Shaheed Zulfikar Ali Bhutto
Institute of Science & Technology

Fundamentals of Programming

(CSCL1103)

LABORATORY MANUAL

(Fall 2021)

Student Name:	______________________
Roll No:	______________________
Section:	______________________

Lab Instructor:
Sheikh Usama Khalid

LIST OF EXPERIMENTS

S.no	Date	Experiment Name	Marks	Signature
1		Introduction: Computer Organization; Algorithms, flowchart; Computer languages; Compiler, Assembler and Interpreter; Familiarization with Code Block		
2		Pseudo Code & Input and Output with C		
3		Defining and implement different Data types, Data representation, Identifiers, Reserved words, Variables and constants.		
4		Arithmetic and logical Operators		
5		Conditional Statements		
6		Switch case statements		
7		Loops		
8		Arrays		
9		Functions		
10		Pointers, Dynamic memory allocation, ragged arrays		
11		Searching and sorting algorithm.		
12		Structures, Structure declaration, accessing structure members, array of structures		
13		Passing structures as function arguments		
14		File Handling Read / Write		
15		Project		

Experiment 1

Introduction

Computer Organization:

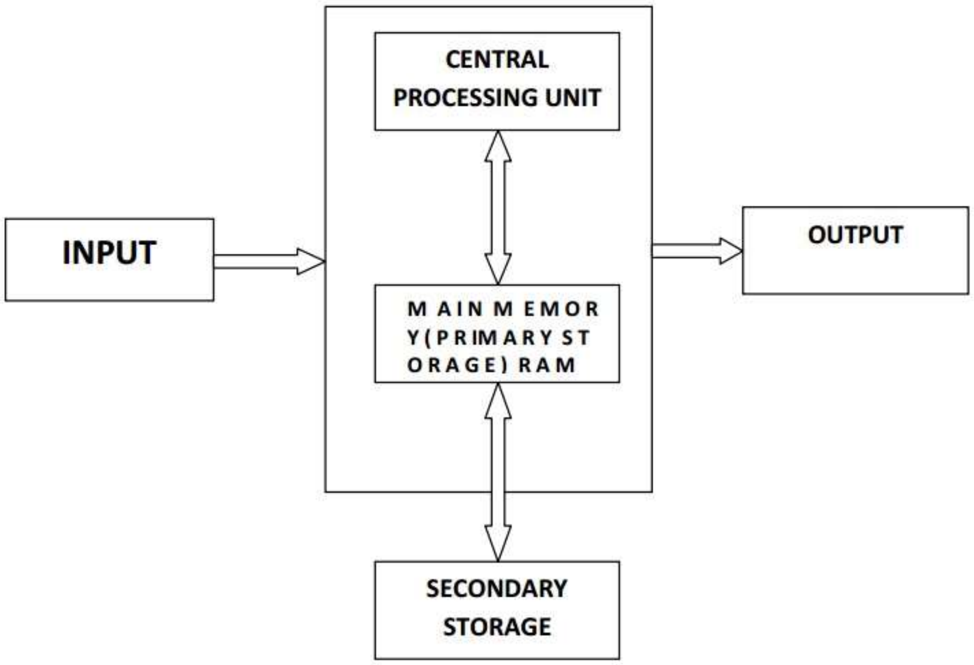
- Computer Organization is concerned with the structure and behavior of a computer system as seen by the user.

- It deals with the components of a connection in a system.

- Computer Organization tells us how exactly all the units in the system are arranged and interconnected.

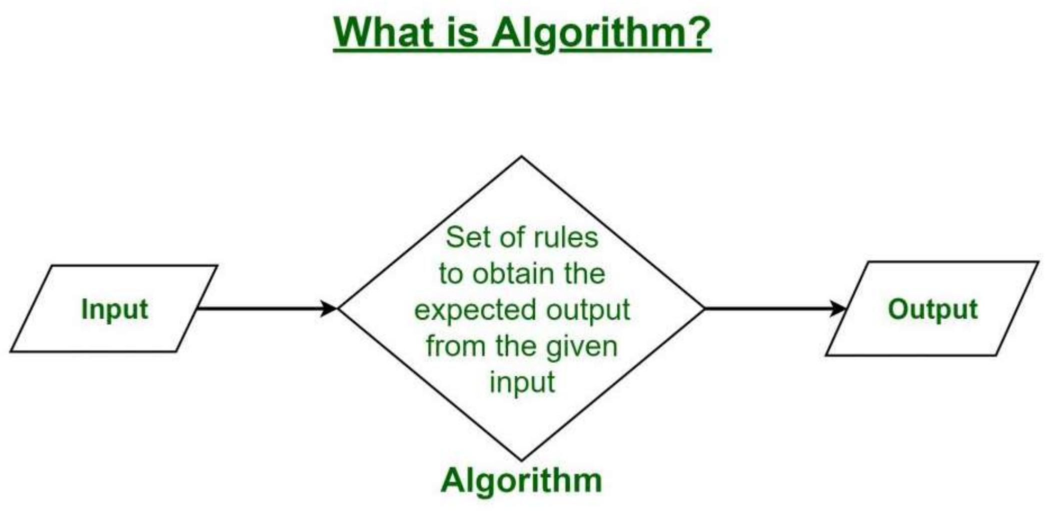
- Computer Organization deals with low-level design issues.

- Computer Organization involves Physical Components (Circuit design, Adders, Signals, and Peripherals).

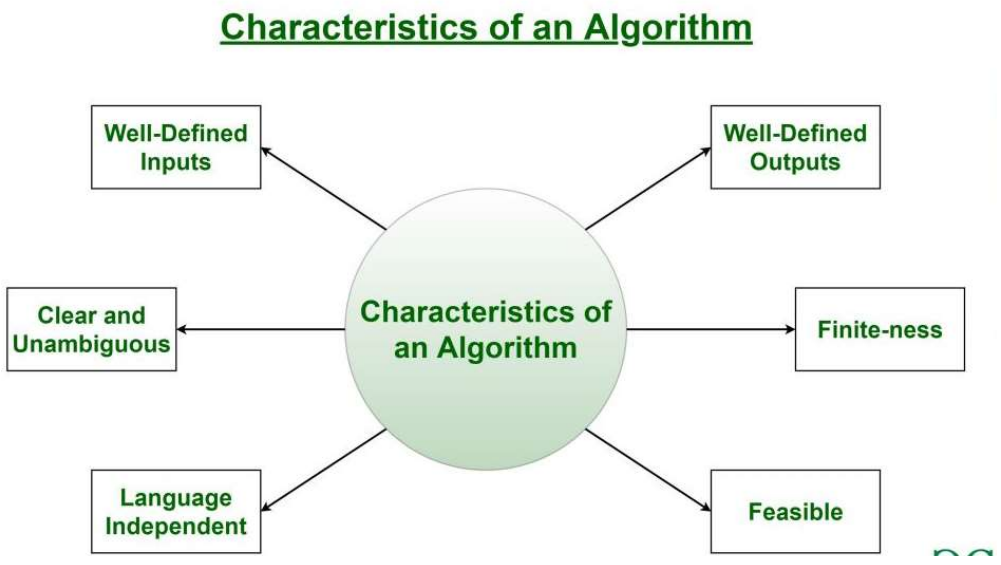


Algorithms:

- ❖ An algorithm is a set of instructions for solving a problem or accomplishing a task. One common example of an algorithm is a recipe, which consists of specific instructions for preparing a dish or meal.


What is Algorithm?
Input
Set of rules to obtain the expected output from the given input
Output
Algorithm

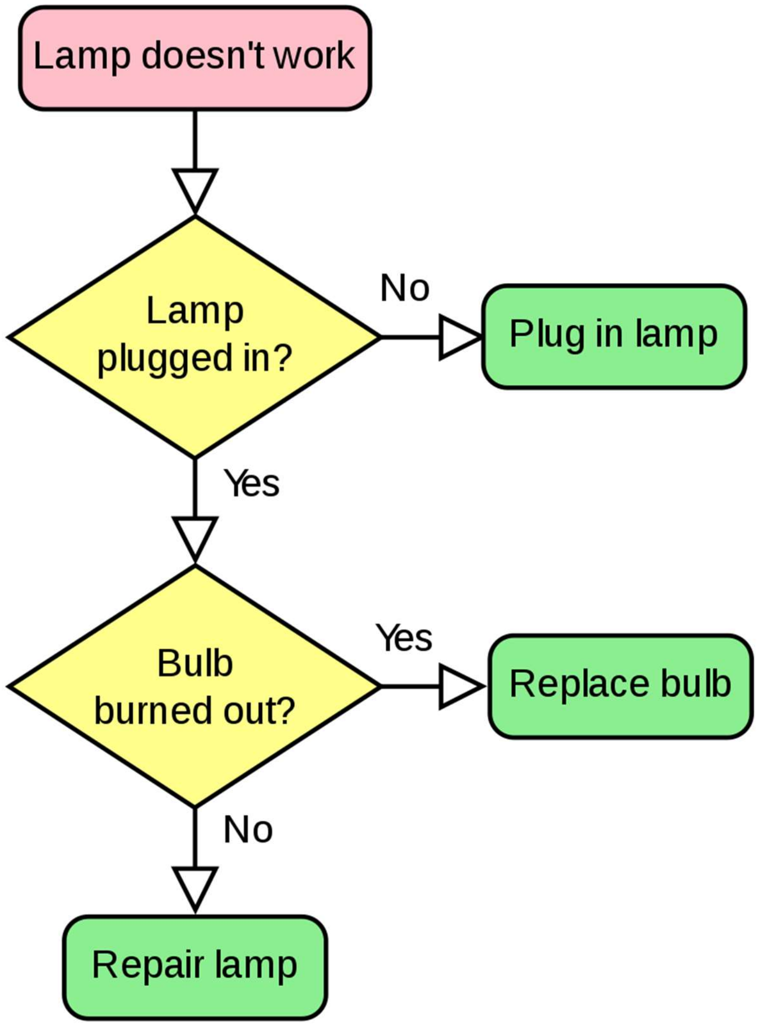
Characteristics of Algorithm:



Flowchart:

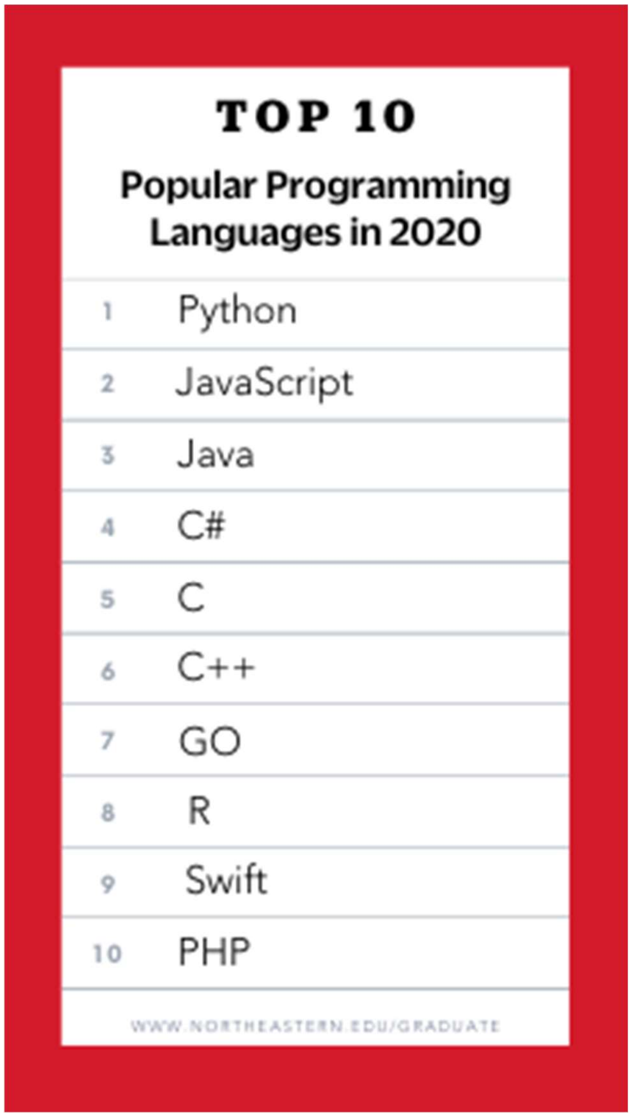
- A flowchart is a type of diagram that represents a workflow or process. A flowchart can also be defined as a diagrammatic representation of an algorithm, a step-by-step approach to solving a task.

- The flowchart shows the steps as boxes of various kinds, and their order by connecting the boxes with arrows. This diagrammatic representation illustrates a solution model to a given problem. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.



Computer languages:

❖ Computer language is a formal language used in communication with a computer.



TOP 10

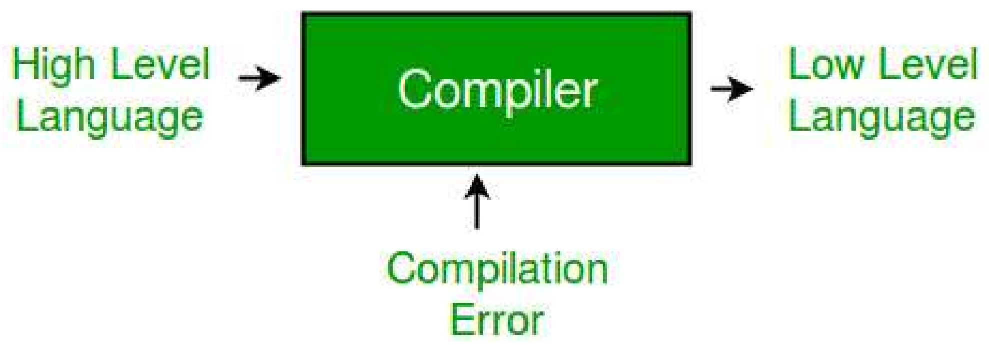
Popular Programming Languages in 2020

1	Python
2	JavaScript
3	Java
4	C#
5	C
6	C++
7	GO
8	R
9	Swift
10	PHP

WWW.NORTHEASTERN.EDU/GRADUATE

Compiler:

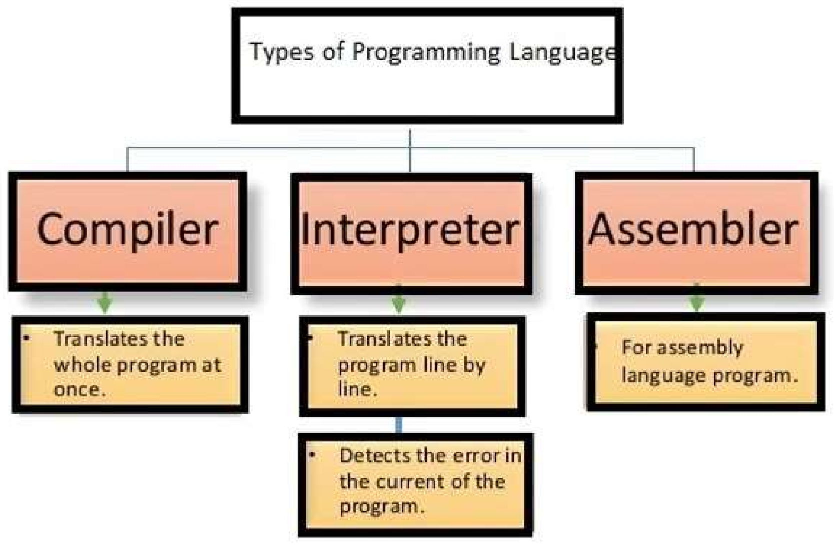
❖ A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language). The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g. assembly language, object code, or machine code) to create an executable program.



Assembler and Interpreter:

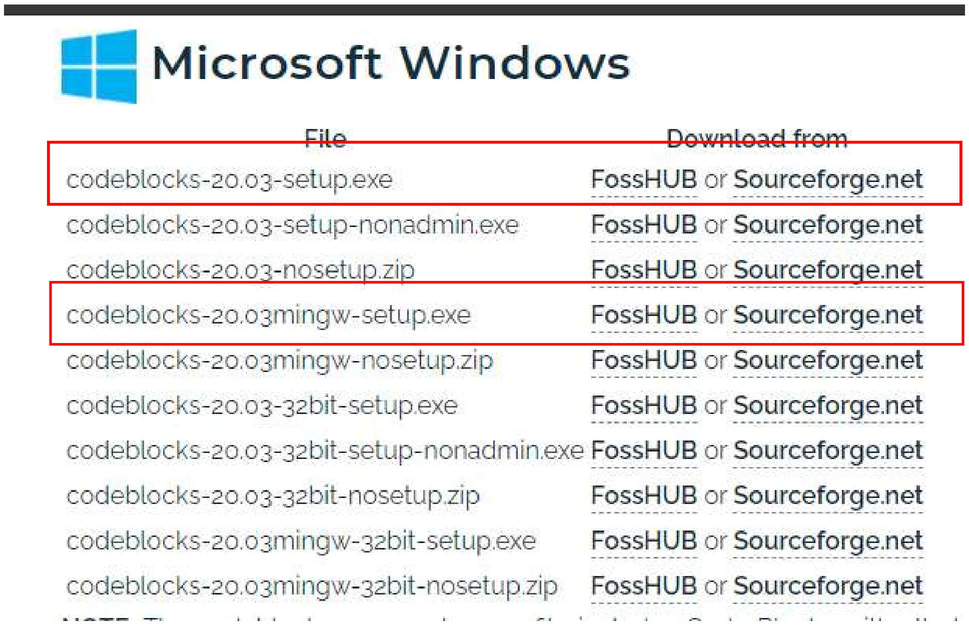
- The programs created in high level languages can be executed by using two different ways. The first one is the use of compiler and the other method is to use an interpreter. High level instruction or language is converted into intermediate from by an interpreter. The advantage of using an interpreter is that the high level instruction does not goes through compilation stage which can be a time consuming method. So, by using an interpreter, the high level program is executed directly. That is the reason why some programmers use interpreters.

