Introduction to Computing and AI Lab CS101L

Lab Manual

2023

Faculty of Computer Science and Engineering, GIKI

I. Table of Contents

Contents

CS101L - Introduction to Computing and AI Lab

Lab Outline

Sania Akhtar – PhD Scholar
Faculty of Computer Science and Engineering (FCSE)
Ghulam Ishaq Khan Institute of Engineering Science and Technology (GIKI)

CS101L - Introduction to Computing and AI Lab

II. Lab Outline

CS101L – Introduction to Computing and AI Lab
Pre-Requisite: NO **Instructor: Ms. Sania Akhtar** **Email:** *sania.akhtar@giki.edu.pk* **Office #** F08(B Block), third floor FCSE. **Office Hours:** 11:00 am ~ 5:00 pm

Lab Introduction

This course will introduce you to the field of computer science and the fundamentals of computer programming. CS101 is specifically designed for students with no prior programming experience and touches upon a variety of fundamental topics. This course uses C++ programming platform to understand the principles of basic computer language. We do this by demonstrating C++ primitive data types, relational operators, logical operators, control statements, iterative statements, arrays, recursion, pointers, debugging, exception handling. By the end of the course, you will understand the basics of computer science and the C++ programming language. The principles you learn here will be developed further as you progress through the computer science discipline.

Mapping of CLOs and PLOs

Sr. No	Course Learning Outcomes+	PLOs*	Blooms Taxonomy
CLO_1	Utilize the basic techniques of C++programming language.	P L O 1	P2 (Set)
CLO_2	Implement programming structures to design solutions for the given problems.	PLO 1	P3 (Guided Response)
	+Please add the prefix "Upon successful completion of this course, the student will be able to" *PLOs are for BS (CE) only		

CLO Assessment Mechanism (Tentative)

Assessment tools	CLO_1	CLO_2
Lab Performance	**50%**	**50%**
Project	-	**15%**
Midterm Exam	**20%**	-
Final Exam	**30%**	**35%**

Overall Grading Policy (Tentative)

Assessment Items	Percentage
Lab Performance	**35%**
Midterm Exam	**30%**
Project	**5%**
Final Exam	**30%**

Text and Reference Books

Textbooks:

- D.S. Malik, "C++ Programming: from Problem Analysis to Program Design", 7th Edition.
- Lab Manual for CS101L

CS101L - Introduction to Computing and AI Lab

Lecture Breakdown	
Lab	**Contents/Topics**
Lab#01:	Introduction to C++ compiler and Installation guide, Basics of a C++ program, Algorithm Design, Data Types, Variables, Inputs/Outputs and Arithmetic Operators
Lab#02:	Control Structures I - Relational Operators, Logical Operators, If else, Multiple IF else, Nested if else
Lab#03:	Control Structures II – Iterative Structures (While Loop, Do While)
Lab#04:	Switch statement, case, break statement, default, for loop.
Lab#05:	User defined Functions I– Predefined Functions, User-defined Functions, Value Returning Functions
Lab#06:	User defined Functions II - Void Functions, Value Parameters, Reference Variables as parameters, Pass by Value and Pass by reference parameters
	Mid Term Exams
Lab#07:	User Defined Functions III – Reference parameter and value-returning functions, Scope of an identifier, Global variables, named constants and side effects, Static and automatic variables, Function overloading, Default parameters.
Lab#08:	Arrays & Strings – Arrays (Accessing array components, Processing one-dimensional arrays, array index out of bound, array initialization during declaration, partial initialization of arrays during declaration, some restrictions on array processing, arrays as parameters to functions, constant arrays as formal parameters, base address of an array and array in computer memory, functions cannot return a value of type array, other ways to declare arrays), Searching an array for a specific item, C-Strings (Character Arrays), Two and Multi-Dimensional Arrays.
Lab#09:	Recursion
Lab#10:	Pointers, Static Arrays, Dynamic Arrays, Two-Dimensional Arrays using Pointers.
Lab#11:	File Handling
Lab#12	Open-Ended Lab
	Final Exam

