::cout << intCompare.isEqual(1, 2) << std::endl; // 0 (false)

 Compare<char*> strCompare;
 std::cout << strCompare.isEqual("hello", "hello") << std::endl;
    // 1 (true)
 std::cout << strCompare.isEqual("hello", "world") << std::endl;
    // 0 (false)

 return 0;
}
```

7.8.4 Template Member Functions

You can define member functions as templates within a non-template class.

- Example: Template Member Function

```cpp
#include <iostream>

class Printer {
 public:
 template <typename T>
 void print(T value) {
  std::cout << value << std::endl;
 }
};

int main() {
 Printer printer;
 printer.print(123);
 printer.print(4.56);
 printer.print("Hello, World!");

 return 0;
}
```

7.8.5 Non-Type Template Parameters

Templates can also accept non-type parameters, such as integers or pointers.

- Example: Array Class with Non-Type Parameter

```
#include <iostream>

template <typename T, int size>
class Array {
 private:
 T arr[size];

 public:
 T& operator[](int index) {
  if (index >= size) {
   std::cout << "Index out of bounds\n";
   exit(EXIT_FAILURE);
  }
  return arr[index];
 }

 int getSize() {
  return size;
 }
};

int main() {
 Array<int, 10> intArray;

 for (int i = 0; i < intArray.getSize(); ++i) {
  intArray[i] = i * 2;
 }

 for (int i = 0; i < intArray.getSize(); ++i) {
  std::cout << intArray[i] << " ";
 }
 std::cout << std::endl;

 return 0;
}
```

7.8.6 Variadic Templates

Variadic templates allow you to create functions or classes that take a variable number of template arguments.

- Example: Variadic Function Template for Sum

```
#include <iostream>

template <typename T>
T sum(T value) {
 return value;
}

template <typename T, typename... Args>
T sum(T first, Args... args) {
 return first + sum(args...);
}

int main() {
 std::cout << sum(1, 2, 3, 4, 5) << std::endl; // Output: 15
 std::cout << sum(1.1, 2.2, 3.3) << std::endl; // Output: 6.6

 return 0;
}
```

Templates are a key feature in C++ that enable generic programming, making code more flexible, reusable, and type-safe. By leveraging templates, developers can create highly efficient and maintainable code that can handle a wide variety of data types and scenarios.

7.9 Operator Overloading

Operator overloading in C++ allows you to define how operators work with user-defined types (`classes` or `structs`). This makes your custom types behave more like built-in types, enhancing readability and usability.

7.9.1 Basics of Operator Overloading

Operators that can be overloaded include:

- Arithmetic operators: `+`, `-`, `*`, `/`, `%`
- Relational operators: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Logical operators: `&&`, `||`, `!`
- Increment/decrement operators: `++`, `--`
- Bitwise operators: `&`, `|`, `^`, `~`, `<<`, `>>`
- Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`
- Subscript operator: `[]`
- Function call operator: `()`
- Dereference operator: `*`
- Member access operators: `->`, `.`

Some operators cannot be overloaded, such as `::`, `.:`, `.*`, `?:`, and `sizeof`.

Operator overloading is done by defining a function with the keyword operator followed by the operator to be overloaded. These functions can be member functions or non-member functions.

- Example 1: Overloading the `+` Operator for a `Complex` Class

```
#include <iostream>

class Complex {
 private:
```

```
 double real;
 double imag;

 public:
 Complex(double r = 0, double i = 0) : real(r), imag(i) {}

 // Overload + operator
 Complex operator+(const Complex& other) {
  return Complex(real + other.real, imag + other.imag);
 }

 void print() const {
  std::cout << "(" << real << ", " << imag << "i)" << std::endl;
 }
};

int main() {
 Complex c1(1.0, 2.0), c2(2.5, 3.5);
 Complex c3 = c1 + c2;
 c3.print(); // Output: (3.5, 5.5i)

 return 0;
}
```

- Example 2: Overloading the `<<` Operator for Output

Overloading the `<<` operator allows custom types to be output using `std::cout`.