- An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. Some people call these instructions assembler language and others use the term assembly language.


Types of Programming Language
Compiler
Interpreter
Assembler
Translates the whole program at once.
Translates the program line by line.
Detects the error in the current of the program.
For assembly language program.

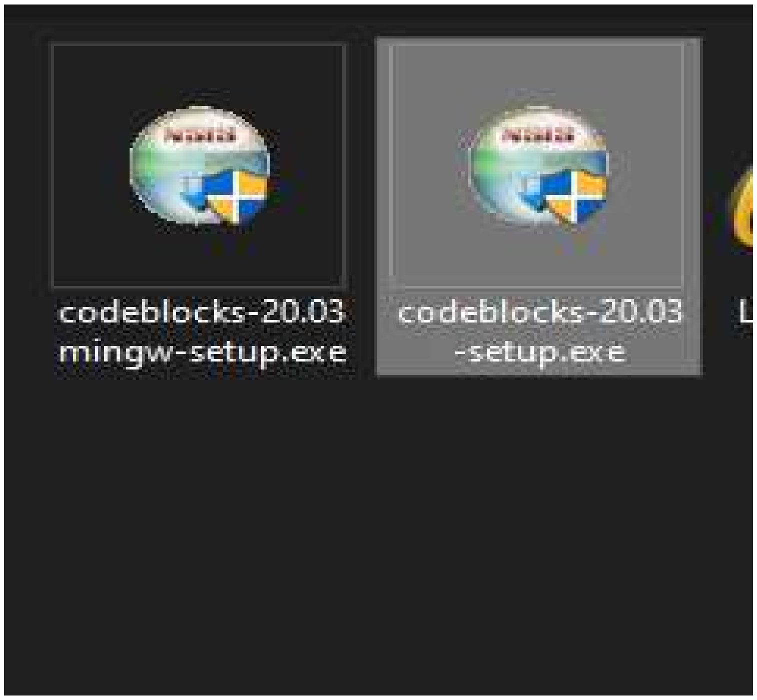
Familiarization with Code Block:

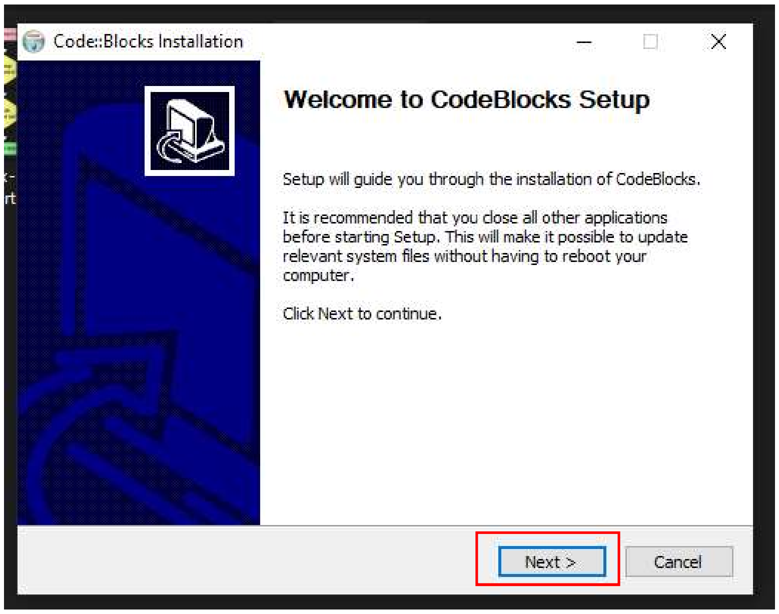
- ❖ Visit the site to download Code Block: https://www.codeblocks.org/downloads/binaries/



Microsoft Windows

File	Download from
codeblocks-20.03-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03-setup-nonadmin.exe	FossHUB or Sourceforge.net
codeblocks-20.03-nosetup.zip	FossHUB or Sourceforge.net
codeblocks-20.03mingw-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03mingw-nosetup.zip	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-setup-nonadmin.exe	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-nosetup.zip	FossHUB or Sourceforge.net
codeblocks-20.03mingw-32bit-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03mingw-32bit-nosetup.zip	FossHUB or Sourceforge.net



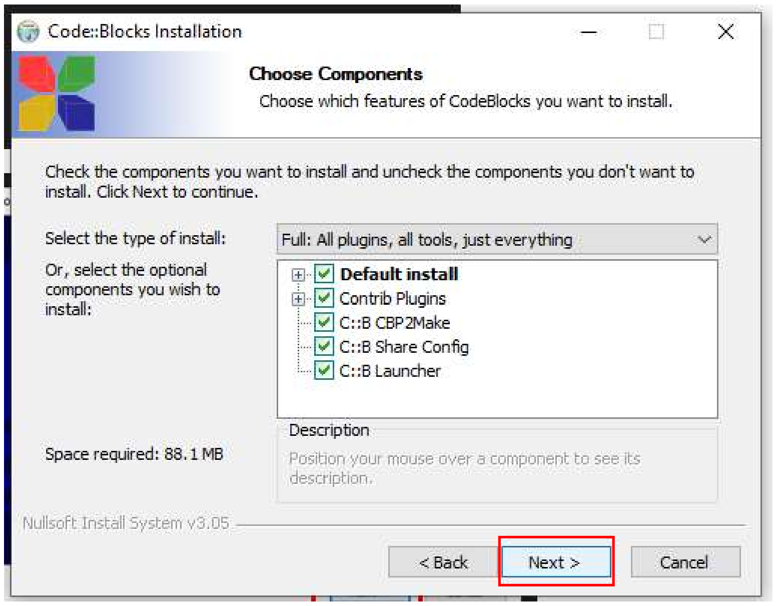

Code::Blocks Installation

Welcome to CodeBlocks Setup

Setup will guide you through the installation of CodeBlocks.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.


Code::Blocks Installation

Choose Components

Choose which features of CodeBlocks you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select the type of install: Full: All plugins, all tools, just everything

Or, select the optional components you wish to install:

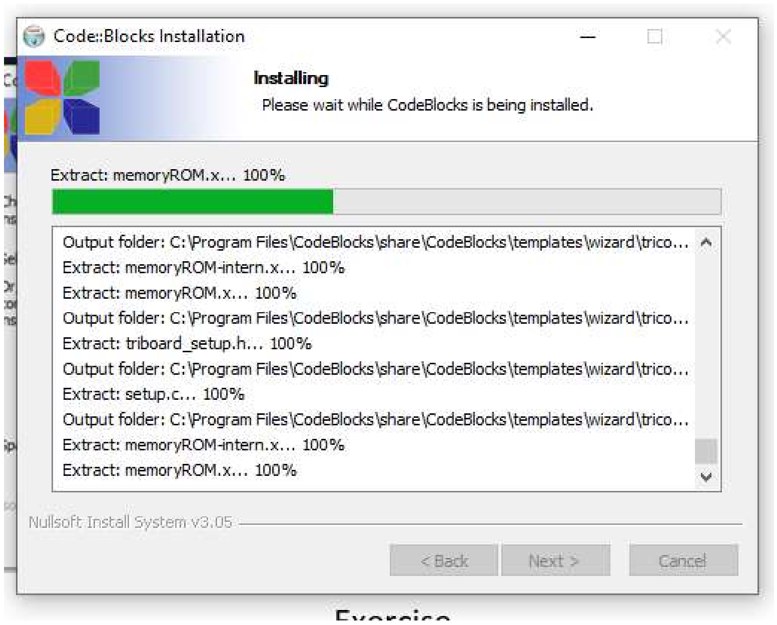
- [✓] **Default install**
- [✓] Contrib Plugins
- [✓] C::B CBP2Make
- [✓] C::B Share Config
- [✓] C::B Launcher

Space required: 88.1 MB

Description

Position your mouse over a component to see its description.

Nullsoft Install System v3.05



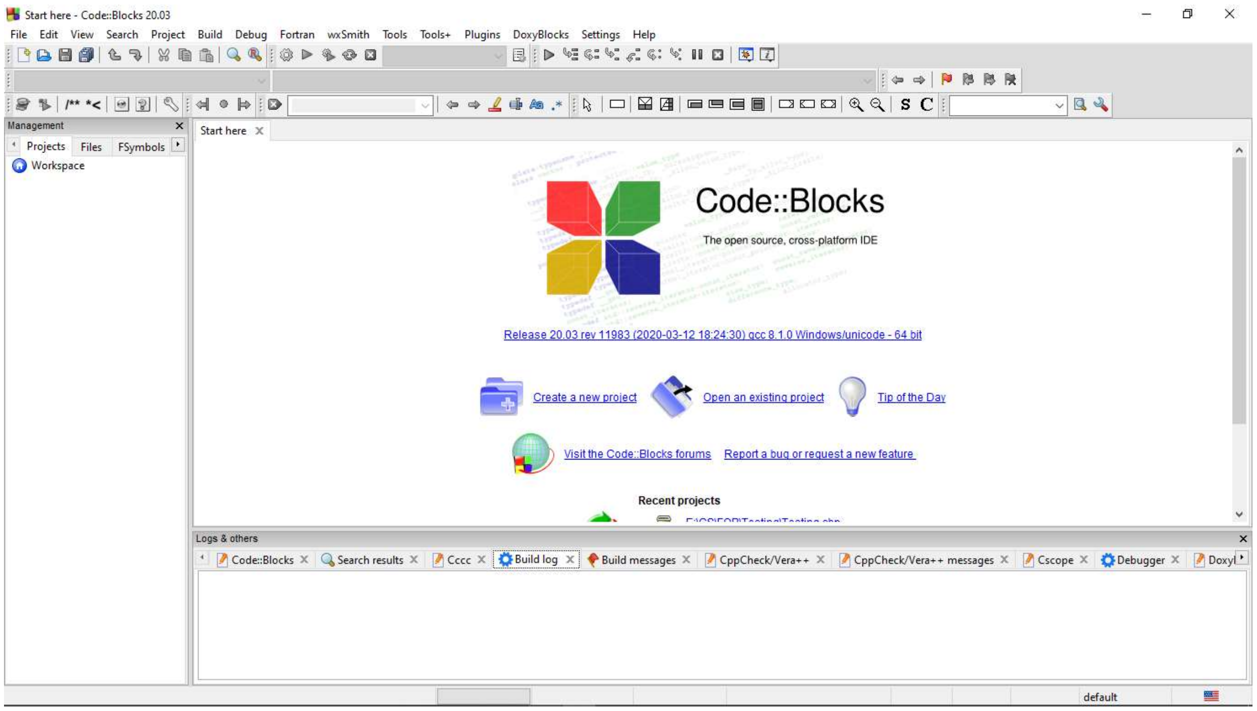
Code::Blocks Installation

Installing

Please wait while CodeBlocks is being installed.

```
Extract: memoryROM.x... 100%

Output folder: C:\Program Files\CodeBlocks\share\CodeBlocks\templates\wizard\trico...
Extract: memoryROM-intern.x... 100%
Extract: memoryROM.x... 100%
Output folder: C:\Program Files\CodeBlocks\share\CodeBlocks\templates\wizard\trico...
Extract: triboard_setup.h... 100%
Output folder: C:\Program Files\CodeBlocks\share\CodeBlocks\templates\wizard\trico...
Extract: setup.c... 100%
Output folder: C:\Program Files\CodeBlocks\share\CodeBlocks\templates\wizard\trico...
Extract: memoryROM-intern.x... 100%
Extract: memoryROM.x... 100%
```

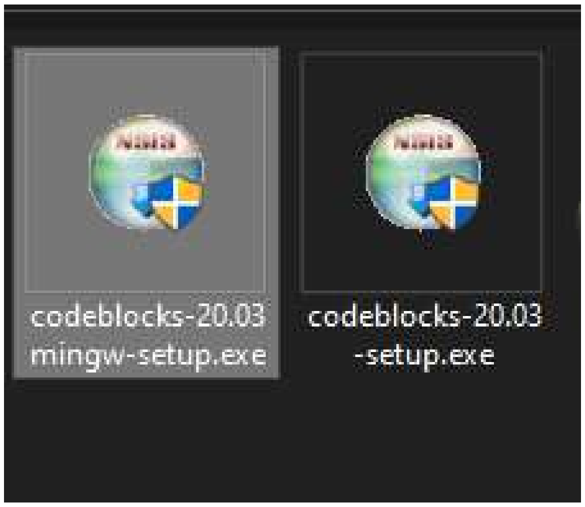
Nullsoft Install System v3.05



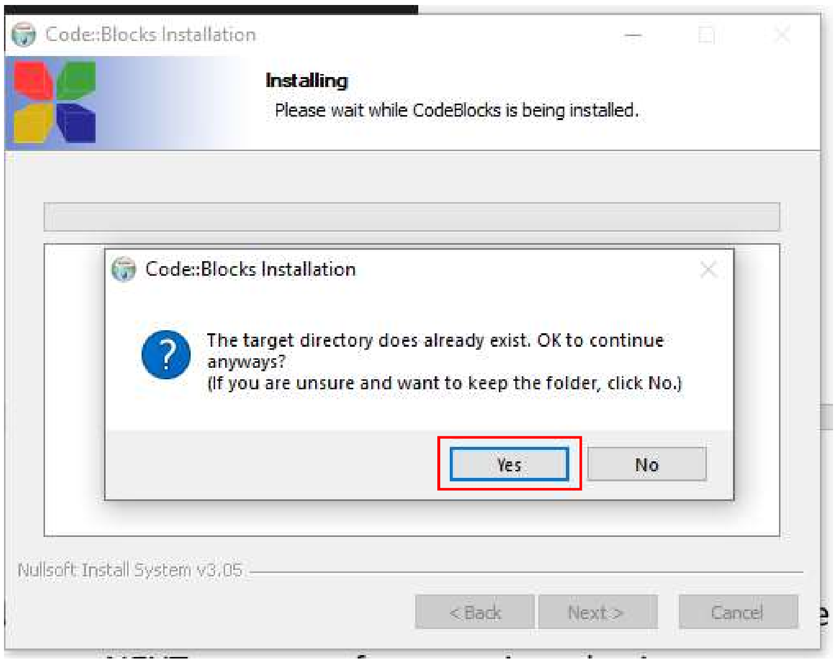
Start here - Code::Blocks 20.03

Release 20.03 rev 11983 (2020-03-12 18:24:30) gcc 8.1.0 Windows/unicode - 64 bit

❖ Now Close the code block window



❖ Run codeblock mingw setup, and do presses the NEXT button same as aforementioned snippets


Code::Blocks Installation

Installing

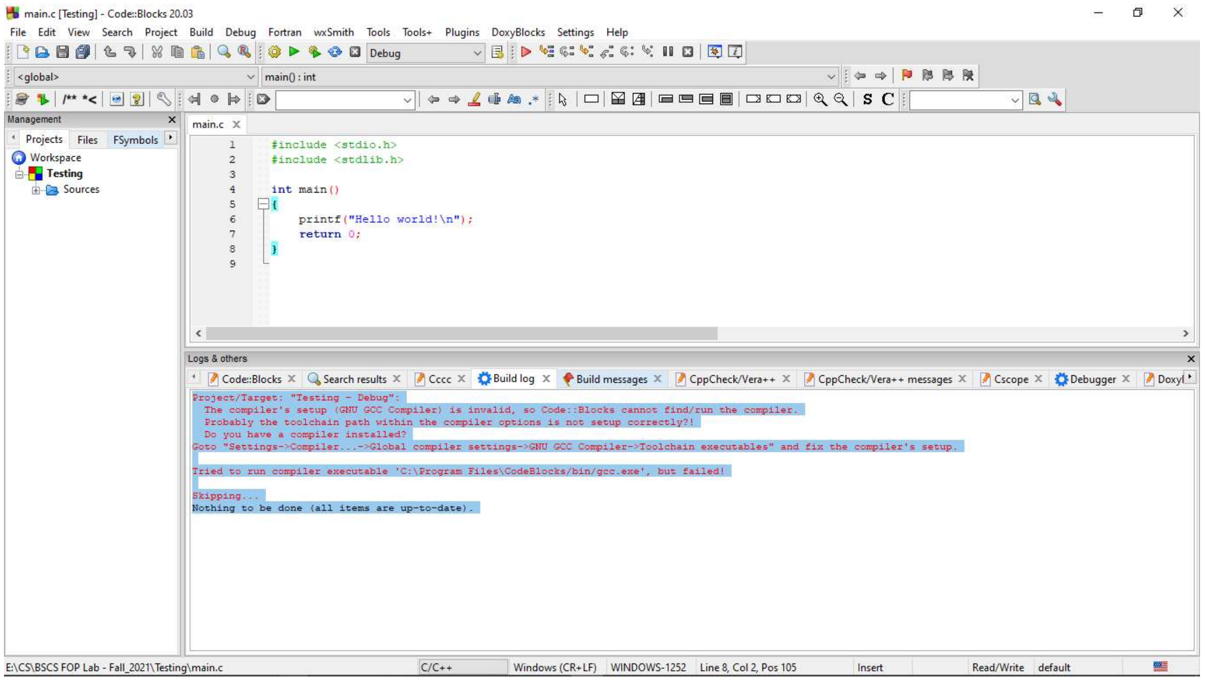
Please wait while CodeBlocks is being installed.