III. System Requirements

HARDWARE REQUIREMENT

- Core i3 or Above
- 2 GB RAM
- 20 GB HDD

SOFTWARE REQUIREMENT

- Visual Studio

CS101L - Introduction to Computing and AI Lab

Lab Evaluation and Rubrics

Sania Akhtar – PhD Scholar
Faculty of Computer Science and Engineering (FCSE)
Ghulam Ishaq Khan Institute of Engineering Science and Technology (GIKI)

IV. Evaluation Rubrics

	Weekly Evaluation	
Rubrics	**Marks (10)**	
Code writing	1	
Error/exception Handling	2	
Code Comments / Necessary Documentation	**3**	
Correctness	**4**	

Detailed Rubric of weekly Evaluation:

Code writing (No credit, if not relevant implementation):

* Indentation: 0.5
* Meaningful/relevant variable/function names: 0.5

Error/exception Handling:

* Runs without exception: 1
* Interactive menu: 0.5
* Displays proper messages/operations clearly: 0.5

Code Comments / Necessary Documentation:

* Knowledge about the topic 2
* clarify meaning where needed 1

Correctness:

* Accurate methodology 2
* All tasks are done: 1
* Passes all test cases:

V. Instruction For Students

- Students are expected to install lab software on their personal computers.
- Students are expected to read respective lab contents before attending it.
- Since the contents of a lab are covered in class before practicing in the lab, students are expected to read, understand and run all examples and sample problems before attending the lab.
- Reach the lab on time. Latecomers shall not be allowed to attend the lab.
- Students need to maintain 100% attendance in lab if not a strict action will be taken.
- Wear your student card in the lab.
- Mobile phones, USBs, and other electronic devices are strictly prohibited in the lab.
- In case of hardware/power issues, immediately inform the lab attendant and do not attempt to fix it by yourself.
- In case of queries during lab task implementation, raise your hand, inform your instructor, and wait for your turn.

CS101L - Introduction to Computing and AI Lab

Lab # 01

Introduction to Generative AI and C++

Learning Objectives

- To learn the basic concepts of Generative AI
- Creative Writing with Generative AI
- Large Language Model (LLM's), Code Generation and Art Creation
- Installation of IDE (Visual Studio)
- Introduction to Algorithms

Outcomes:

- Able to understand the concept of Generative AI
- Should be able to write prompts and get the desired results
- Get familiar with IDE

CS101L - Introduction to Computing and AI Lab

1.1 GENERATIVE AI

Generative AI refers to a subset of artificial intelligence techniques and models that focus on generating new data or content that is similar to existing data. These models learn patterns and structures from a given dataset and use that knowledge to create new, original content.

Experiment 1: Creative Writing with Generative AI

Generative AI is a valuable tool for creative writing, content drafting and generating ideas. There are several ways to use generative AI for creative writing. Some of them are:

1. **Prompt Generation:**
   In prompt generation, user provides a general topic or theme, and AI suggests creative and unique prompts. Generative models like GPT-3 are commonly used to generate writing prompts.
2. **Idea Expansion:**
   Generative AI can be used to expand and explain basic ideas given by the user.
3. **Character and World Building:**
   When creating characters or fictional worlds, users can generate backstories, personalities, and characteristics using generative AI.
4. **Dialogue Generation:**
   Generative AI can be used to create natural-sounding dialogues between characters. In this way users can experiment with different tones and find the right voice for each character.
5. **Descriptive Writing:**
   In descriptive writing, users provide a basic description of settings, scenes, or emotions to AI and it helps them to expand and enhance the writing.
6. **Poetry:**
   Generative AI can be used to create poetry by generating lines, stanzas, or even entire poems.
7. **Collaboration:**
   Users can work with AI as a co-writer. They can write a paragraph or a few sentences, and then have the AI continue the text. In this way, unexpected and creative ideas can be generated.
8. **Creative Prompts for writer's block:**
   Generative AI can be used to generate random ideas or sentences. In this way, users' creativity can be enhanced, and new ideas can be generated.
9. **Editing and Rewriting:**
   Users can use Generative AI to suggest alternative phrases, sentence structures, or ways to rephrase their writing and improve the flow and style of their work.
10. **Generating short stories:**
    Generative AI can be used to continue the stories from a starting point gives by users. In this way, it can create short stories.

