Contents

1 Introduction . . . 3
1.1 A Short Overview of Programming Languages . . . 3
1.1.1 C, Strengths and Weaknesses . . . 4
1.2 Shell Basics on Linux . . . 5
2 Fundamentals of C Language . . . 9
2.1 Writing a Simple Program, `Hello, McMaster!` . . . 9
2.1.1 A General Form of a Program . . . 9
2.1.2 Compiling and Executing the Program . . . 11
2.2 Integer Data Type . . . 13
2.2.1 Binary Representation . . . 13
2.2.2 Using `printf` and `limits.h` to Get the Limits of Integers . . . 15
2.2.3 Declaring and Naming with and without Initializing . . . 19
2.2.4 Constant Variables . . . 20
2.2.5 Arithmetic Operations on Integers . . . 21
2.2.6 A Simple Example for Integer Overflows . . . 22
2.2.7 Fixed-width Integer types . . . 25
2.3 Characters and strings . . . 27
2.4 Floating-point Numbers . . . 29
2.4.1 Rounding Error in Floating-point Numbers . . . 33
2.4.2 Type Conversion . . . 35
2.4.3 Arithmetic Operations on Floating-point Numbers . . . 35
2.5 Mathematical Functions . . . 35
2.6 Statements . . . 35
2.6.1 Comparison: `if` and `switch` . . . 35

2.6.2 Loops and Iterations: `while` . . . 35
2.6.3 Loops and Iterations: `do` . . . 35
2.6.4 Loops and Iterations: `for` . . . 35
2.6.5 Loops and Iterations: Exiting from a Loop . . . 36
2.6.6 Nested loops . . . 36
2.6.7 Variable Scope . . . 36
2.7 Arrays . . . 36
2.8 Functions . . . 36
3 Intermediate Topics in C . . . 36
4 Advanced Topics in C . . . 37
5 A Short Overview of ANN . . . 37

1 Introduction

1.1 A Short Overview of Programming Languages

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

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

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

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

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

1.1.1 C, Strengths and Weaknesses

Weaknesses

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

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

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

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

Strength

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

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

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

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

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

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

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

1.2 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 "COMPSCI1XC3," you would type `mkdir COMPSCI1XC3`. 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 `COMPSCI1XC3` 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 `COMPSCI1XC3` folder in the file manager, use the command `open COMPSCI1XC3`.

2. Moving back and forth in directories: To change your current directory to the `COMPSCI1XC3` folder, use the command `cd COMPSCI1XC3` 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 `COMPSCI1XC3` directory. If you want to create a text file named "AnneMarie", use the command `nano AnneMarie.txt`. This will open the file editor where you can write and edit text. To open the `AnneMarie.txt` file in the Nano text editor, use the command `nano AnneMarie.txt` again. This allows you to make changes to the file. If you wish to save your changes, press `Ctrl+O` and then press `Enter`. This will save the file. Alternatively, if you want to save and exit the Nano text editor, press `Ctrl+X` and then press `Enter`. This will save the changes and return you to the terminal.

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

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

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

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

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

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

```
/var/lib/plocate/:  Permission denied
```

indicating that you do not have the necessary permissions to access the directory. To solve the issue and update the directories index successfully, use the `sudo updatedb` which prompts you to a password which you picked. Enter the password, it might not be shown when you are entering the password, then press `Enter`. By this command, you are running the `updatedb` with superuser

(root) privileges. `sudo` stands for "Super User Do" and is a command that allows regular users to execute commands with the security privileges of the superuser. Now you can use command `locate`, but this time it shows the directory that the file is saved.

Alternatively, you can use:

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

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

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

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

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

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

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

8. Let's checkout some OS characteristics.

Linux distribution: distThe `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 key takes you to 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:

```
Incldуing 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.

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

int main() {

 printf("Size of char: %zu bits\n", 8 * sizeof(char));
```

```
    printf("Signed char range: %d to %d\n", SCHAR_MIN, SCHAR_MAX);
    printf("Unsigned char range: %u to %u\n", 0, UCHAR_MAX);
    printf("\n");

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`-std=c99`: This flag sets the C language standard that the compiler should adhere to. In this case, c99 indicates the **C99 standard**. The **C99 standard** refers to the ISO/IEC 9899:1999 standard for the C programming language. It introduces several new features and improvements compared to earlier versions of the C standard, such as support for variable declarations anywhere in a block, support for `//` single-line comments, and new data types like `long long`. By specifying `-std=c99`, you ensure that the compiler follows the **C99 standard** while compiling your code.

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

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

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

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

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

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

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

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

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

Overflows.

2.2.3 Declaring and Naming with and without Initializing

To declare a variable you can simply:

```
int Pedram;
```

where `int` is the type of variable and `Pedram` is the name of variable. You can after this line of code calculate the value of `Pedram`. You may also declare the value for this variable when it is initialized:

```
int Pedram = 10;
```

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

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

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

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

```
#include <stdio.h>

int main() {
```

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

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

```
-1240674203
```

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

```
233918565
```

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

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

2.2.4 Constant Variables

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

```
#include <stdio.h>

int main() {
 const int MAX_VALUE = 100;

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

Start developing this habit to mention the constant values during programming which make you code to be more understandable. After using `const` for variable `MAX_VALUE`, I cannot change the

value for this variable. You can try to do it and you will see errors like:

`expression must be a modifiable lvalue`

or

`assignment of read-only variable ‘MAX_VALUE’`

2.2.5 Arithmetic Operations on Integers

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

```c
#include <stdio.h>

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

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

 return 0;
}
```

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

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

2.2.6 A Simple Example for Integer Overflows

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

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

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

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

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

 return 0;
}
```

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

```
Value: -2147483648
```

while I was expecting to see 2,147,483,64**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.