Code::Blocks Installation

The target directory does already exist. OK to continue anyways?
(If you are unsure and want to keep the folder, click No.)

Frequent Error:

❖ GCC compiler Error

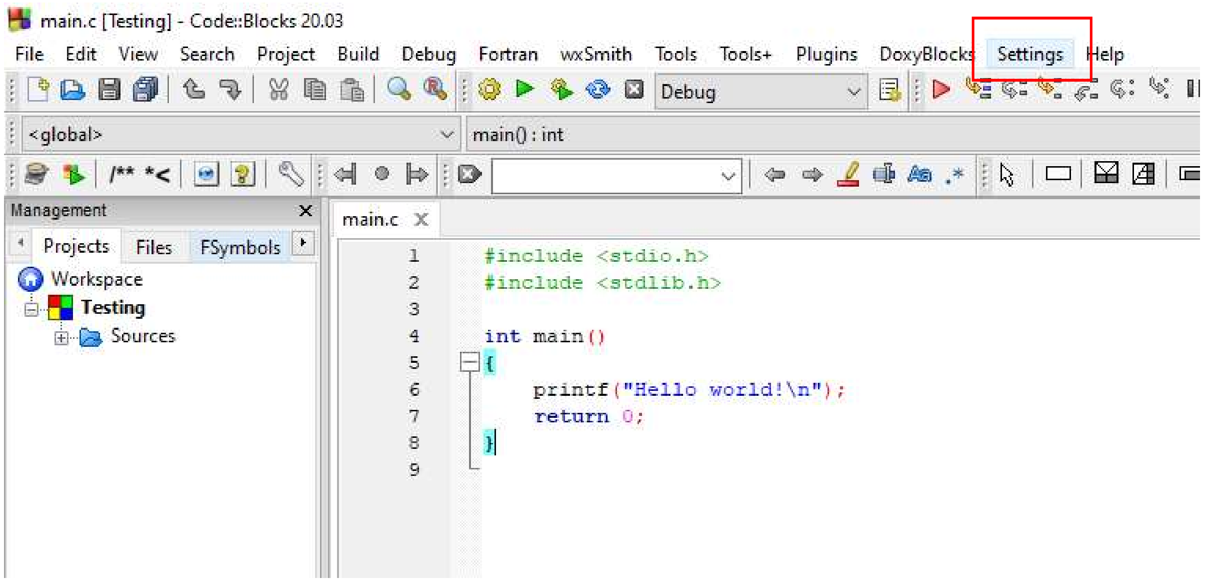


main.c [Testing] - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Management
Projects Files FSymbols
Workspace
Testing
Sources
Logs & others
Code::Blocks Search results Cccc Build log Build messages CppCheck/Vera++ CppCheck/Vera++ messages Cscope Debugger
E:\CS\BSCS FOP Lab - Fall_2021\Testing\main.c C/C++ Windows (CR+LF) WINDOWS-1252 Line 8, Col 2, Pos 105 Insert Read/Write default

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

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

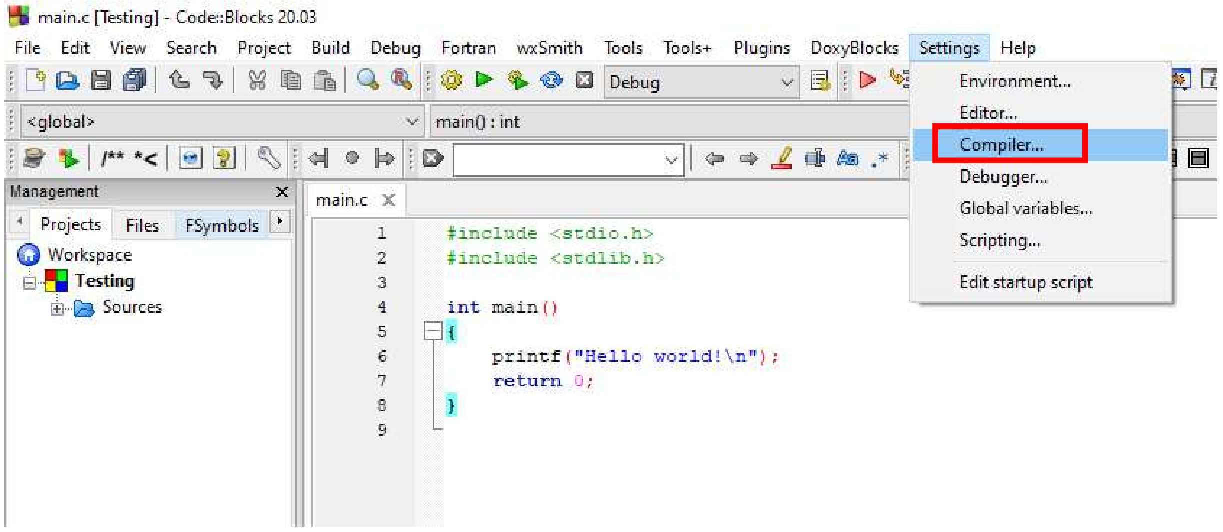
```
Project/Target: "Testing - Debug":
  The compiler's setup (GNU GCC Compiler) is invalid, so Code::Blocks cannot find/run the compiler.
  Probably the toolchain path within the compiler options is not setup correctly?!
  Do you have a compiler installed?
Goto "Settings->Compiler...->Global compiler settings->GNU GCC Compiler->Toolchain executables" and fix the compiler's setup.

Tried to run compiler executable 'C:\Program Files\CodeBlocks/bin/gcc.exe', but failed!

Skipping...
Nothing to be done (all items are up-to-date).
```

❖ How to resolve:

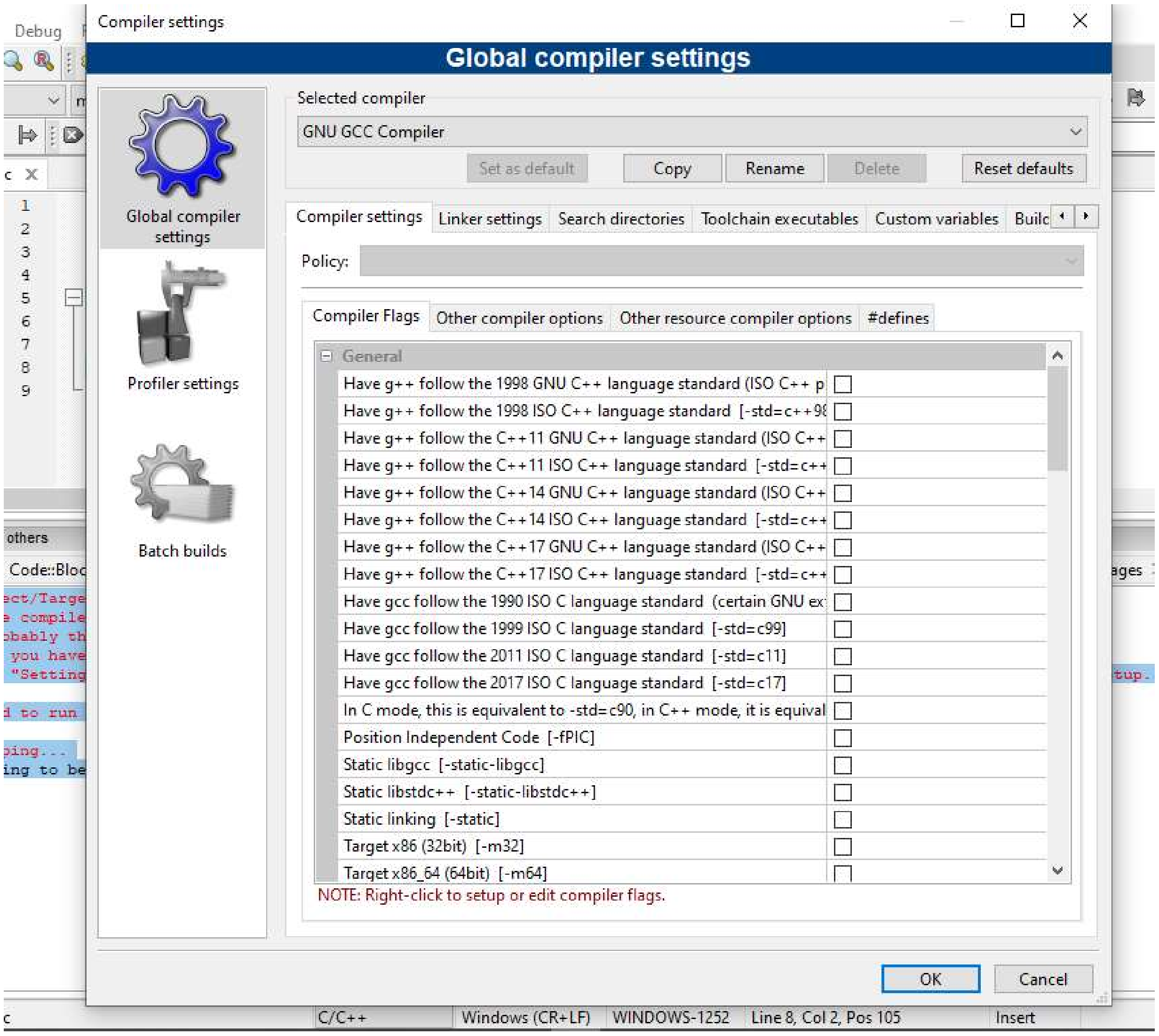


main.c [Testing] - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Debug
<global>
main() : int
Management
Projects Files FSymbols
Workspace
Testing
Sources
main.c