In this experiment, we shall use Generative AI for Content writing and Summarization.

1.2 Content Writing

Content writing is the process of creating and publishing written content for a variety of purposes, including marketing, education, and entertainment.

We can use Generative AI for content writing. For this, we can give prompts to a tool like GPT-3, and it creates content.

1.3 Content Writing using Chat GPT:

IM THE BEST ADVENTURE DESTINATIONS FOR THE ULTIMATE VACATIONS

When seeking the best adventure destinations for the ultimate vacations, the world offers a plethora of options that cater to different interests and preferences. Here are some top adventure destinations known for their thrilling experiences and breathtaking landscapes:

1. **New Zealand**: Known as the adventure capital of the world, New Zealand offers a wide range of outdoor activities. From bungee jumping in Queenstown to hiking in the stunning Fiordland National Park and exploring geothermal wonders in Rotorua, New Zealand is an adventurer's paradise.

Content Writing without using Chat GPT:

1. New Zealand

New Zealand isn't called the adventure capital of the world for nothing! New Zealand is the land of adventure and you can check off most of your travel bucket list dreams here. We took part in pretty much every adventure that we could dream of trying during our trips to New Zealand and can honestly say, this is the best place on earth for adventures.

From whitewater rafting class 5 rapids in freezing glacial waters to magnificent glacier walks and bunjy jumping to skydiving it's the adrenaline junkies dream destination. With beaches, watersports, wildlife and glaciers, New Zealand has it all.

- Check out all the Things to do in New Zealand

CS101L - Introduction to Computing and AI Lab

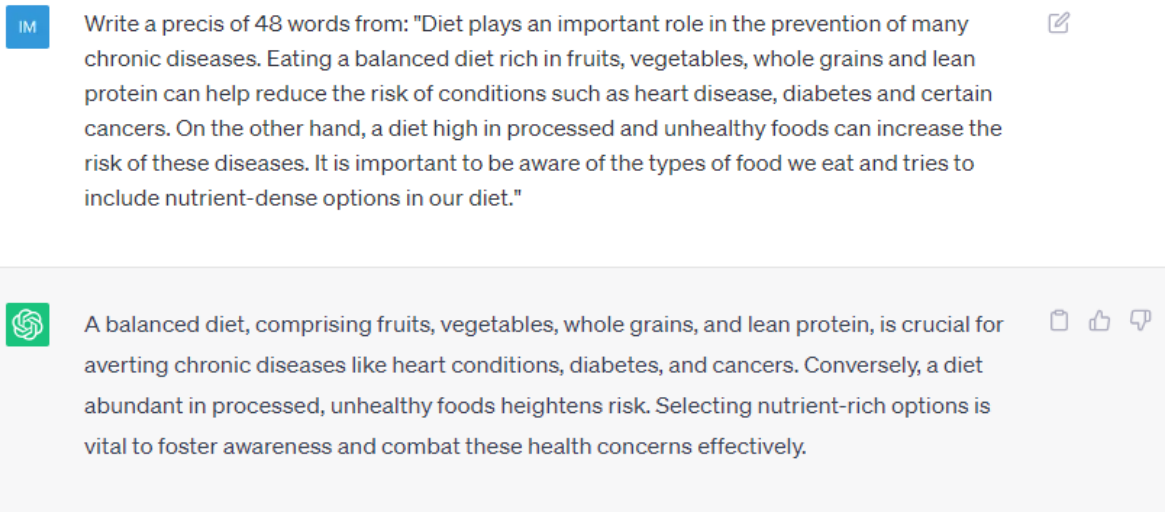
1.4 Precis Writing

Precis writing is summarising a comprehension in limited words, covering all the important aspects and details of the passage given.