```
#include <iostream>

class Point {
 private:
 int x, y;

 public:
 Point(int x = 0, int y = 0) : x(x), y(y) {}

 // Overload << operator
 friend std::ostream& operator<<(std::ostream& os, const Point&
    pt);
};

std::ostream& operator<<(std::ostream& os, const Point& pt) {
 os << "(" << pt.x << ", " << pt.y << ")";
 return os;
}

int main() {
 Point p1(3, 4);
 std::cout << p1 << std::endl; // Output: (3, 4)

 return 0;
}
```

- Example 3: Overloading the `[]` Operator for Array-Like Access

```
#include <iostream>

class IntArray {
 private:
 int* arr;
 int size;

 public:
 IntArray(int s) : size(s) {
  arr = new int[size];
 }

 ~IntArray() {
  delete[] arr;
 }

 // Overload [] operator
 int& operator[](int index) {
  if (index >= 0 && index < size) {
   return arr[index];
  } else {
   std::cout << "Index out of bounds" << std::endl;
   exit(EXIT_FAILURE);
  }
 }
};

int main() {
 IntArray array(10);
 array[0] = 5;
 array[1] = 10;

 std::cout << "array[0] = " << array[0] << std::endl; // Output: array[0] = 5
 std::cout << "array[1] = " << array[1] << std::endl; // Output: array[1] = 10

 return 0;
}
```

- Example 4: Overloading the `++` Operator (Prefix and Postfix)

```
#include <iostream>

class Counter {
 private:
 int count;

 public:
 Counter() : count(0) {}

 // Overload prefix ++ operator
 Counter& operator++() {
  ++count;
  return *this;
 }

 // Overload postfix ++ operator
 Counter operator++(int) {
  Counter temp = *this;
  ++count;
  return temp;
 }
 int getCount() const {
  return count;
 }
};

int main() {
 Counter counter;

 ++counter;
 std::cout << "Count after prefix increment: " << counter.
    getCount() << std::endl; // Output: 1

 counter++;
 std::cout << "Count after postfix increment: " << counter.
    getCount() << std::endl; // Output: 2

 return 0;
}
```

- Example 5: Overloading the `==` and `!=` Operators

```
#include <iostream>

class Box {
 private:
 int length, width, height;

 public:
 Box(int l = 0, int w = 0, int h = 0) : length(l), width(w),
    height(h) {}

 // Overload == operator
 bool operator==(const Box& other) const {
  return length == other.length && width == other.width && height
      == other.height;
 }

 // Overload != operator
 bool operator!=(const Box& other) const {
  return !(*this == other);
 }
};

int main() {
 Box box1(10, 20, 30);
 Box box2(10, 20, 30);
 Box box3(5, 15, 25);

 if (box1 == box2) {
  std::cout << "box1 is equal to box2" << std::endl; // Output: box1 is equal to box2
 } else {
  std::cout << "box1 is not equal to box2" << std::endl;
 }

 if (box1 != box3) {
  std::cout << "box1 is not equal to box3" << std::endl; // Output: box1 is not equal to box3
 } else {
  std::cout << "box1 is equal to box3" << std::endl;
 }

 return 0;
}
```

- Example 6: Overloading the `()` Operator for Functors

```
#include <iostream>

class Multiply {
 private:
 int factor;

 public:
 Multiply(int f) : factor(f) {}

 // Overload () operator
 int operator()(int x) const {
  return x * factor;
 }
```

```cpp
};

int main() {
 Multiply multiplyBy3(3);

 std::cout << "3 * 5 = " << multiplyBy3(5) << std::endl; // Output: 3 * 5 = 15
 std::cout << "3 * 10 = " << multiplyBy3(10) << std::endl; // Output: 3 * 10 = 30

 return 0;
}
```

7.9.2 Guidelines for Operator Overloading

1. **Maintain Intuitive Behavior:** Overloaded operators should behave in a way that is intuitive and consistent with their built-in counterparts.

2. **Non-Member vs. Member Functions:** Use non-member functions when the left-hand operand is not the same type as the class. Use member functions when the left-hand operand is of the class type.

3. **Symmetric Operators:** If you overload an operator like `==`, you should also overload `!=` to ensure consistency.

4. **Efficiency:** Avoid unnecessary copying. Use references and move semantics where applicable.

Operator overloading enhances the readability and usability of user-defined types in C++. By properly overloading operators, you can make your custom classes behave like built-in types, leading to more natural and intuitive code. Use operator overloading judiciously to maintain code clarity and consistency.

7.10 Digging into Memory Management and Smart Pointers in C++

(in progress!)