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

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



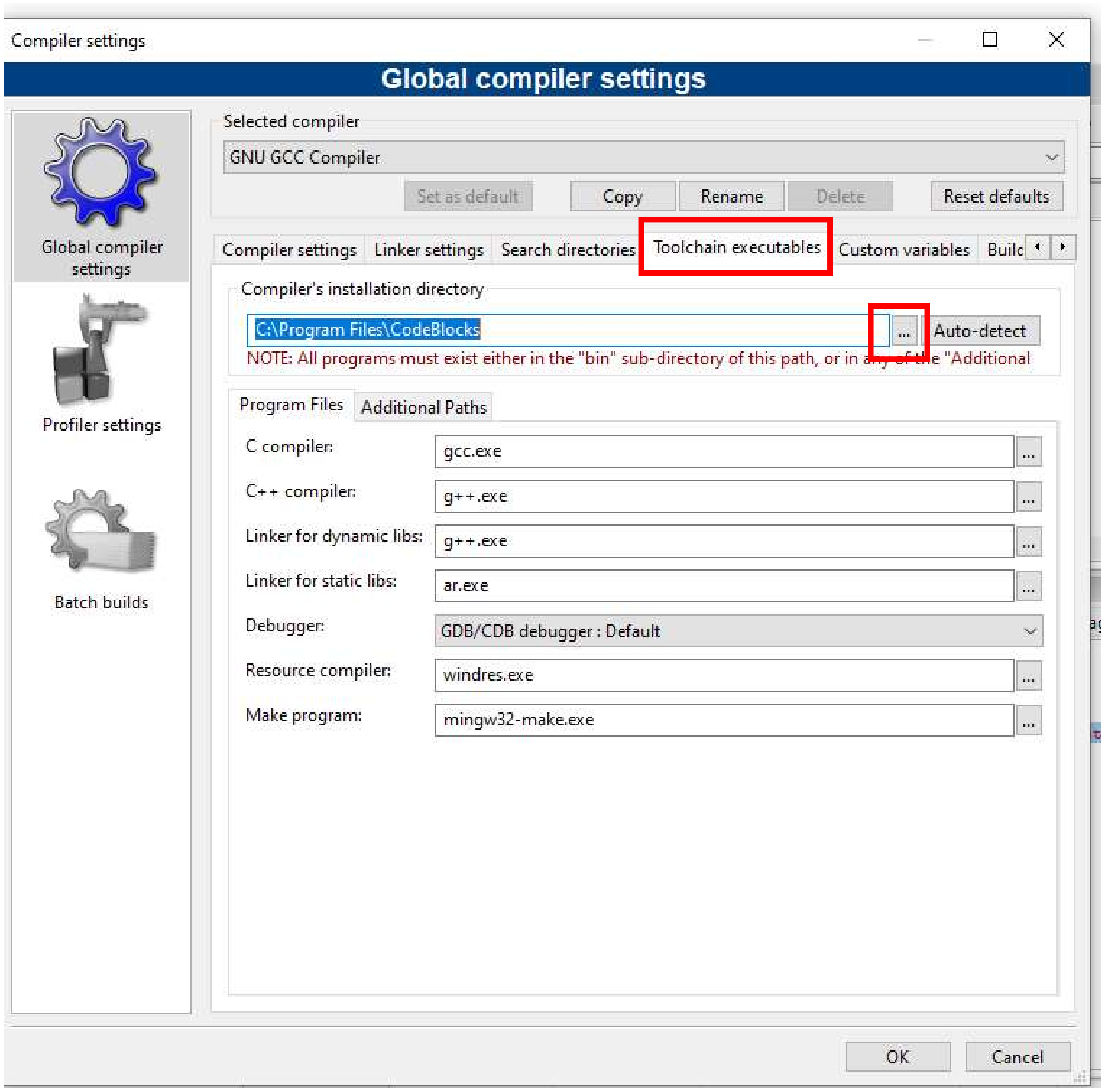
```c
#include <stdio.h>
#include <stdlib.h>

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



Have g++ follow the 1998 GNU C++ language standard (ISO C++ p	☐
Have g++ follow the 1998 ISO C++ language standard [-std=c++98	☐
Have g++ follow the C++11 GNU C++ language standard (ISO C++	☐
Have g++ follow the C++11 ISO C++ language standard [-std=c++	☐
Have g++ follow the C++14 GNU C++ language standard (ISO C++	☐
Have g++ follow the C++14 ISO C++ language standard [-std=c++	☐
Have g++ follow the C++17 GNU C++ language standard (ISO C++	☐
Have g++ follow the C++17 ISO C++ language standard [-std=c++	☐
Have gcc follow the 1990 ISO C language standard (certain GNU ex	☐
Have gcc follow the 1999 ISO C language standard [-std=c99]	☐
Have gcc follow the 2011 ISO C language standard [-std=c11]	☐
Have gcc follow the 2017 ISO C language standard [-std=c17]	☐
In C mode, this is equivalent to -std=c90, in C++ mode, it is equival	☐
Position Independent Code [-fPIC]	☐
Static libgcc [-static-libgcc]	☐
Static libstdc++ [-static-libstdc++]	☐
Static linking [-static]	☐
Target x86 (32bit) [-m32]	☐
Target x86_64 (64bit) [-m64]	☐

NOTE: Right-click to setup or edit compiler flags.

C/C++ Windows (CR+LF) WINDOWS-1252 Line 8, Col 2, Pos 105 Insert


Compiler settings

Global compiler settings

Selected compiler: GNU GCC Compiler

Toolchain executables

Compiler's installation directory: C:\Program Files\CodeBlocks

NOTE: All programs must exist either in the "bin" sub-directory of this path, or in any of the "Additional

Program Files

C compiler: gcc.exe

C++ compiler: g++.exe

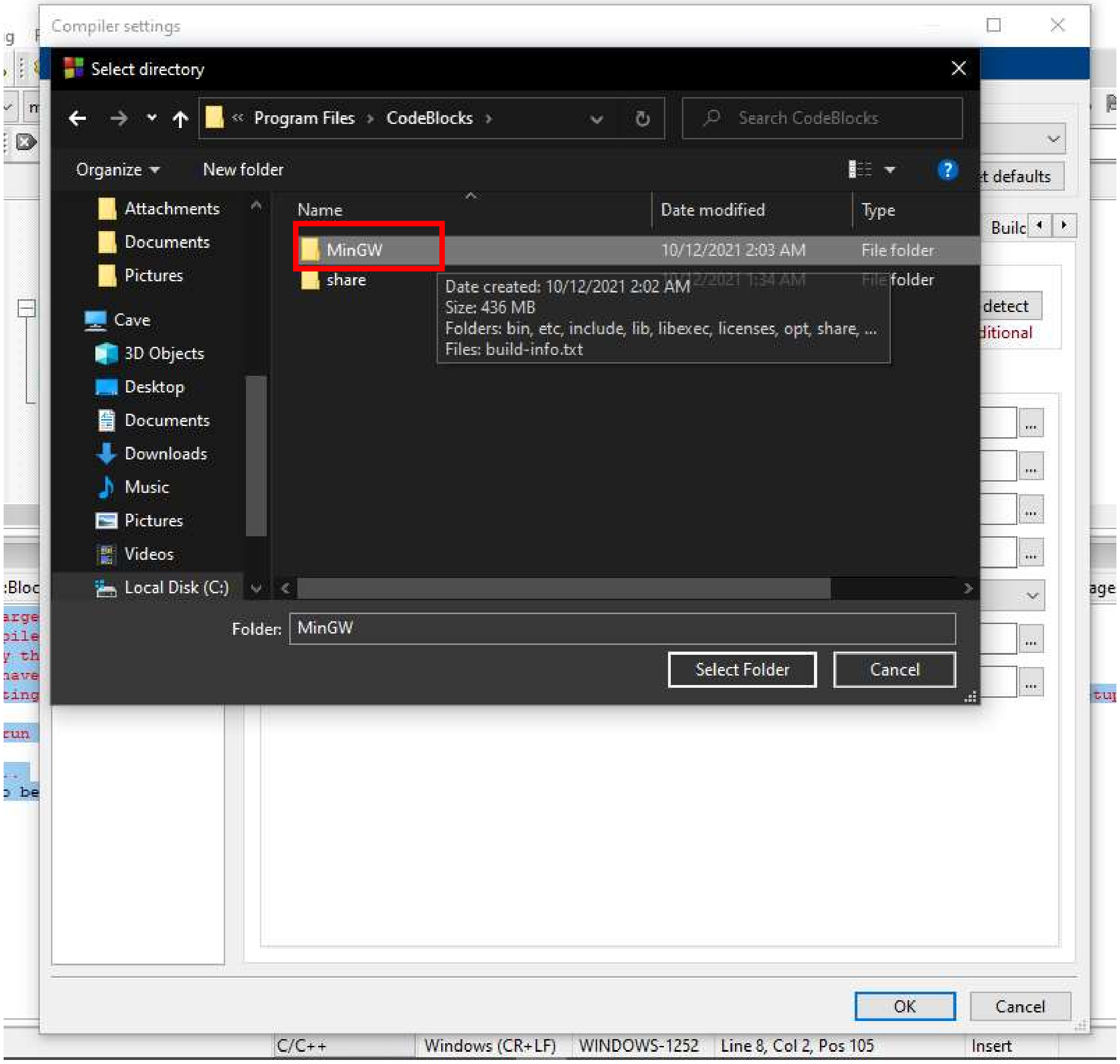
Linker for dynamic libs: g++.exe

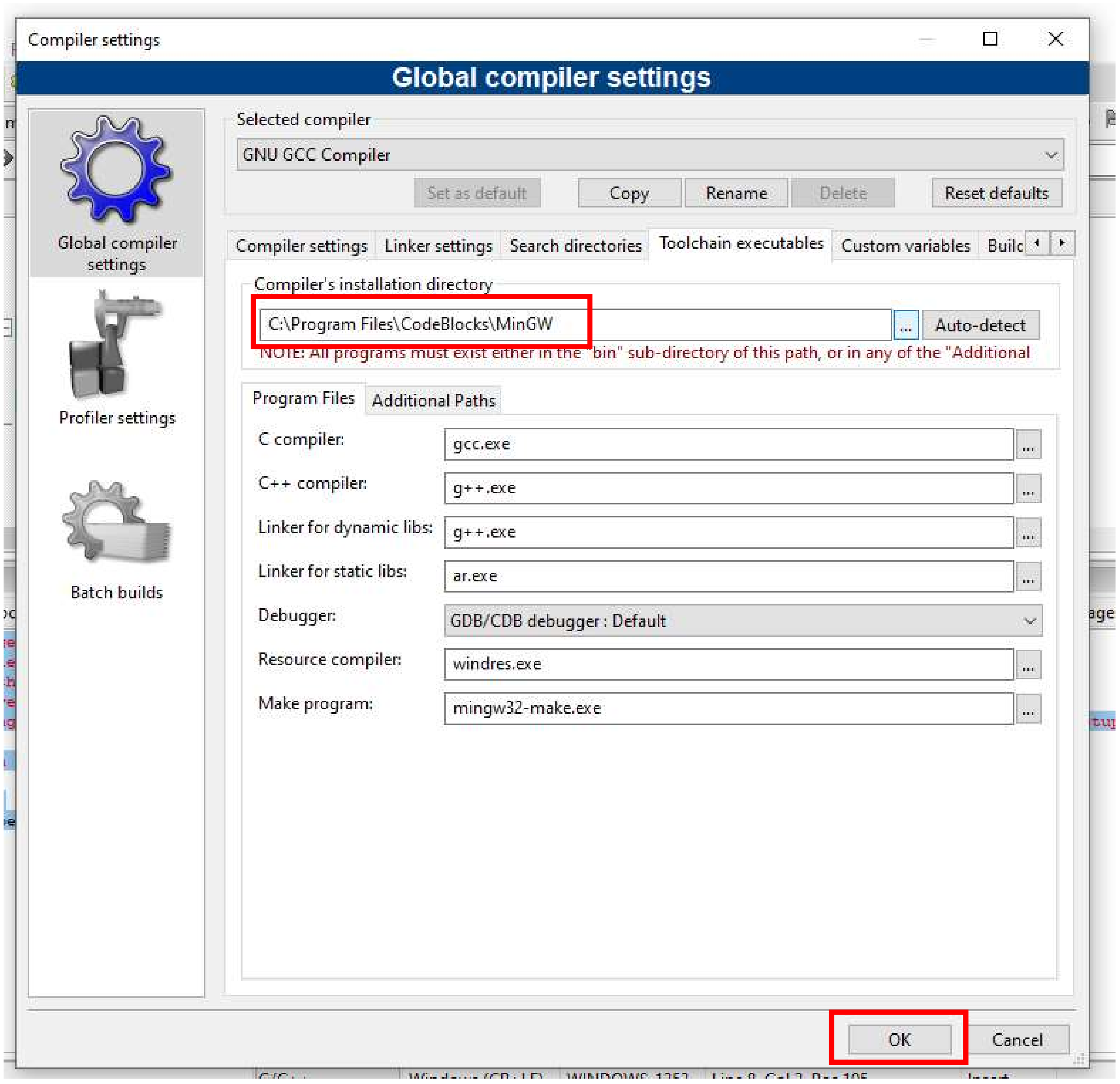
Linker for static libs: ar.exe

Debugger: GDB/CDB debugger : Default

Resource compiler: windres.exe

Make program: mingw32-make.exe


Compiler settings
Select directory
Program Files › CodeBlocks ›
Search CodeBlocks
Organize
New folder
Attachments
Documents
Pictures
Cave
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Local Disk (C:)
Name
Date modified
Type
MinGW
10/12/2021 2:03 AM
File folder
share
File folder
Date created: 10/12/2021 2:02 AM
Size: 436 MB
Folders: bin, etc, include, lib, libexec, licenses, opt, share, ...
Files: build-info.txt
Folder: MinGW
Select Folder
Cancel
OK
Cancel
C/C++
Windows (CR+LF)
WINDOWS-1252
Line 8, Col 2, Pos 105
Insert



Compiler settings

Global compiler settings

Selected compiler: GNU GCC Compiler

Toolchain executables

Compiler's installation directory: C:\Program Files\CodeBlocks\MinGW

NOTE: All programs must exist either in the "bin" sub-directory of this path, or in any of the "Additional

Program Files

C compiler: gcc.exe

C++ compiler: g++.exe

Linker for dynamic libs: g++.exe

Linker for static libs: ar.exe

Debugger: GDB/CDB debugger : Default

Resource compiler: windres.exe

Make program: mingw32-make.exe

Exercise

Objective:

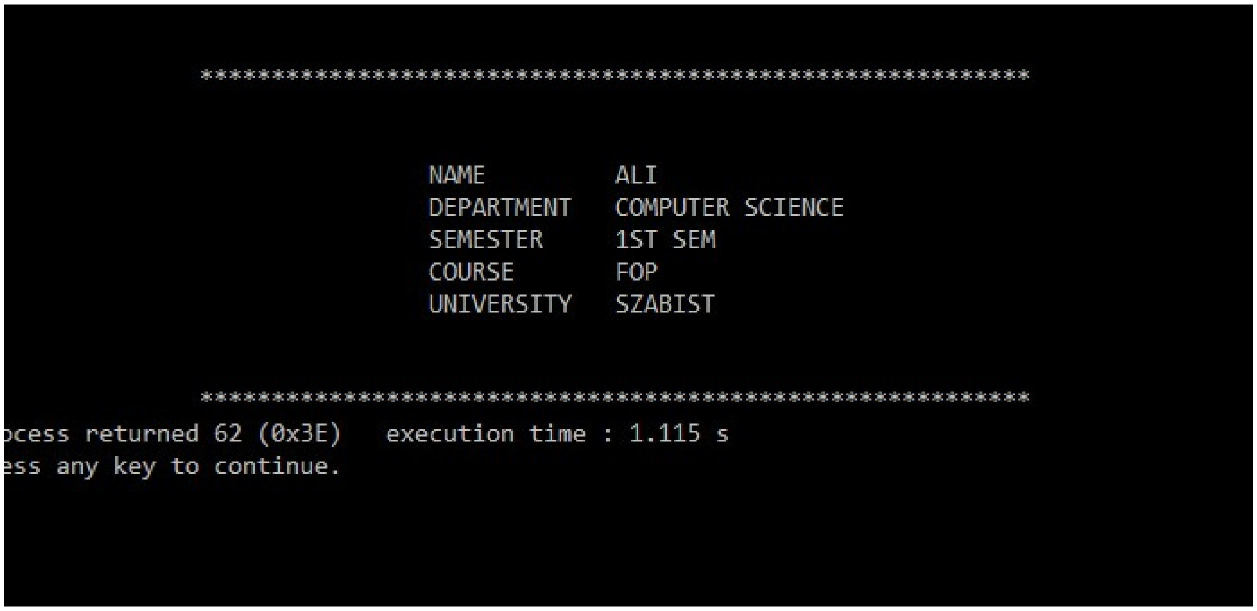
To give introduction about the basic knowledge of fundamentals of programming using C Language.

Sample C Program:

```
#include <stdio.h>

        int main(void)
                {
                        printf(“Hello World");
                }
```

Lab Task:

1. Write a C program to introduce yourself.



```
          **********************************************************


                      NAME         ALI
                      DEPARTMENT   COMPUTER SCIENCE
                      SEMESTER     1ST SEM
                      COURSE       FOP
                      UNIVERSITY   SZABIST


          **********************************************************
ocess returned 62 (0x3E)   execution time : 1.115 s
ess any key to continue.
```

xxx---------------xxxxx----------------xxx

Experiment 2

Pseudocode

Objective:

To be familiar with writing pseudocode in programming language.

1. Write a pseudocode to print your Name

```
1. BEGIN
2. PRINT "Ali"
3. END
```

2. Write a pseudo code to take an input from user and print on console window

```
1. BEGIN
2. DECLARE A
3. INPUT A
4. PRINT A
5. END
```

3. Write a pseudo code to take two numbers as input from user and add them

```
1. BEGIN
2. DECLARE num1, num2, sum
3. INPUT num1,num2
4. sum = num1 + num2
5. DISPLAY sum
6. END
```

Input and Output with C

Objective:

To understand and implement input and output interaction with user.

INPUTS AND OUTPUTS:

When we say **Input**, it means to feed some data into a program. An **input** can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to **output** the data on the computer screen as well as to save it in text or binary files.

THE STANDARD FILES:

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard I/O	stdio	Screen + Keyboard

Format String	Meaning
%d	Scan or print an integer as signed decimal number
%f	Scan or print a floating point number
%c	To scan or print a character
%s	To scan or print a character string. The scanning ends at whitespace.

EXERCISE

```
#include <stdio.h>

        int main()
                {
                    int a;
                    printf("Please input an integer value: ");
                    scanf("%d", &a);
                    printf("You have entered: %d\n", a);
                    return 0;
                }
```

<u>Lab Task:</u>

1. Write a code to show the following output on screen:

    a. Hi Ali, This is the second lab of computer programming.
    b. How many vowels are there in English? (getting input from user)
    c. This code is properly working.

2. Write a program and show your name and roll number on screen?

3. Write a pseudocode to take employee's salary and age as an input from user and print them on console window.
4. Write a pseudo code to take two numbers as input from user and subtract them.
5. Take an integer input n and print n, its square (n2) and its cube (n3).
6. Calculate and print the surface area and volume for a box with dimensions (length, width, and height) entered by the user
   Volume=h × W × L
   Surface Area=2(h × W) + 2(h × L) + 2(W × L)
7. Program to convert temperature from Celsius to Fahrenheit
   T(°F) = T(°C) × 1.8 + 32
   °C = (°F - 32) × 1.8

xxx--------------xxxxx-----------------xxx

Experiment 3

Defining and implement different Data types, Data representation, Identifiers, Reserved words, Variables and constants.

Objective:

To understand and implement variables and reserved words.

Data Type and Data Representation:

In programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.
The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C. A byte can store a relatively small amount of data, one single character or a small integer (generally an integer between 0 and 255). In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Name	Description	Size*	Range*
`char`	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
`short int(short)`	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
`int`	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`long int (long)`	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`bool`	Boolean value. It can take one of two values: true or false.	1byte	`true` or `false`
`float`	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
`double`	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
`long double`	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)

Size	Unique representable values	Notes
8-bit	256	= 2^{8}
16-bit	65 536	= 2^{16}
32-bit	4 294 967 296	= 2^{32} (~4 billion)
64-bit	18 446 744 073 709 551 616	= 2^{64} (~18 billion billion)

Identifiers and Reserved Words:

A ***valid identifier*** is a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underline character (_), but in some cases these may be reserved for compiler specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. In no case can they begin with a digit.

Another rule that you have to consider when inventing your own identifiers is that they cannot match any keyword of the C language nor your compiler's specific ones, which are ***reserved keywords***. The standard reserved keywords are:

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while.

Additionally, alternative representations for some operators cannot be used as identifiers since they are reserved words under some circumstances:

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq

Your compiler may also include some additional specific reserved keywords.

Very important: The C language is a *"case sensitive"* language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the *RESULT variable* is not the same as the result variable or the Result variable. These are three different variable identifiers

Variables, Constants:

In order to use a variable in C, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier.
For example:

int **a;**
float **mynumber;**

These are two valid declarations of variables. The first one declares a variable of type *int* with the identifier a. The second one declares a variable of type float with the identifier *mynumber*. Once declared, the variables a and *mynumber* can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

int **a, b, c;**

This declares three variables (a, b and c), all of them of type int, and has exactly the same meaning as:

int **a;**
int **b;**
int **c;**

The integer data types char, short, long and int can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier signed or the specifier unsigned before the type name.
For example:

unsigned short int **NumberOfSisters;**
signed int **MyAccountBalance;**

By default, if we do not specify either signed or unsigned most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:

int **MyAccountBalance;**

With exactly the same meaning (with or without the keyword signed) An exception to this general rule is the char type, which exists by itself and is considered a different fundamental data type from signed char and unsigned char, thought to store characters. You should use either signed or unsigned if you intend to store numerical values in a char-sized variable.

short and long can be used alone as type specifiers. In this case, they refer to their respective integer fundamental types: short is equivalent to short int and long is equivalent to long int. The following two variable declarations are equivalent:

***short* Year;**
***short int* Year;**

Finally, signed and unsigned may also be used as standalone type specifiers, meaning the same as signed int and unsigned int respectively. The following two declarations are equivalent:

***unsigned* NextYear;**
***unsigned int* NextYear;**

He simplest use is to declare a named constant. This was available in the ancestor of C++, C.

To do this, one declares a constant as if it was a variable but add ‘const’ before it. One has to initialize it immediately in the constructor because, of course, one cannot set the value later as that would be altering it. For example,

***const int* my_var=96;**

will create an integer constant, unimaginatively called ‘my_var, with the value 96.

Such constants are useful for parameters which are used in the program but do not need to be changed after the program is compiled. It has an advantage for programmers over the C preprocessor ‘#define’ command in that it is understood & used by the compiler itself, not just substituted into the program text by the preprocessor before reaching the main compiler, so error messages are much more helpful.

EXERCISE

1. Write a program, in which you declare an integer type of variable with the defined values and show values which was assigned on screen:

    a. integer variable=4
    b. constant variable =6.

2. Write a program to take capital alphabet as input and show the smaller letter on the screen?

3. Write down the code in which you take 2-numbers as an input, by applying addition on them and show the result on screen?

xxx---------------xxxxx------------------xxx

Experiment 4

Arithmetic and logical Operators

Objective:

To understand what is arithmetical and logical operations in C/C++ and implement them.

Arithmetic and Logical Operators:

Operators are the foundation of any programming language. Thus the functionality of C/C++ programming language is incomplete without the use of operators. We can define operators as symbols that helps us to perform specific mathematical and logical computations on operands. In other words we can say that an operator operates the operands.
For example, consider the below statement:
c = a + b;
Here, '+' is the operator known as addition operator and 'a' and 'b' are operands. The addition operator tells the compiler to add both of the operands 'a' and 'b'. C/C++ has many built-in operator types and they can be classified as:

Arithmetic Operators:

These are the operators used to perform arithmetic/mathematical operations on operands.
Examples: (+, -, *, /, %,++,–). Arithmetic operator are of two types:

- **Unary Operators**:

  Operators that operates or works with a single operand are unary operators. For example: (++ , –)

- **Binary Operators:**

  Operators that operates or works with two operands are binary operators.

For example: (+ , – , * , /)

- **Relational Operators:**

  Relational operators are used for comparison of the values of two operands. For example: checking if one operand is equal to the other operand or not, an operand is greater than the other operand or not etc. Some of the relational operators are (==, > , = , <=).

- **Logical Operators:**

  Logical Operators are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a boolean value either true or false.

- **Bitwise Operators:**

  The Bitwise operators is used to perform bit-level operations on the operands. The operators are first converted to bit-level and then calculation is performed on the operands. The mathematical operations such as addition, subtraction, multiplication etc. can be performed at bit-level for faster processing.

- **Assignment Operators:**

  Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of variable on the left side otherwise the compiler will raise an error. Different types of assignment operators are shown below:
  "=": This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left. For example:
  a = 10;
  b = 20;
  ch = 'y';
  "+=": This operator is combination of '+' and '=' operators. This operator first adds the current value of the variable on left to the value on right and then assigns the result to the variable on the left.
  Example:
  (a += b) can be written as (a = a + b)

If initially value stored in a is 5. Then (a += 6) = 11.
"-=": This operator is combination of '-' and '=' operators. This operator first subtracts the current value of the variable on left from the value on right and then assigns the result to the variable on the left.
Example:
(a -= b) can be written as (a = a - b)
If initially value stored in a is 8. Then (a -= 6) = 2.
"*=": This operator is combination of '*' and '=' operators. This operator first multiplies the current value of the variable on left to the value on right and then assigns the result to the variable on the left.
Example:
(a *= b) can be written as (a = a * b)
If initially value stored in a is 5. Then (a *= 6) = 30.
"/=": This operator is combination of '/' and '=' operators. This operator first divides the current value of the variable on left by the value on right and then assigns the result to the variable on the left.
Example:
(a /= b) can be written as (a = a / b)
If initially value stored in a is 6. Then (a /= 2) = 3.

Other Operators:

Apart from the above operators there are some other operators available in C or C++ used to perform some specific task. Some of them are discussed here:

- **sizeof operator**:

sizeof is a much used in the C/C++ programming language. It is a compile time unary operator which can be used to compute the size of its operand. The result of sizeof is of unsigned integral type which is usually denoted by size_t. Basically, sizeof operator is used to compute the size of the variable.

- **Comma Operator:**

The comma operator (represented by the token ,) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type). The comma operator has the lowest precedence of any C operator. Comma acts as both operator and separator.

- **Conditional Operator:**

  Conditional operator is of the form Expression1 ? Expression2 : Expression3 . Here, Expression1 is the condition to be evaluated. If the condition(Expression1) is True then we will execute and return the result of Expression2 otherwise if the condition(Expression1) is false then we will execute and return the result of Expression3. We may replace the use of if..else statements by conditional operators.

Operator precedence chart

The below table describes the precedence order and associativity of operators in C / C++ . Precedence of operator decreases from top to bottom.

OPERATOR	DESCRIPTION	ASSOCIATIVITY
( )	Parentheses (function call)	left-to-right
[ ]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++/–	Postfix increment/decrement	
++/–	Prefix increment/decrement	right-to-left
+/-	Unary plus/minus	
!~	Logical negation/bitwise complement	
(type)	Cast (convert value to temporary value of type)	
*	Dereference	
&	Address (of operand)	
sizeof	Determine size in bytes on this implementation	
*,/,%	Multiplication/division/modulus	left-to-right
+/-	Addition/subtraction	left-to-right
<>	Bitwise shift left, Bitwise shift right	left-to-right
< , <=	Relational less than/less than or equal to	left-to-right
> , >=	Relational greater than/greater than or equal to	left-to-right
== , !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right

?:	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= , -=	Addition/subtraction assignment	
*= , /=	Multiplication/division assignment	
%= , &=	Modulus/bitwise AND assignment	
^= , |=	Bitwise exclusive/inclusive OR assignment	
<>=	Bitwise shift left/right assignment	
,	expression separator	left-to-right

EXERCISE

1. Write a program, in which you take 3 inputs from user and apply the operations, Add, subtract, divide and multiply.
2. Write a program to interchange values of two operands a and b.
3. Write a program for calculating the percentage of a student, by taking input of 5 subject's marks and each course have max 100 marks.
4. Write a program to interchange two variables values without using third variable.
5. Write a program to print the size of char, float, double and long double data types in C.
6. Write a program to input two numbers and display the maximum number.
7. Write a program to calculate the area and perimeter of circle, triangle, and square. Input is dependent on the type shape.
8. Write a program to find unit , ten , hundred , and thousand values in a four digit number

   Example

   Number = 1234

   Unit = 4

   Ten = 3

   Hundred = 2

   Thousand = 1
9. Write a program to find the sum of individual digits of an integer.
10. Write a program to reverse a four digit number

Example

Number = 1234

Reverse = 4321

11. What was the difference between a++, a= a+1? Discuss them?

xxx---------------xxxxx------------------xxx

Experiment 5

Conditional Statements

Objective:

To understand relational operators in C and how to make Decision Statements using if, if-else, ifelse if ladder and switch-case.

Decision Control Structure:

A simple C statement is each of the individual instructions of a program, like the variable declarations and expressions seen in previous sections. They always end with a semicolon (;), and are executed in the same order in which they appear in a program. But programs are not limited to a linear sequence of statements. During its process, a program may repeat segments of code, or take decisions and bifurcate. For that purpose, C provides flow control statements that serve to specify what has to be done by our program, when, and under which circumstances.
Many of the flow control statements explained in this section require a generic (sub) statement as part of its syntax. This statement may either be a simple C++ statement, such as a single instruction, terminated with a semicolon (;) or a compound statement. A compound statement is a group of statements (each of them terminated by its own semicolon), but all grouped together in a block, enclosed in curly braces: {}:

{ statement1; statement2; statement3; }

The entire block is considered a single statement (composed itself of multiple sub-statements). Whenever a generic statement is part of the syntax of a flow control statement, this can either be a simple statement or a compound statement.

Selection statements: if and else

The if keyword is used to execute a statement or block, if, and only if, a condition is fulfilled. Its syntax is:

if (condition) statement

Here, condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is not executed (it is simply ignored), and the program continues right after the entire selection statement.

For example:

the following code fragment prints the message (x is 100), only if the value stored in the x variable is indeed 100:

```
if (x == 100)
        printf("x is 100");
```

If x is not exactly 100, this statement is ignored, and nothing is printed.

If you want to include more than a single statement to be executed when the condition is fulfilled, these statements shall be enclosed in braces ({}), forming a block:

```
if (x == 100)
        {
                printf("x is”);
                printf("x”);
        }
```

As usual, indentation and line breaks in the code have no effect, so the above code is equivalent to:

```
if (x == 100) printf("x is "); printf(“x”); }
```

Selection statements with if can also specify what happens when the condition is not fulfilled, by using the else keyword to introduce an alternative statement. Its syntax is:

if (condition) statement1 else statement2

where statement1 is executed in case condition is true, and in case it is not, statement2 is executed.

For example:

```
if (x == 100)
        printf("x is 100");
```

```
else
printf( "x is not 100");
```

This prints x is 100, if indeed x has a value of 100, but if it does not, and only if it does not, it prints x is not 100instead.
Several if + else structures can be concatenated with the intention of checking a range of values.

For example:

```
if (x > 0)
        {
        Printf("x is positive");
        }
else if (x < 0)
        {
        Printf("x is negative");
        }
else
        {
        printf("x is 0");
        }
```

This prints whether x is positive, negative, or zero by concatenating two if-else structures. Again, it would have also been possible to execute more than a single statement per case by grouping them into blocks enclosed in braces: {}.

Nested if Statement:
It is always legal to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax:
The syntax for a nested if statement is as follows:

```
if( boolean_expression 1)
{
        // Executes when the boolean expression 1 is true
        if(boolean_expression 2)
         {
                // Executes when the boolean expression 2 is true
```

```
          }
}
```
You can nest else if...else in the similar way as you have nested if statement.

Example

```
int main ()
{
// local variable declaration:
int a = 100;
int b = 200;

// check the boolean condition
if( a == 100 )
{
// if condition is true then check the following
           if( b == 200 )
          {
          // if condition is true then print the following
          Printf( "Value of a is 100 and b is 200");
          }
}
Printf( "Exact value of a is :%d ",a);
Printf( "Exact value of b is :%d ",b );

return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

The Conditional Expression:

There are expressions of a special kind, the conditional expressions, these are not statements, but they are one sort of contraction of the if...then construct. This kind of expression can help to produce highly readable assignment statements fitting onto one line of the source code. This is the syntax (there is no semicolon at the end, since it is not a statement but an expression embeddable into even larger expressions):

(condition) ? expressionIfTrue : expressionIfFalse

First the condition is evaluated and the side effects of this evaluation carry out their impact on the local environment. If the result is true then only the expressionIfTrue is evaluated (causing side effects) and this second result is the value of the whole conditional expression, and the expressionIfFalse is not evaluated (and hence cause no side effects). If the condition evaluates to false, then the situation is converse, the resulting values is given by the evaluation of the false branch of the conditional expression, and the true branch is not evaluated.

Also known as Conditional (ternary) statement

Common use of the conditional expression is to assign the value x or y to a, depending on an easily decidable condition, say x>y.
Example:

```
int a = ( x > y ) ? x : y;

//this is equivalent to:

int a;
if (x > y)
{
        a = x;
}
else
{
        a = y;
}
```

As you can see, this makes simple conditionals all the simpler.

EXERCISE

1. Write a program for calculating the percentage and grade of a student by taking input of 5 subjects marks (100 marks each course max).

   - Less than 55 = fail
   - 55 to 65 = C+
   - 65 to 75 = B
   - 75 to 85 = B+
   - 85 to 95 = A
   - 95 to 100 = A+

2. Write a program to make a calculator where the operator selection is dependent upon user input. Following are the requirements.

   - two inputs numeric
   - one selection for operation
   - output with respect to user selection
   - When selecting division operator handle ‘0’ in denominator.
   - Operators (+,-,*,/)

3. Write a program to find whether the number entered by user is even or odd.

4. Write a program to check whether the number entered by the user is positive or negative.

6. Write a program to print “good morning” if time is less than 12 else print “good after noon”

7. Write a program to print “You can vote” if the voter’s age is 18 or above, otherwise print “You are not eligible”

8. What is Ternary operator? Why we use them?

____

____

____

Experiment 6

Switch statements

Objective:

To understand and implement the Switch case.

Decision Control Structure:

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case. But this will only work on assignment operator i.e. (==).

The syntax for a switch statement in C is as follows:

```
switch(expression)
{
    case constant-expression  :
    {
    statement(s);
    break; //optional
    }
    case constant-expression  :
    {
    statement(s);
    break; //optional
    }
    // you can have any number of case statements.
    default : //Optional
    {
    statement(s);
    }
}
```

The following rules apply to a switch statement:

- The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.

- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

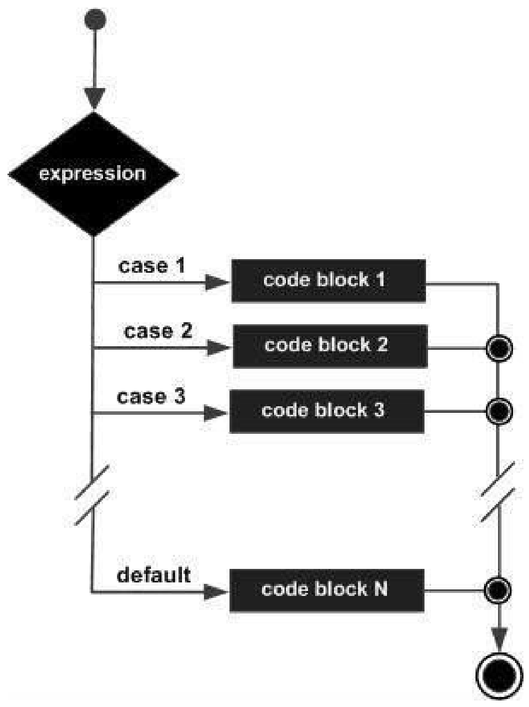
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.

- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.

- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.

- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.



Example:

```
int main ()
{
// local variable declaration:
```

```
char grade = 'D';

switch(grade)
{
        case 'A' :
        {
                Printf("Excellent!");
                break;
        }
        case 'C' :
        {
                Printf( "Well done" );
                break;
        }
        case 'D' :
        {
                Printf( "You passed" );
                break;
        }
        case 'F' :
        {
                Printf("Better try again");
                break;
        }
        default :
        {
                Printf(“Invalid grade");
        }
}
Printf( "Your grade is ");
return 0;
}
```

EXERCISE

1. Write a program to design a calculator that performs all the arithmetic operations (by Switch case)

Experiment 7

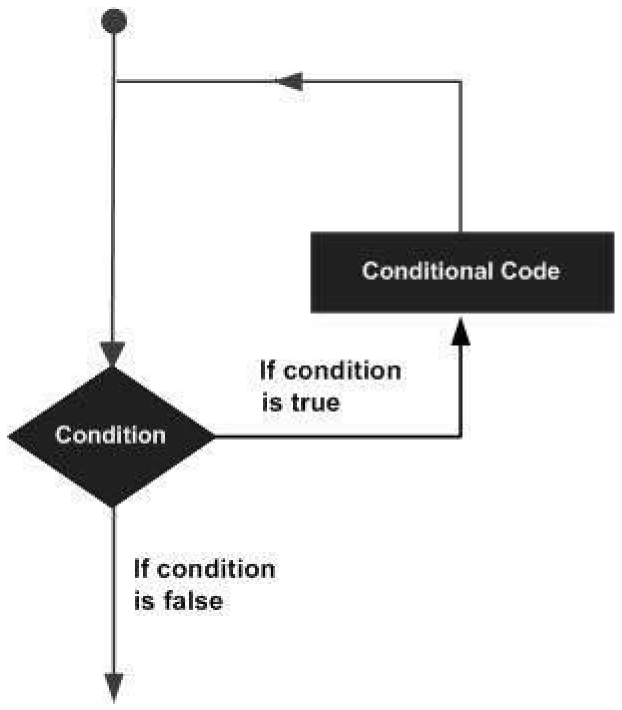
Loops

Objective:

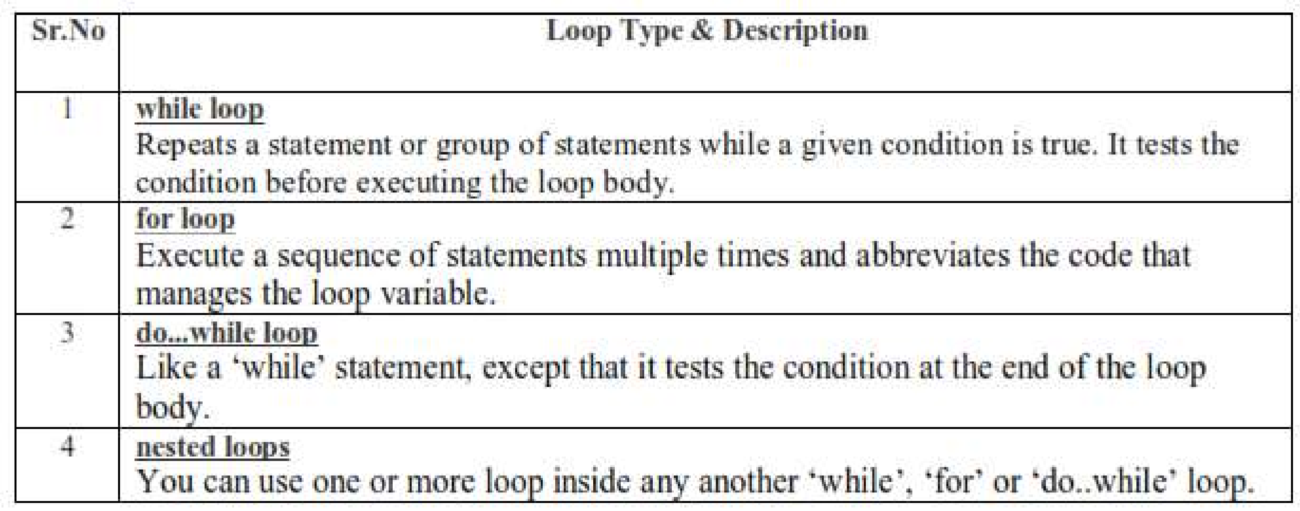
To understand and implement loop and function.

Loop Structure:

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
Programming languages provide various control structures that allow for more complicated execution paths.
A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages,



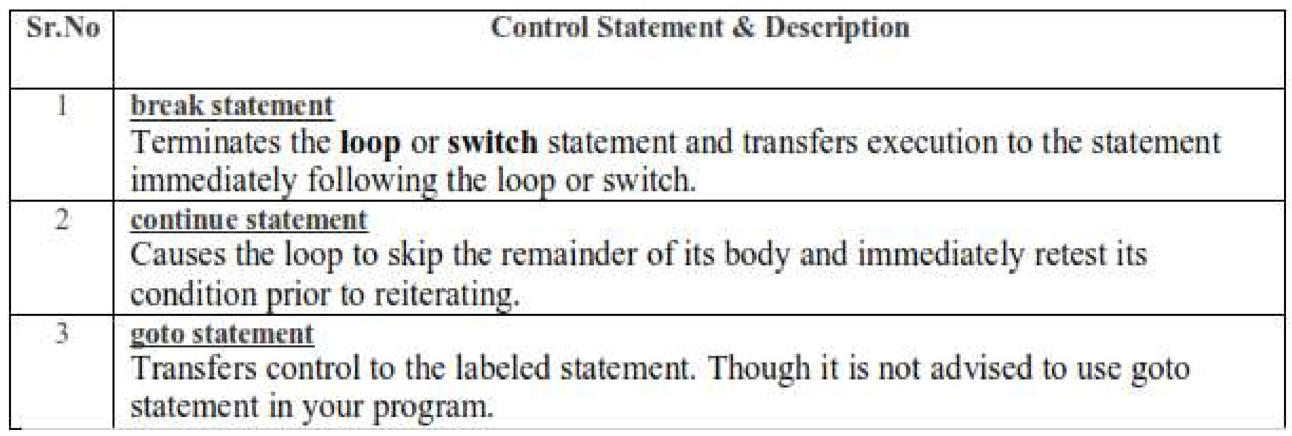
C programming language provides the following type of loops to handle looping requirements.