We can use Generative AI for precis writing by providing it the original passage and number of words required in precis to tools such as Chat GPT.

In the example below, we shall write precis with a word limit of 48 words. The original word count is 124.

Precis Writing Using Chat GPT:



Precis Writing without using Chat GPT:

Original Passage: Diet plays an important role in the prevention of many chronic diseases. Eating a balanced diet rich in fruits, vegetables, whole grains and lean protein can help reduce the risk of conditions such as heart disease, diabetes and certain cancers. On the other hand, a diet high in processed and unhealthy foods can increase the risk of these diseases. It is important to be aware of the types of food we eat and tries to include nutrient-dense options in our diet. **(124 words)**

Precis: Eating a balanced diet rich in fruits, vegetables, whole grains and lean protein can help reduce the risk of chronic diseases, while a high-processed and unhealthy diet can increase the risk. It is important to be aware of the types of food we eat and include nutrient-dense options.(48 words)

TASK 01:

Create a prompt for large language Model, such as ChatGPT, to describe how large language models works. Ask at least three follow up questions based on output. Provide the Prompts and outputs from the conversation.

TASK 02:

Create A Prompt for large language Model, such as Chat GPT, that asks it to perform an instruction other than answering a question. For example, ask ChatGPT to create a meal plan. plan a trip. create a recipe, or help you diagnose a problem. Provide the Prompts and outputs from the conversation.

CS101L - Introduction to Computing and AI Lab

1.5 Introducing New Information to the Large Language Model

Large language models (ChatGPT) are going to be trained up to some point and then there's going to be a cut-off and they don't know what's happened after that cutoff.
The most important thing we can do in a prompt is provide the large language model with information that it doesn't have access to.

I apologize for any confusion, but I don't have access to real-time data and cannot provide information about the current number of birds outside your home. If you're interested in knowing the current bird count, I recommend looking out the window or going outside to observe the birds in your area. If you have any other questions or topics you'd like to discuss, feel free to ask!

Prompt

CS101L - Introduction to Computing and AI Lab

DATA Generated

Based on the data you've provided and the fact that it's currently March, we can estimate the number of birds outside your house by calculating the average increase in bird count from January to February, and then applying that average increase to the March count. Here's the calculation:

Average increase from January to February: (150 - 120) / 1 = 30 birds
Estimated increase from February to March: (210 - 150) / 1 = 60 birds

Now, applying the average increase to the March count:

Estimated bird count for March = March count + Estimated increase from February to March
Estimated bird count for March = 210 + 60 = 270 birds

So, based on the provided data and calculations, the estimated number of birds outside your house in March is 270 birds. Please remember that this is a simple estimation and actual bird populations can be influenced by various factors.

Task 03

Create A Prompt for large language Model, such as Chat GPT, that Introduces new information since 2021. Please only us public information. The prompt should require the large language model to perform a task, other than answering a question, with the information. Please provide your prompt additional information and output.

CS101L - Introduction to Computing and AI Lab

Experiment 2: Code Generation with Generative AI

Generative AI can be used to generate code snippets, scripts, or even entire programs based on given input or prompts. This can be helpful for automating repetitive coding tasks, generating prototypes, or assisting developers in writing code more efficiently. One popular approach is to use OpenAI's GPT-3 model to generate code snippets.

In this experiment, we shall use Generative AI for code generation.

Selection Sort

Sorting:

Sorting refers to the process of arranging a collection of elements (such as an array or a list) in a specific order, typically in ascending or descending order based on a certain key or comparison criteria. Sorting is a fundamental operation in computer science and is used extensively in various applications, such as databases, search algorithms, and data analysis.

Selection Sort:

Selection sort divides the list into two parts:

- The sorted part
- The unsorted part

It repeatedly finds the minimum (or maximum) element from the unsorted part and swaps it with the first element of the unsorted part.