```
#include <stdio.h>

int main() {
```

```
// ------------------------- char -------------------------
char Ped_RealChar = 'P';
// char range: -128 to 127
char Ped_NumChar = 80;
// unsigned char range: 0 to 255
unsigned char Ped_NumChar_unsigned = 252;
// ------------------------- short -------------------------
// short range: -32768 to 32767
short Ped_sh = -1234;
// unsigned short range: 0 to 65535
unsigned short Ped_sh_unsigned = 56789;
//-------------------------- int --------------------------
// int range: -2147483648 to 2147483647
int Ped_int = -42;
// unsigned int range: 0 to 4294967295
unsigned int Ped_int_unsigned = 123456;
// ------------------------- long -------------------------
// long range: -9223372036854775808 to 9223372036854775807
long Ped_long = -9876543210;
// unsigned long range: 0 to 18446744073709551615
unsigned long Ped_long_unsigned = 9876543210;
// #----------------------- long long -----------------------
// long long range: -9223372036854775808 to
   9223372036854775807
long long Ped_longlong = -123456789012345;
// unsigned long long range: 0 to 18446744073709551615
unsigned long long Ped_longlong_unsigned = 123456789012345;

// ----------------------- printing -----------------------
printf("char used for saving character: %c\n", Ped_RealChar);
printf("char used saving integer BUT the character is printed
   : %c\n", Ped_NumChar);
printf("char used saving integer: %d\n", Ped_NumChar);
printf("unsigned char saving integer: %u\n",
   Ped_NumChar_unsigned);
printf("short: %hd\n", Ped_sh);
printf("unsigned short: %hu\n", Ped_sh_unsigned);
printf("int: %d\n", Ped_int);
printf("unsigned int: %u\n", Ped_int_unsigned);
```

```
    printf("long: %ld\n", Ped_long);
    printf("unsigned long: %lu\n", Ped_long_unsigned);
    printf("long long: %lld\n", Ped_longlong);
    printf("unsigned long long: %llu\n", Ped_longlong_unsigned);
}
```

Now you have the source code `Pedram.c`, open it in VScode by executing `code Pedram.c` in the terminal. You should be able to see the code in the opened window. Now we need another terminal inside the VScode to compile and run the code. To open a terminal in VScode, go to the View menu at the top of the window. From the View menu, select "Terminal". You can see which directory this terminal is in by executing `pwd` in the terminal. Change your directory to the one where source code `Pedram.c` is. We need to do this so when we are compiling the code, the compiler can find the source code and translate it to the machine's language!

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

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

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

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

```
    unsigned long long: 123456789012345
```

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

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

2.2.7 Fixed-width Integer types

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

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

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

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

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

`gcc -o FixedInteger FixedInteger.c`

and

`./FixedInteger`.

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

int main() {

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

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

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

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

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

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

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

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

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

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

Size of uint8_t: 8 bits
uint8_t range: 0 to 255

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

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

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

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

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

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

2.3 Characters and strings

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

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

To print a string, we use the placeholder `%s` in `printf`. For example, if `char s[] = "Pedram"`,

then `printf("%s", s);` will print the contents of the string variable `s`.

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

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

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

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

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


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

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

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

 return 0;
}
```

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

`/home/username/COMPSCI1XC3`.

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:

```c
#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!**

```c
#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 1.234567880630493164 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
```

```c
#include <stdio.h>

int main() {
 printf("float:       %.30f\n", 0.25f);
 printf("double:      %.30f\n", 0.25);
 printf("long double: %.30Lf\n\n", 0.25L);

 printf("float:       %.30f\n", 0.30f);
 printf("double:      %.30f\n", 0.30);
 printf("long double: %.30Lf\n", 0.30L);
```

```
    return 0;
}
```

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

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

3rd W/1and2

2.4.2 Type Conversion

(Explicit conversion: type casting,)

2.4.3 Arithmetic Operations on Floating-point Numbers

.

2.5 Mathematical Functions

2.6 Statements

2.6.1 Comparison: `if` and `switch`

including boolean and switch

2.6.2 Loops and Iterations: `while`

2.6.3 Loops and Iterations: `do`

2.6.4 Loops and Iterations: `for`

2.6.5 Loops and Iterations: Exiting from a Loop

including break, continue, goto

2.6.6 Nested loops

4th W/1

2.6.7 Variable Scope

2.7 Arrays

2.8 Functions

Constant vs. non-constant arguments

Forward declaration and mutual recursion

Global variables

4th W/2 + remained

3 Intermediate Topics in C

including:

debugging

MakeFiles

5th W/1

pointers

Dynamic memory allocation

5th W/2

4 Advanced Topics in C

including:

structs and typedefs

input and output reading a file or output a file

6th W/1 parallel computing

further topics

6th W/2

5 A Short Overview of ANN