Sr.No	Loop Type & Description
1	<u>while loop</u> Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	<u>for loop</u> Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	<u>do...while loop</u> Like a 'while' statement, except that it tests the condition at the end of the loop body.
4	<u>nested loops</u> You can use one or more loop inside any another 'while', 'for' or 'do..while' loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
C supports the following control statements.



Sr.No	Control Statement & Description
1	<u>break statement</u> Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	<u>continue statement</u> Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	<u>goto statement</u> Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

For Loop:

A for loop is a repetition control structure which allows us to write a loop that is executed a specific number of times. The loop enables us to perform n number of steps together in one
line.

Syntax:

```
for (initialization expr; test expr; update expr)
{
// body of the loop
// statements we want to execute
}
```

In for loop, a loop variable is used to control the loop. First initialize this loop variable to some value, then check whether this variable is less than or greater than counter value. If statement is true, then loop body is executed and loop variable gets updated. Steps are repeated till exit condition comes.

Initialization Expression: In this expression we have to initialize the loop counter to some value. for example: int i=1;

Test Expression: In this expression we have to test the condition. If the condition evaluates to true then we will execute the body of loop and go to update expression otherwise we will exit from the for loop. For example: i <= 10;

Update Expression: After executing loop body this expression increments/decrements the loop variable by some value. for example: i++;

```
// C program to illustrate for loop

int main()
{
        for (int i = 1; i <= 10; i++)
        {
                Printf(“Checking\n”);
        }
return 0; }
```

While Loop:

While studying for loop we have seen that the number of iterations is known beforehand, i.e. the number of times the loop body is needed to be executed is known to us. While loops are used in situations where we do not know the exact number of iterations of loop beforehand. The loop execution is terminated on the basis of test condition.

Syntax:

We have already stated that a loop is mainly consisted of three statements – initialization expression, test expression, update expression. The syntax of the three loops – For, while and do while mainly differs on the placement of these three statements.

```
Initialization expression;

while (test_expression)
{
// statements
update_expression;
}
```

```
// C program to illustrate for loop

int main()
{
// initialization expression
 int i = 1;

// test expression
while (i < 6)
        {
                Print("Checking\n");
                // update expression
                i++;
        }
return 0;
}
```

Do While Loop:

In do while loops also the loop execution is terminated on the basis of test condition. The
main difference between do while loop and while loop is in do while loop the condition is tested at the end of loop body, i.e. do while loop is exit controlled whereas the other two loops are entry controlled loops.
Note: In do while loop the loop body will execute at least once irrespective of test condition.

Syntax:

```
initialization expression;
do
{
        // statements
        update_expression;
}
while (test_expression);
```

Note: Notice the semi – colon(“;”) in the end of loop.

```
// C program to illustrate do-while loop

int main()
{
int i = 2; // Initialization expression

do
        {
                // loop body
                Print("Checking\n");

                // update expression i++;

        }
while (i < 1);  // test expression
return 0;
}
```

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the ‘for’ loop are required, you can make an endless loop by leaving the conditional expression empty.

```
int main ()
{
for( ; ; )
        {
                printf("This loop will run forever.\n");
        }

return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the ‘for (;;)’ construct to signify an infinite loop.

Nested Loop:
Example: Code

```
  printf("\n\nNested loops are usually used to print a pattern in c. \n\n");
    printf("\n\nThey are also used to print out the matrix using a 2 dimensional array.
\n\n");

    int i,j,k;
    printf("\n\nOutput of the nested loop is :\n\n");
    for(i = 0; i < 5; i++)
    {
        printf("\t\t\t\t");
        for(j = 0; j < 5; j++)
        printf(" *");

        printf("\n");
    }
    printf("\n\n\t\t\tCoding is Fun !\n\n\n");
```

EXERCISE

1. Write a program for finding the factorial of a number. Number is user dependent.

2. Write a program by using loops: in which they take user input of number for table, and other input for their range.
   EX: input number for table of 2;
   Then put range 7: after that the table starts to print:

```
2x1=2
2x2=4
 ......
2x7=14
```

For Loop

1. Print following series
   5 7 9 11 13
   71 65 59 53 47
   1,3,9,27,81.....2187
2. Write a program to input two integer numbers and display the sum of even numbers between these two input numbers.

3. Write a program to count the even, odd and negative numbers in the list of input numbers?

While Loop

1. Print series:
   1 7 49 343 3401

2. Write a program to print Fibonacci series upto 15 terms?

3. Write a program to reverse a number?

4. Write a program to find the sum of 5 digits number (i.e. sum of individual number) using for loop statement

5. Write a program to count total digits in a given integer using loop

Do-While Loop

1. Write a program to perform division between two numbers till the user wants.

Nested Loop

Nested loops are usually used to print a pattern in C. They are also used to print out the matrix using a 2 dimensional array and a lot of other patterns like pyramid of numbers etc.

Below is a simple program on nested loops.

```
int main()
{
   printf("Here we used to print out the matrix using a 2 dimensional array. \n\n");

   int i,j;
   printf("\n\nOutput of the nested loop is :\n\n");
   for(i = 0; i < 5; i++)
   {
      printf("\t\t\t\t");
      for(j = 0; j < 5; j++)
                {
                printf("* ");
                }
      printf("\n");
   }

   return 0;
}
```

Example 2:

```
printf("Here we used to print out a 2 dimensional array. \n\n");

    int i,j;
    printf("\n\nOutput of the nested loop is :\n\n");
    for(i = 1; i <= 3; i++)
    {
        printf("\t\t\t\t");
        for(j = 1; j <= 5; j++)
                    {
                    printf("%d\t",i*j);
                    }
        printf("\n");
    }
```

xxx---------------xxxxx---------------xxx

Experiment 8

Functions

Objective:

To understand and implement function.

Functions:

A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.
You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.
A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, function strcat() to concatenate two strings, function memcpy() to copy one memory location to another location and many more functions.
A function is known with various names like a method or a sub-routine or a procedure etc.

Advantages:

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

FUNCTIONS TYPES:

1. Function with no argument and no Return value

Used where no data is being passed and no value will have returned.

2. Function with no argument and with a Return value

Used where the function will return an answer form where the function is being called.

3. Function with argument and No Return value

Used where the data is being passed for processing but the output will only returned in that particular function.

4. Function with argument and Return value

Used where the data is being passed and returned.

DEFINING A FUNCTION:

The general form of a C function definition is as follows −

```
return_type function_name ( parameter list )
 {
body of the function
}
```

A C function definition consists of a function header and a function body. Here are all the
parts of a function.

- **Return Type** − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- **Function Body** − The function body contains a collection of statements that define what the function does.

Example:

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and return the biggest of both,

```
// function returning the max between two numbers

int max(int num1, int num2) {
// local variable declaration int result;
if (num1 > num2)
result = num1;
else
result = num2;

return result;
}
```

FUNCTION DECLARATIONS:

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts

```
 return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

CALLING A FUNCTION:

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it’s return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example,

```
// function declaration
int max(int num1, int num2);

int main () {
// local variable declaration:
int a = 100; int b = 200; int ret;

// calling a function to get max value.
ret = max(a, b);
Printf( "Max value is : ",ret);

return 0;
}

// function returning the max between two numbers
int max(int num1, int num2) {
// local variable declaration
int result;

if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result,

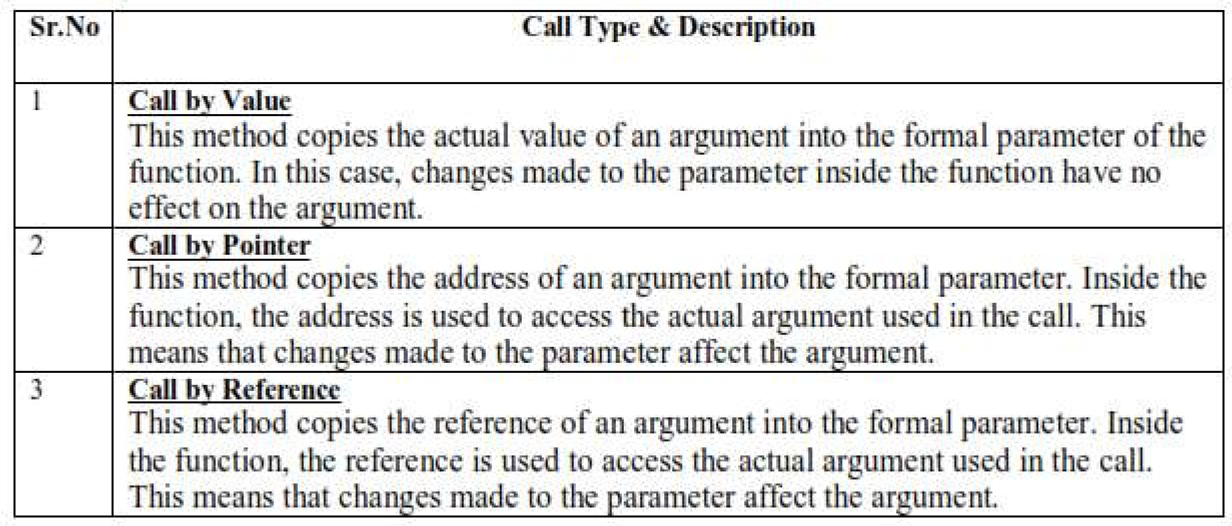
```
Max value is: 200
```

FUNCTION ARGUMENTS:

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are three ways that arguments can be passed to a function



Sr.No	Call Type & Description
1	<u>**Call by Value**</u> This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	<u>**Call by Pointer**</u> This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3	<u>**Call by Reference**</u> This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

<u>DEFAULT VALUES FOR PARAMETERS:</u>

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example,

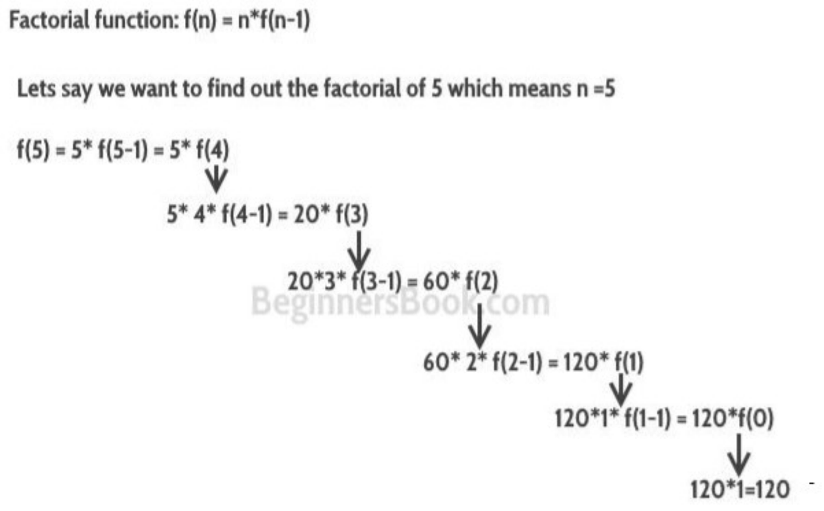
```
int sum(int a, int b = 20) {
int result;

result = a + b;
return (result);
}
int main () {
// local variable declaration:
int a = 100; int b = 200; int result;

// calling a function to add the values.
result = sum(a, b);
printf("Total value is :",result);

// calling a function again as follows.
result = sum(a);
printf( "Total value is :", result);

return 0;
}
```

FUNCTION RECURSION:

The process in which a function calls itself is known as recursion and the corresponding function is called the recursive function. The popular example to understand the recursion is factorial function.

Factorial function: f(n) = n*f(n-1), base condition: if n<=1 then f(n) = 1.
Don’t worry we will discuss what is base condition and why it is important.

In the following diagram. I have shown that how the factorial function is calling itself until the function reaches to the base condition


Factorial function: f(n) = n*f(n-1)
Lets say we want to find out the factorial of 5 which means n =5
f(5) = 5* f(5-1) = 5* f(4)
5* 4* f(4-1) = 20* f(3)
20*3* f(3-1) = 60* f(2)
BeginnersBook.com
60* 2* f(2-1) = 120* f(1)
120*1* f(1-1) = 120*f(0)
120*1=120

Let’s solve the problem using C program.

RECURSION EXAMPLE: FACTORIAL

```
//Factorial function
int f(int n){
/* This is called the base condition, it is very important to specify the base condition
in recursion, otherwise your program will throw stack overflow error.
*/
if (n <= 1)
return 1;
else
return n*f(n-1);
}
int main(){
int num;
printf("Enter a number: ");
scanf(“%d”,&num);
printf("Factorial of entered number: ",f(num));
return 0;
}
```

Output:

```
Enter a number: 5
Factorial of entered number: 120
```

BASE CONDITION:

In the above program, you can see that I have provided a base condition in the recursive function. The condition is:

```
if (n <= 1)
return 1;
```

The purpose of recursion is to divide the problem into smaller problems till the base condition is reached. For example in the above factorial program I am solving the factorial function f(n) by calling a smaller factorial function f(n-1), this happens repeatedly until the n value reaches base condition(f(1)=1). If you do not define the base condition in the recursive function then you will get stack overflow error.

EXERCISE

1. Write a program to swap two numbers using the function.
2. Write a program for calculating the power of a number take a number and his power as an input, and perform this task by using function and show the output. Example: 2 power 4 so the answer is=16
3. Write a function power (a, b), to calculate the value of a raised to b.
4. Write a program in C to convert decimal number to binary number using the function.
5. Write a program to calculate factorial of a number using recursion.
6. Write a program to calculate sum of first 50 natural numbers using recursive function.

xxx--------------xxxxx---------------xxx

Experiment 9

Arrays

Objective:

To understand and implement the array.

Arrays:

C provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.
All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays:

To declare an array in C, the programmer specifies the type of the elements and the number of elements required by an array as follows,

type arrayName [arraySize];

This is called a single-dimension array. The arraySize must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 10-element array called balance of type double, use this statement,

double balance[10];

Initializing Arrays:

You can initialize C array elements either one by one or using a single statement as follows,

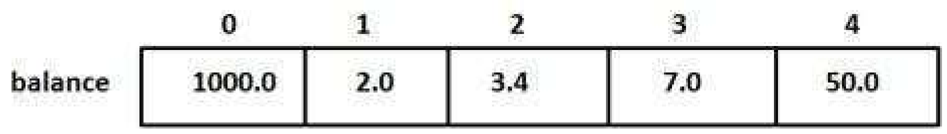
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array,

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write,

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};

You will create exactly the same array as you did in the previous example. balance[4] = 50.0;
The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above



Accessing Array Elements:

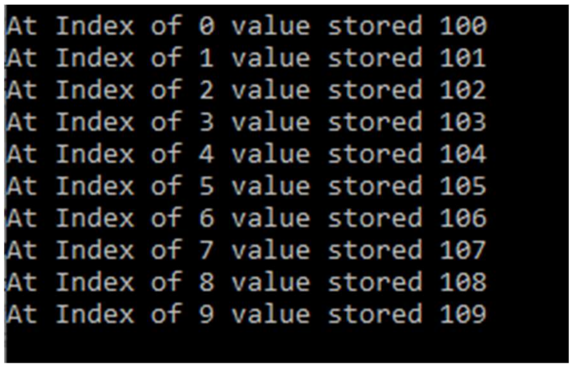
An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts. declaration, assignment and accessing arrays,

```
int main () {

int n[ 10 ]; // n is an array of 10 integers

// initialize elements of array n to 0
for ( int i = 0; i < 10; i++ ) {
n[ i ] = i + 100; // set element at location i to i + 100
}

// output each array element's value
for ( int j = 0; j < 10; j++ ) {
printf("At Index of %d value stored %d", j, n[ j ]);
}

return 0;
}
```

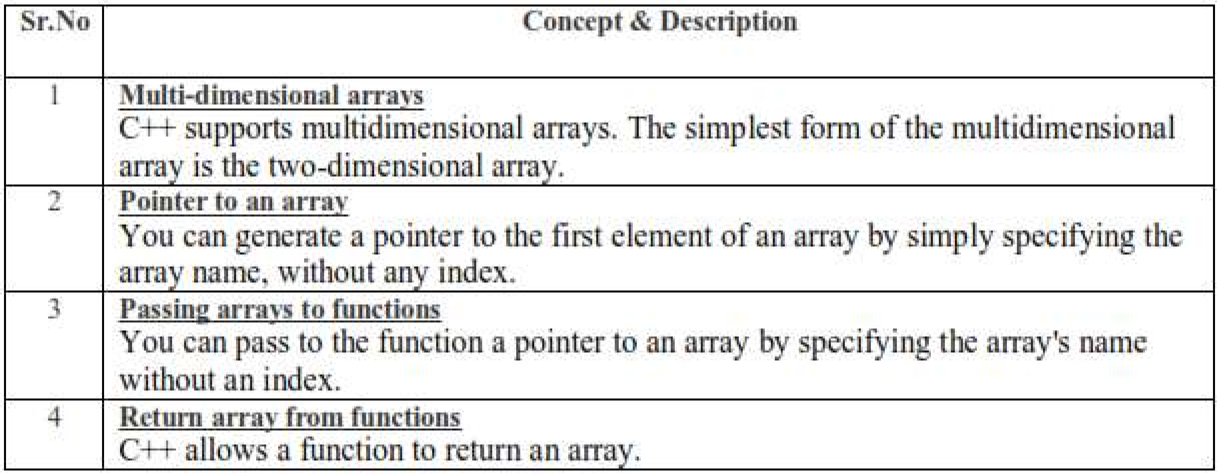
When the above code is compiled and executed, it produces the following result