The time complexity of selection sort is O(n^2) i.e., Quadratic Time Complexity.

We shall use Chat GPT to generate code for Selection sort in python programming language.

Selection Sort code using Chat GPT:

```
import time.

def selection_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

# Generate a list of random numbers for sorting
numbers = [9, 2, 7, 5, 1, 8, 6]

# Measure selection sort execution time
start_time = time.time()
selection_sort(numbers)
end_time = time.time()
execution_time = end_time - start_time

print("Sorted array:", numbers)
print(f"Selection sort execution time: {execution_time:.12f} seconds")
```

Output:

```
Sorted array: [1, 2, 5, 6, 7, 8, 9]
Selection sort execution time: 0.000079393387 seconds
```

Selection Sort without using Chat GPT:

```
import time

def selection_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

# Generate a list of random numbers for sorting
numbers = [9, 2, 7, 5, 1, 8, 6]

# Measure selection sort execution time
num_iterations = 1000
total_execution_time = 0

for _ in range(num_iterations):
    start_time = time.time()
    selection_sort(numbers.copy())
    end_time = time.time()
    execution_time = end_time - start_time
    total_execution_time += execution_time

average_execution_time = total_execution_time / num_iterations

print("Sorted array:", numbers)
print(f"Average selection sort execution time: {average_execution_time:.8f} seconds")
```

Output:

```
Sorted array: [1, 2, 5, 6, 7, 8, 9]
Average selection sort execution time: 0.00000644 seconds
```

Problem:

The execution time of generated code is longer compared to standard code.
There is not a 100% guarantee of the accuracy of its responses.
ChatGPT can give ambiguous responses when its codebase does not contain enough information on the requested topic.
Therefore, we should not rely entirely on ChatGPT-generated, instead we can seek assistance from these LLMs.

Task 01.

Create a Prompt for code generation of programming concepts. Provide the Prompt and output.

CS101L - Introduction to Computing and AI Lab

Experiment 3 : Art Creation with Generative AI

Generative AI is a fascinating tool for creating various forms of art, ranging from visual art to music and more. There are several ways for using Generative AI for art creation. Some of them are:

1. **Visual Art:**
   Generative models can be trained to create paintings, drawings, 3D art, sculptures, architectural designs etc. They can transform images using artistic styles, thus creating unique visual interpretations by transferring the style of one image onto another. They can even mimic the style of famous artists. Tools like Stable Diffusion Web are commonly used for Visual Art generation.
2. **Music Composition:**
   Generative AI can compose melodies, harmonies, and even chord progressions. It can generate music in different genres and styles, offering artists and musicians to come up with creative ideas and to experiment beyond their usual boundaries.
3. **Literature and Poetry:**
   Generative AI models like GPT-3 can help create poetry, short stories, dialogues, and more based on given prompts. Writers can co-create content with AI, resulting in unexpected and intriguing combinations of human and machine creativity.
4. **Fashion and Design:**
   Generative AI can offer suggestions for interior design layouts, furniture arrangements, colour schemes and can also generate new fashion designs, patterns, and 3D models of clothing items.
5. **Digital Art and Animation:**
   Generative AI can automate the process of creating animations, visual effects, digital illustrations, characters, and scenes, helping artists bring their ideas to life and enhancing the efficiency of animation production.
6. **Interactive Art:**
   Generative AI can generate virtual worlds, landscapes, and interactive elements. Moreover, AI powered interactive installations can respond to user input or environment factors, creating dynamic and evolving art experiences.

In this experiment, we shall use Generative AI for Art creation.

Mona Lisa:

The Mona Lisa is a half-length portrait painting by of Italian Artist ‘Leonardo da Vinci’. The painting has been definitively identified to depict Italian noblewoman Lisa del Giocondo. It has been described as the best known, the most visited, the most written about, the most sung about, and the most parodied work of art in the world.

CS101L - Introduction to Computing and AI Lab

Original Moni Lisa

CS101L - Introduction to Computing and AI Lab

1.6 Algorithm

An algorithm is a procedure having well defined steps for solving a particular problem. Algorithms are a finite set of logic or instructions, written in order for accomplish the certain predefined task. It is not the complete program or code, it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudo code.

The performance of algorithm is measured based on following properties:

Time complexity: It is a way of representing the amount of time needed by a program to run to completion.

Space complexity: It is the amount of memory space required by an algorithm, during the course of its execution. Space complexity is required in situations when limited memory is available and for the multiuser system.

Each algorithm must have:

Specification: Description of the computational procedure.

Pre-conditions: The condition(s) on input.

Body of the Algorithm: A sequence of clear and unambiguous instructions.

Post-conditions: The condition(s) on output.

Example: Design an algorithm to multiply the two numbers x and y and display the result in z.

Step 1 START

Step 2 declare three integers x, y

& z Step 3 define values of x & y

Step 4 multiply values of x & y

Step 5 store the output of step 4

in z Step 6 print z

Step 7 STOP

Alternatively, the algorithm can be written as ?

Step 1 START MULTIPLY

Step 2 get values of x & y

Step 3 z← x * y

Step 4 display z

Step 5 STOP

Characteristics of an Algorithm

An algorithm must follow the mentioned below characteristics:

Input: An algorithm must have 0 or well-defined inputs.

Output: An algorithm must have 1 or a well-defined output and should match with the desired output.

Feasibility: An algorithm must be terminated after a finite number of steps.

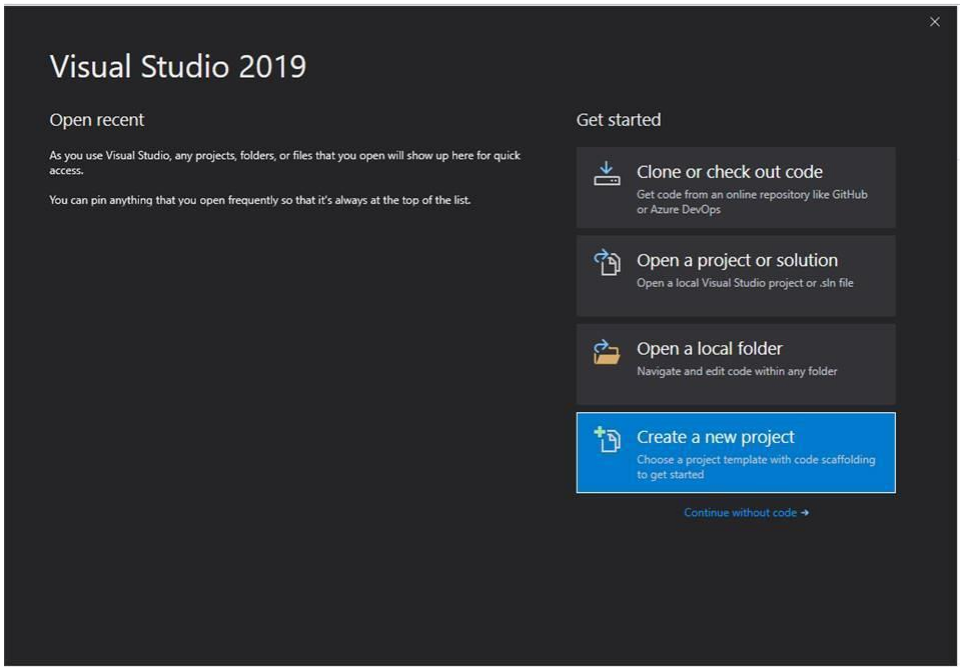
Independent: An algorithm must have step-by-step directions which is independent of any programming code.

Unambiguous: An algorithm must be unambiguous and clear. Each of their steps and input/outputs must be clear and lead to only one meaning.