```
At Index of 0 value stored 100
At Index of 1 value stored 101
At Index of 2 value stored 102
At Index of 3 value stored 103
At Index of 4 value stored 104
At Index of 5 value stored 105
At Index of 6 value stored 106
At Index of 7 value stored 107
At Index of 8 value stored 108
At Index of 9 value stored 109
```

ARRAYS TYPES:

Arrays are important to C & C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C & C++ programmer



Sr.No	Concept & Description
1	**Multi-dimensional arrays** C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	**Pointer to an array** You can generate a pointer to the first element of an array by simply specifying the array name, without any index.
3	**Passing arrays to functions** You can pass to the function a pointer to an array by specifying the array's name without an index.
4	**Return array from functions** C++ allows a function to return an array.

EXERCISE

1. Write a program and take 5 integers as an input in array, find the sum and product of that elements by using for loop.

2. Write a program and take 10 integers as an input from user and stored them in array, print them on screen.
NOTE: WITHOUT USING ANY LOOP.

3. Write a program for multi-dimensional array, show the output in the form of Matrices and perform multiplication of 2d array,

Data is:

Array 1:
[2, 1]
[3, 4]

Array 2:
[3, 4]
[5, 1]

4. Write a program to Search an element in array.

5. Write a program to perform addition of all elements in array.

6. Write a program to find the largest and smallest element in array.

7. Write a program to reverse the array elements in C.

8. Write a Program to delete an element from the specified location in array.

9. Write a Program to access an element in 2-D array.

10. Write a program to perform addition between two matrices.

11. Write a program to read a sentence & replace all ‘a’ with ‘A’.

xxx--------------xxxxx---------------xxx

Experiment 10

Pointers, Dynamic memory allocation, ragged arrays

Objective

To understand and implement arrays with pointers.

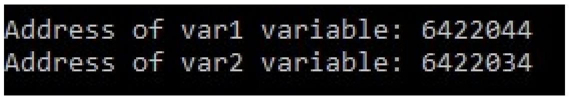
Pointers

C pointers are easy and fun to learn. Some C tasks are performed more easily with pointers, and other C tasks, such as dynamic memory allocation, cannot be performed without them.

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined,

```
int main () {
int  var1;
char var2[10];
printf("Address of var1 variable: %d\n",&var1);
printf(""Address of var2 variable: %d\n",&var2);
return 0;
}
```

When the above code is compiled and executed, it produces the following result



```
Address of var1 variable: 6422044
Address of var2 variable: 6422034
```

What Are Pointers?

A pointer is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is,
type *var-name;

Here, type is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration,

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

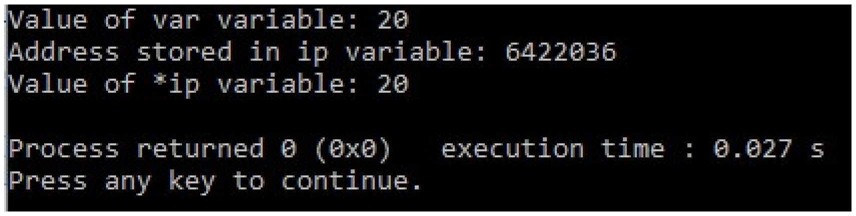
Using Pointers In C

There are few important operations, which we will do with the pointers very frequently.

(a) We define a pointer variable.
(b) Assign the address of a variable to a pointer.
(c) Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations,

```
int main () {
int  var = 20;   // actual variable declaration.
int  *ip;        // pointer variable
ip = &var;       // store address of var in pointer variable
printf( "Value of var variable: %d",var);

// print the address stored in ip pointer variable
printf("Address stored in ip variable: %d",ip);

// access the value at the address available in pointer
Printf( "Value of *ip variable: ",*ip);

return 0;
}
```

When the above code is compiled and executed, it produces result something as follows,


```
Value of var variable: 20
Address stored in ip variable: 6422036
Value of *ip variable: 20

Process returned 0 (0x0)   execution time : 0.027 s
Press any key to continue.
```

Pointers In C

Pointers have many but easy concepts and they are very important to C programming. There are following few important pointer concepts which should be clear to you as a C programmer,

Sr.No	Concept & Description
1	**Null Pointers** C supports null pointer, which is a constant with a value of zero defined in several standard libraries
2	**Pointer Arithmetic** There are four arithmetic operators that can be used on pointers: ++, --, +, -
3	**Pointers vs Arrays** There is a close relationship between pointers and arrays.
4	**Array of Pointers** You can define arrays to hold a number of pointers.
5	**Pointer to Pointer** C allows you to have pointer on a pointer and so on.
6	**Passing Pointers to Functions** Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function.
7	**Return Pointer from Functions** C allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well.

Dynamic Memory

In the programs seen in previous chapters, all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C language integrates the operators new and delete.

Operators New And New[]

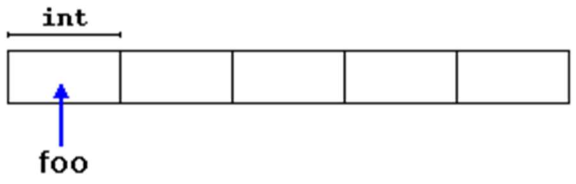
Dynamic memory is allocated using operator new. new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. Its syntax is:

```
pointer = new type
pointer = new type [number_of_elements]
```

The first expression is used to allocate memory to contain one single element of type type. The second one is used to allocate a block (an array) of elements of type type, where number_of_elements is an integer value representing the amount of these.
For example:

```
int * foo;
foo = new int [5];
```

In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to foo (a pointer).
Therefore, foo now points to a valid block of memory with space for five elements of type int.



Here, foo is a pointer, and thus, the first element pointed to by foo can be accessed either with the expression foo[0]or the expression *foo (both are equivalent). The second element can be accessed either with foo[1] or *(foo+1), and so on...

There is a substantial difference between declaring a normal array and allocating dynamic memory for a block of memory using new. The most important difference is that the size of a regular array needs to be a constant expression, and thus its size has to be determined at the moment of designing the program, before it is run, whereas the dynamic memory allocation performed by new allows to assign memory during runtime using any variable value as size.

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, there are no guarantees that all requests to allocate memory using operator new are going to be granted by the system.

C provides two standard mechanisms to check if the allocation was successful:

One is by handling exceptions. Using this method, an exception of type bad_alloc is thrown when the allocation fails. Exceptions are a powerful C feature explained later in these tutorials. But for now, you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.

This exception method is the method used by default by new, and is the one used in a declaration like:

```
foo = new int [5]; // if allocation fails, an exception is thrown
```

The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a null pointer, and the program continues its execution normally.

This method can be specified by using a special object called nothrow, declared in header <new>, as argument for new:

```
foo = new (nothrow) int [5];
```

In this case, if the allocation of this block of memory fails, the failure can be detected by checking if foo is a null pointer:

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
// error assigning memory. Take measures.
}
```

This nothrow method is likely to produce less efficient code than exceptions, since it implies explicitly checking the pointer value returned after each and every allocation. Therefore, the exception mechanism is generally preferred, at least for critical allocations. Still, most of the coming examples will use the nothrow mechanism due to its simplicity.

Operators Delete And Delete[]

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator delete, whose syntax is:

```
delete pointer;
delete[] pointer;
```

Ragged Arrays

This specification will use the C notation for arrays. For example, { a, b c } will denote a size-3 array with elements a, b, and c.
A simple example of a ragged array is
x = { { a }, { b, c, d } }
The ragged array x is two dimensional and has size 2, with elements that are 1D arrays of sizes 1 and 3 respectively. The valid indexes and values are:
x[1] = { a }
x[2] = { b, c, d }
x[1,1] = a
x[2,1] = b
x[2,2] = c
x[2,3] = d
and sizes are size(x) = 2 size(x[1]) = 1 size(x[2]) = 3
This concept (and notation) generalizes to any dimensionality of nested arrays, and to arrays of vectors and matrices of different sizes.

Declaring And Accessing Ragged Arrays

Rather than introducing new keywords, the existing array and matrix declarations will be generalized. New I/O will be needed within the Stan dump parser, var_context base class, and for all the interfaces. This will probably interact with the command refactoring that Michael's undertaking.
Currently, arrays are of fixed rectangular sizes and declared giving the size of each dimension.

For example,
int a[3, 2];
real b[I, J, K];
declares a as a (3 x 2) array of integers and b as a (I x J x K) array of continuous values, where I, J, and K are expressions denoting single integer values
The proposed generalizing will allow arrays to be declared using (ragged) arrays of integers.
For example, suppose the declarations are
int<lower=1> K;
int<lower=1> dims[K];
real x[dims];
with K = 2 and dims = { 1, 3 }. Then
x = { { a },{ b, c, d } }

EXERCISE

1. Write a program to find the address of two integer variables
2. Give two separate statements that assign the starting address of array values to pointer variable vPtr.
3. Write a program in C to add two numbers using pointers
4. Write a program to find biggest among three numbers using pointer.
5. Write a program to swap value of two variables using pointer
6. Write a program to perform division between two variables using pointer
7. Write a program and show the memory address of 10 indexes, array data type was integer and shows the addresses of particular array on screen?
8. Write a program with two integer variables and two pointer variables, then take input value in pointer variable but stored them in *integer variable?
*indirect accessing method.
9. Write a program for taking array[10] inputs from user and stored the particular values by using pointer, use for loop for indexing of array increment *a=a+1 where a is array 1st index address. And print accordingly.
*this will vary according to datatype memory consumption use indirect memory accessing method.

xxx--------------xxxxx---------------xxx

Experiment 11

Searching and Sorting Algorithm

Objective:

To understand and implement Algorithms for sorting and searching.

Searching In Array

The simplest type of searching process is the sequential search. In the sequential search, each element of the array is compared to the key, in the order it appears in the array, until the first element matching the key is found. If you are looking for an element that is near the front of the array, the sequential search will find it quickly. The more data that must be searched, the longer it will take to find the data that matches the key using this process.

```
//sequential search routine (non-function ... used in main)

int main(void)
{
int array[10];
//"drudge" filling the array
array[0]=20; array[1]=40; array[2]=100; array[3]=80; array[4]=10; array[5]=60;
array[6]=50; array[7]=90; array[8]=30; array[9]=70;
printf( "Enter the number you want to find (from 10 to 100)…");
int key;
scanf("%d", &key);
int flag = 0;    // set flag to off
int i=0;

for(i=0; i<10; i++)    // start to loop through the array
{
if (array[i] == key)   // if match is found
{
flag = 1;   // turn flag on
break;    // break out of for loop
}
}
```

```
if (flag==1)    // if flag is TRUE (1)
{
Printf( "Your number is %d at subscript position %d \n",array[i],i);
}
else
{
Printf( "Sorry, I could not find your number in this array.");
}
return 0;
```

SORTING OF ARRAY

Sort an array elements means arrange elements of array in Ascending Order and Descending Order. You can easily sort all elements using sorting algorithm.

C Program to Sort Elements of Array in Ascending Order:

```
void main()
{
int i,a[10],temp,j;
printf("Enter any 10 num in array: \n");
for(i=0;i<=10;i++)
{
Scanf(“%d”,a[i];
}
Printf("\nData before sorting: \n");
for(j=0;j<10;j++)
{
Printf(“%d”,a[j]);
}
for(i=0;i<=10;i++)
{
        for(j=0;j<=10-i;j++)
        {
        if(a[j]>a[j+1])
        {
        temp=a[j];
        a[j]=a[j+1];
        a[j+1]=temp;
        }
        }
}
Printf("\nData after sorting: ");
for(j=0;j<10;j++)
{
Printf("%d",a[j]);
}
}
```

EXERCISE

1. Write a program to perform searching in array, in which you take 10 integers input from user and stored them in array. Ask a user to give a number which you they want to search and show the output if number is present in an array or not.
2. Write a program to perform sorting in array of limit 5, take 5 integers as an input form user and sort them in ascending order and show the sorted result on screen.
3. One[10]={0,3,4,5,8,6,9,1,7,2};
    1. Search two elements e1 and e2
    2. Interchange their positions
    3. Delete element from index [e2-e1]
    4. Insert 10 on position [e1+3]
    5. Print array in ascending order using Bubble sort Algorithm

4. O1[8] = {0,1,2,3,4,5,6,7};
   O2[8] = {1,3,5,7,9,2,4,6};
    1. Pick odd position elements from array O1 and put them on even positions of array O2
    2. Search elements from array O2 which are greater than 5 and insert 0's on these positions
    3. Print array O2 in ascending order using Selection sort Algorithm

xxx---------------xxxxx----------------xxx

Experiment 12

Structures, Structure declaration, accessing structure members, array of structures

Objective:

To understand and implement the structure.

Structure:

C arrays allow you to define variables that combine several data items of the same kind, but structure is another user defined data type which allows you to combine data items of different kinds.

Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book

- Title
- Author
- Subject
- Book ID

Defining Structure:

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member, for your program. The format of the struct statement
is this,

```
struct [structure tag] {
member definition;
member definition;
...
member definition;
} [one or more structure variables];
```

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure,

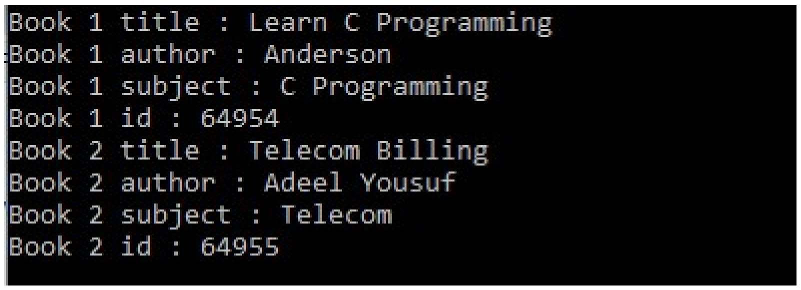
```
struct Books {
char title[50];
char author[50];
char subject[100];
int  book_id;
} book;
```

Accessing Structure Members:

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use struct keyword to define variables of structure type. Following is the example to explain usage of structure,

```
struct Books {
char  title[50];
char  author[50];
char  subject[100];
int   book_id;
};

int main() {
struct Books Book1;       // Declare Book1 of type Book
struct Books Book2;       // Declare Book2 of type Book

// book 1 specification
strcpy( Book1.title, "Learn C Programming");
strcpy( Book1.author, "Anderson");
strcpy( Book1.subject, "C Programming");
Book1.book_id = 64954;
// book 2 specification
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Adeel Yousuf");
strcpy( Book2.subject, "Telecom");
Book2.book_id = 64955;

// Print Book1 info
printf( "Book 1 title : %s\n",Book1.title);
printf( "Book 1 author : %s\n",Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 id : %d\n",Book1.book_id);

// Print Book2 info
printf( "Book 2 title : %s\n",Book2.title );
printf( "Book 2 author : %s\n",Book2.author);
printf( "Book 2 subject : %s\n",Book2.subject);
printf( "Book 2 id : %d\n",Book2.book_id);

return 0;
}
```

When the above code is compiled and executed, it produces the following result.



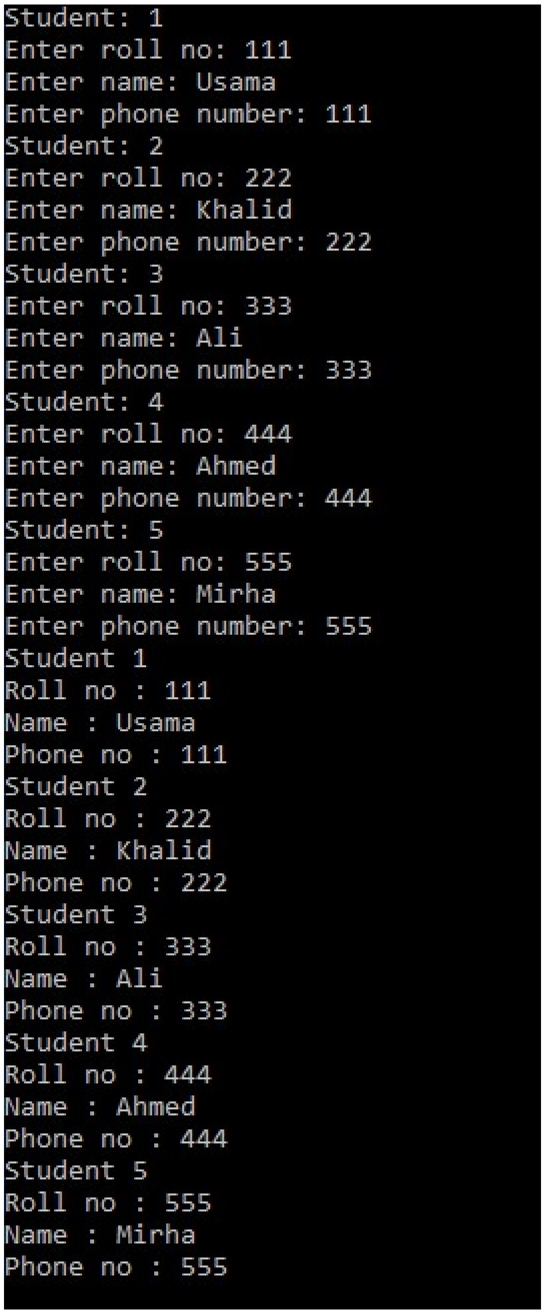
```
Book 1 title : Learn C Programming
Book 1 author : Anderson
Book 1 subject : C Programming
Book 1 id : 64954
Book 2 title : Telecom Billing
Book 2 author : Adeel Yousuf
Book 2 subject : Telecom
Book 2 id : 64955
```

Array Structure:

We can also make an array of structures. Now suppose we need to store the data of 100 children. Declaring 100 separate variables of the structure is definitely not a good option. For that, we need to create an array of structures.
Let's see an example for 5 students.

```
struct student
{
int roll_no;
char name[20];
int phone_number;
};

int main(){

struct student stud[5];
int i;

for(i=0; i<5; i++){            //taking values from user
printf("Student: %d\n", i+1);
printf("Enter roll no: ");
scanf("%d",&stud[i].roll_no);
printf("Enter name: ");
scanf("%s",&stud[i].name);
printf("Enter phone number: ");
scanf("%d",&stud[i].phone_number);
}

for(i=0; i<5; i++){       //Printing Values
//printing

printf("Student %d\n",i+1);
printf("Roll no : %d\n",stud[i].roll_no);
printf("Name : %s\n",stud[i].name);
printf("Phone no : %d\n",stud[i].phone_number);

}
return 0;
}
```

Output



```
Student: 1
Enter roll no: 111
Enter name: Usama
Enter phone number: 111
Student: 2
Enter roll no: 222
Enter name: Khalid
Enter phone number: 222
Student: 3
Enter roll no: 333
Enter name: Ali
Enter phone number: 333
Student: 4
Enter roll no: 444
Enter name: Ahmed
Enter phone number: 444
Student: 5
Enter roll no: 555
Enter name: Mirha
Enter phone number: 555
Student 1
Roll no : 111
Name : Usama
Phone no : 111
Student 2
Roll no : 222
Name : Khalid
Phone no : 222
Student 3
Roll no : 333
Name : Ali
Phone no : 333
Student 4
Roll no : 444
Name : Ahmed
Phone no : 444
Student 5
Roll no : 555
Name : Mirha
Phone no : 555
```

Here we created an array named stud having 5 elements of structure student. Each of the element stores the information of a student. For example, stud[0] stores the information of the first student, stud[1] for the second and so on.
We can also copy two structures at one go.

```
struct student
{
int roll_no;
char name[20];
int phone_number;
};
```

```
int main(){

struct student p1 = {1,"Abc",123443};
struct student p2;
p2 = p1;

printf( "roll no :  %d", p2.roll_no);
printf( "name :  %s",p2.name);
printf( "phone number :  %d",p2.phone_number);
return 0;
}
```

Exercise

1. Make a Program using structure, and take input of four books each includes.

Book Title, Author name, Book subject, Book id

show the output of each book on screen.

2. Create a structure named company which has: name, address, phone and noOfEmployee as member variables. Read name of company, its address, phone and noOfEmployee. Finally display these members value.

3. Write a program to print details of student which include id, name, department, marks of five subjects and address of student using nested structure.

4. Write a program to display details of three employees by using structures with arrays.

5. Write a program to generate electricity bill by using structure in which they take user inputs of the following,

Unit consumption, Per unit rate, Address, Username, Month

Calculate the bill and show the output on screen.

xxx--------------xxxxx---------------xxx

Experiment 13

Passing structures as function arguments

Objective:

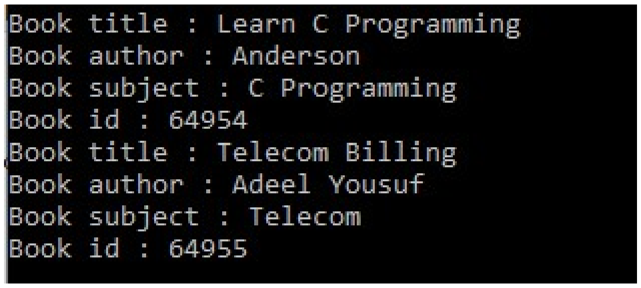
To understand and implement the structure as an input argument.

Structures As Function Arguments:

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above examples.

```
struct Books {
char  title[50];
char  author[50];
char  subject[100];
int   book_id;
};
int main() {
struct Books Book1;       // Declare Book1 of type Book
struct Books Book2;       // Declare Book2 of type Book
// book 1 specification
strcpy( Book1.title, "Learn C Programming");
strcpy( Book1.author, "Anderson");
strcpy( Book1.subject, "C Programming");
Book1.book_id = 64954;
// book 2 specification
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Adeel Yousuf");
strcpy( Book2.subject, "Telecom");
Book2.book_id = 64955;
// Print Book1 info
printBook(Book1);
// Print Book2 info
printBook(Book2);
return 0;
}
void printBook( struct Books book )
{
printf("Book title : %s\n",book.title);
printf("Book author : %s\n",book.author);
printf("Book subject : %s\n",book.subject);
printf("Book id : %d\n",book.book_id);
}
```

When the above code is compiled and executed, it produces the following result



```
Book title : Learn C Programming
Book author : Anderson
Book subject : C Programming
Book id : 64954
Book title : Telecom Billing
Book author : Adeel Yousuf
Book subject : Telecom
Book id : 64955
```

The Typedef Keyword:

There is an easier way to define structs or you could "alias" types you create. For example

```
typedef struct {
```

```
char  title[50];

char  author[50];

char  subject[100];

int   book_id;

} Books;
```

Now, you can use Books directly to define variables of Books type without using struct keyword. Following is the example,

```
Books Book1, Book2;
```

You can use typedef keyword for non-structs as well as follows,

```
typedef long int *pint32;

pint32 x, y, z;
```

x, y and z are all pointers to long ints.

Exercise

1. Write a program for courier service dispatching system in which they take user input of following,

    Sender name
    Sender address
    Receiver name
    Receiver address
    Per kg price
    Total weight

passing the structure data in function and then calculate the bill and show the output on screen.

xxx--------------xxxxx---------------xxx

Experiment 14

File Handling (Read / Write)

Objective:

To understand and implement File handling.

File Handling:

If the program requires the input data either 1 or 2, the user could supply the data at any time. But if the details of the student in a very large institution are required to prepare marks statement for all, we have to enter the whole details every time to our program. This requires large man power, more time & computational resources; which is a disadvantage with the traditional way of giving inputs to a program for execution with the basic I/O functions.
When the data of the students are stored permanently, the data can be referred later and makes the work easier for the user to generate the marks statement. When you have to store data permanently, we have to use FILES. The files are stored in disks.

DEFINITION:

A file can be defined as a collection of bytes stored on the disk under a name. A file may contain anything, in the sense the contents of files may be interpreted as a collection of records or a collection of lines or a collection of instructions etc.
A file is a collection of records. Each record provides information to the user. These files are arranged on the disk.

Basic operations on files include:

- Open a file
- Read data from file/Write data to file
- Close a file

To access the data in a file using C we use a predefined structure **FILE** present in stdio.h header file, that maintains all the information about files we create (such as pointer to char in a file, end of file, mode of file etc).
FILE is a data type which is a means to identify and specify which file you want to operate on because you may open several files simultaneously. When a request is

made for a file to be opened, what is returned is a pointer to the structure FILE. To store that pointer, a pointer variable is declared as follows:
FILE *file_pointer;
where file_pointer points to first character of the opened file.

Opening of a file:

A file needs to be opened when it is to be used for read/write operation. A file is opened by the fopen() function with two parameters in that function. These two parameters are file name and file open mode.

Syntax: fopen(filename, file_open_mode);

The fopen function is defined in the “stdio.h” header file. The filename parameter refers to any name of the file with an extension such as “data.txt” or “program.c” or "student.dat" and so on.
File open mode refers to the mode of opening the file. It can be opened in ‘read mode’or ‘write mode’ or ‘append mode’.
The function fopen() returns the starting address of file when the file we are trying to open is existing (i.e. success) else it returns NULL which states the file is not existing or filename given is incorrect.
E.g.:

```
FILE *fp;
 fp=fopen("data.txt","r");
 /*If file exists fp points to the starting address in memory
 Otherwise fp becomes NULL*/
 if(fp==NULL)
printf("No File exists in Directory");
```
NULL is a symbolic constant declared in stdio.h as 0.

Modes of Operation:

1. "r" (read) mode: open file for reading only.
2. "w" (write) mode: open file for writing only.
3. "a" (append) mode: open file for adding data to it.
4. "r+" open for reading and writing, start at beginning
5. "w+" open for reading and writing (overwrite file)
6. "a+" open for reading and writing (append if file exists)

When trying to open a file, one of the following things may happen:

- When the mode is ‘writing’ a file with the specified name is created if the file does not exist. The contents are deleted, if the file is already exists.
- When the purpose is ‘appending’, the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
- When the purpose is ‘reading’, and if it exists, then the file is opened with the current contents safe; otherwise an error occurs.

With these additional modes of operation (mode+), we can open and use a number of files at a time. This number however depends on the system we use.

Closing a file:

A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all the links to the file are broken. It also prevents any accidental misuse of file.
Closing of unwanted files might help open the required files. The I/O library supports a function to do this for us.

Syntax: fclose(file_pointer);
/*This would close the file associated with the FILE pointer file_pointer*/

Syntax: fcloseall();
/*This would close all the opened files. It returns the number of files it closed. */

Example: Program to check whether given file is existing or not.

```
#include<stdio.h>
main()
{
FILE *fp;
fp=fopen("data.txt","r");
if(fp==NULL) {
printf("No File exists in Directory");
exit(0);
}
else
printf("File Opened");
fclose(fp);
getch();
}
```

Reading and Writing Character on Files:

To write a character to a file, the input function used is putc or fputc. Assume that a file is opened with mode 'w' with file pointer fp. Then the statement,

```
putc(character_variable,file_pointer);
fputc(character_variable,file_pointer);
```

writes the character contained in the 'character_variable' to the file associated with FILE pointer 'file_pointer'.

E.g.:

```
 putc(c,fp);
 fputc(c,fp1);
```

To read a character from a file, the output function used is getc or fgetc. Assume that a file is opened with mode 'r' with file pointer fp. Then the statement,

```
character_variable = getc(file_pointer);
character_variable = fgetc(file_pointer);
```

read the character contained in file whose FILE pointer 'file_pointer' and stores in 'character_variable'.

E.g.:

```
getc(fp);
fgetc(fp);
```

The file pointer moves by one character position for every operation of getc and putc. The getc will return an end-of-file marker EOF, when end of the file has been reached. Therefore, the reading should be terminated when EOF is encountered.

Example: Program for Writing to and reading from a file

```
#include<stdio.h>
main()
{
FILE *f1;
char c;
clrscr();
printf("Write data to file: \n");
f1 = fopen("test.txt","w");
while((c=getchar())!='@')
putc(c,f1);
/*characters are stored into file until '@' is encountered*/
fclose(f1);
printf("\nRead data from file: \n");
f1 = fopen("test.txt","r"); /*reads characters from file*/
while((c=getc(f1))!=EOF)
printf("%c",c);
fclose(f1);
getch();
}
```

Example: Program to write characters A to Z into a file and read the file and print the characters in lowercase.

```
#include<stdio.h>
main()
{
FILE *f1;
char c;
clrscr();
printf("Writing characters to file... \n");
f1 = fopen("alpha.txt","w");
for(ch=65;ch<=90;ch++)
fputc(ch,f1);
fclose(f1);
printf("\nRead data from file: \n");
f1 = fopen("alpha.txt","r");
/*reads character by character in a file*/
while((c=getc(f1))!=EOF)
printf("%c",c+32); /*prints characters in lower case*/
fclose(f1);
getch();
}
```

Example: Program to illustrate the append mode of operation for given file.

```
#include<stdio.h>
main()
{
```

```
FILE *f1;
char c,fname[20];
clrscr();
printf("Enter filename to be appended: ");
gets(fname);
printf("Read data from file: \n");
f1 = fopen(fname,"r");
if(f1==NULL)
{
printf("File not found!");
exit(1);
}
else
{
while((c=getc(f1))!=EOF)
printf("%c",c);
}
fclose(f1);
f1 = fopen(fname,"a");
while((c=getchar())!='@')
putc(c,f1);
/*characters are appended into file with existing data until
'@' is encountered*/
fclose(f1);
getch();
}
```

Reading and Writing Integers getw and putw functions:

These are integer-oriented functions. They are similar to getc and putc functions and are used to read and write integer values. These functions would be useful when we deal with only integer data.

The general forms of getw and putw are:

putw: Writes an integer to a file.

```
putw(integer,fp);
```

getw: Reads an integer from a file.

```
getw(fp);
```

Example: Program to illustrate getw and putw functions.

```
#include<stdio.h>
main()
{
int n,num,i,sum=0;
FILE *fp;
clrscr();
printf("Enter the number of integers to be written to file: ");
scanf("%d",&n);
fp = fopen("numbers.txt","w");
for(i=0;i<n;i++)
{
scanf("%d",&num);
putw(num,fp);
}
fclose(f1);
fp = fopen("numbers.txt","r");
while((num=getw(fp))!=EOF)
{
sum=sum+num;
}
fclose(fp);
printf("Sum of %d numbers is %d\n\n",n,sum);
printf("Average of %d numbers is %d",sum/n);
getch();
}
```

End of File feof():

The feof() function can be used to test for the end of the file condition. It takes a FILE
pointer as its only argument and returns a non-zero integer value if all the data from specified file has been read and returns zero otherwise.
This function returns true if you reached end of file in given file, otherwise returns false.

```
while(!feof(fp))
{
....
....
....
}
```

/*executes set of statements in the loop until EOF is encountered*/

fgets and fputs functions:

fputs writes string followed by new line into file associated with FILE pointer fp.
fputs(char *st, FILE *fp);
E.g.: fputs(str,fp);
fgets reads characters from current position until new line character is encountered or len-1 bytes are read from file associated with FILE pointer fp. Places the characters read from the file in the form of a string into str.
fgets(char *st,int len,FILE *fp);
E.g.: fgets(str,50,fp);

Example: Program to illustrate fgets and fputs function.

```
#include<stdio.h>
main()
{
FILE *fp;
char s[80];
clrscr();
fp=fopen("strfile.txt","w");
printf("\nEnter a few lines of text:\n");
while(strlen(gets(s))>0)
{
fputs(s,fp);
fputs("\n",fp);
}
fclose(fp);
printf("Content in file:\n");
fp=fopen("strfile.txt","r");
while(!feof(fp))
{
puts(s);
fgets(s,80,fp);
}
fclose(fp);
getch();
}
```

fprintf and fscanf functions:

The functions fprintf and fscanf perform I/O operations that are identical to the familiar printf and scanf functions, except of course that they work on files. The first argument of these functions is a file pointer, where specifies the file to be used.
fprintf(): Writes given values into file according to the given format. This is similar to printf but writes the output to a file instead of console.
fscanf(): Reads values from file and places values into list of arguments. Like scanf it returns the number of items that are successfully read. When the end of file is reached, it returns the value EOF. The general form of fprintf and fscanf is

fprintf(file_pointer/stream, “control string”, list);
fscanf(file_pointer/stream, “control string”, list);

where file_pointer is a pointer associated with a file that has been opened for writing. The stream refers to a stream or sequence of bytes that can be associated with a device or a file. The following are the available standard file pointers/streams:

- stdin refers to standard input device, which is by default keyboard.
- stdout refers to standard output device, which is by default screen.
- stdprn refers to printer device.
- stderr refers to standard error device, which is by default screen.

The control string (format specifier) contains output specifications for the items in the list.
The list may include variables, constants and strings.
E.g.:

```
fprintf(fp, “%s %d %f”, name, age, 7.5); /*writes to file*/
fprintf(stdout, “%s %d %f”, name, age, 7.5); /*prints data on console*/
fscanf(fp, “%s %d %f”, name, &age, &per); /*reads from file*/
fscanf(stdin, “%s %d %f”, name, &age, &per); /*reads data from console*/
```

Example: Program to illustrate fprintf and fscanf function.

```
#include<stdio.h>
main()
{
FILE *fp;
int number,quantity,i;
float price,value;
char item[20],filename[10];
clrscr();
printf("Enter filename: ");
gets(filename);
fp=fopen(filename,"w");
printf("Enter Inventory data: \n");
printf("Enter Item Name, Number, Price and Quantity
(3 records):\n");
for(i=1;i<=3;i++) {
fscanf(stdin,"%s %d %f %d",item,&number,&price,&quantity);
fprintf(fp,"%s %d %0.2f %d",item,number,price,quantity);
}
fclose(fp);
fprintf(stdout,"\n\n");
fp=fopen(filename,"r");
printf("Item Name\tNumber\tPrice\tQuantity\tValue\n");
for(i=1;i<=3;i++)
{
fscanf(fp,"%s %d %f %d",item,&number,&price,&quantity);
value=price*quantity;
fprintf(stdout,"%s\t%10d\t%8.2f\t%d\t%11.2f\n",item,number,price,
quantity,value);
}
fclose(fp);
getch();
}
```

EXERCISE

1. Practice all above examples in lab
2. Write a program to write your name into file named 'abc.txt'.
3. Write a program to read your name from file named 'abc.txt'
4. Write name, age and height of a person into a data file "person.txt" and read it (use fgets() and fputs() functions)

xxx--------------xxxxx-----------------xxx

All The Best ☺