CS101L - Introduction to Computing and AI Lab

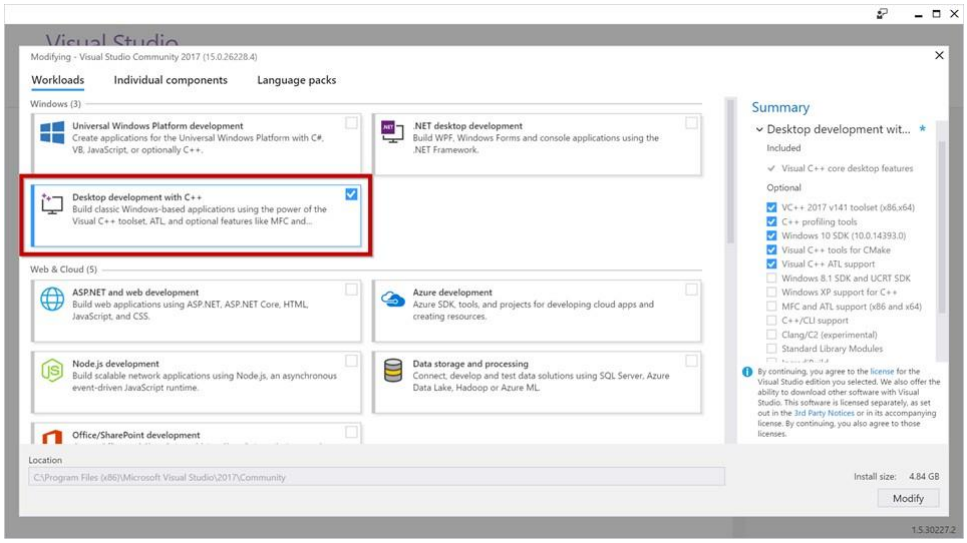
Hello World in C++ using Visual Studio 2019

1. First, understand the layout and views once you launch Visual Studio 2019.

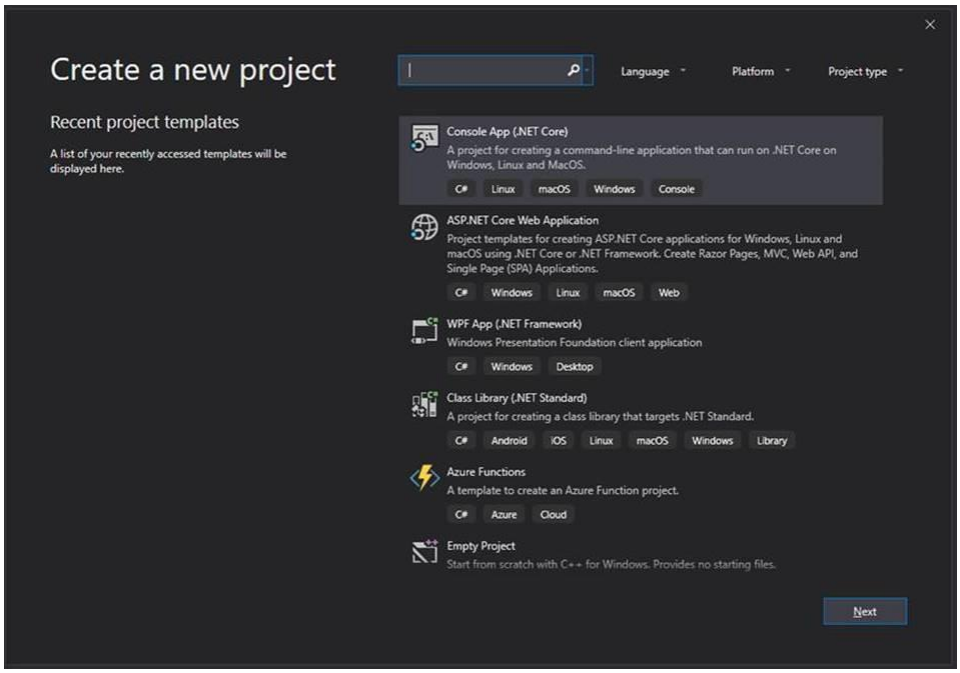


2. Next, review the standard build process for a Visual Studio project.

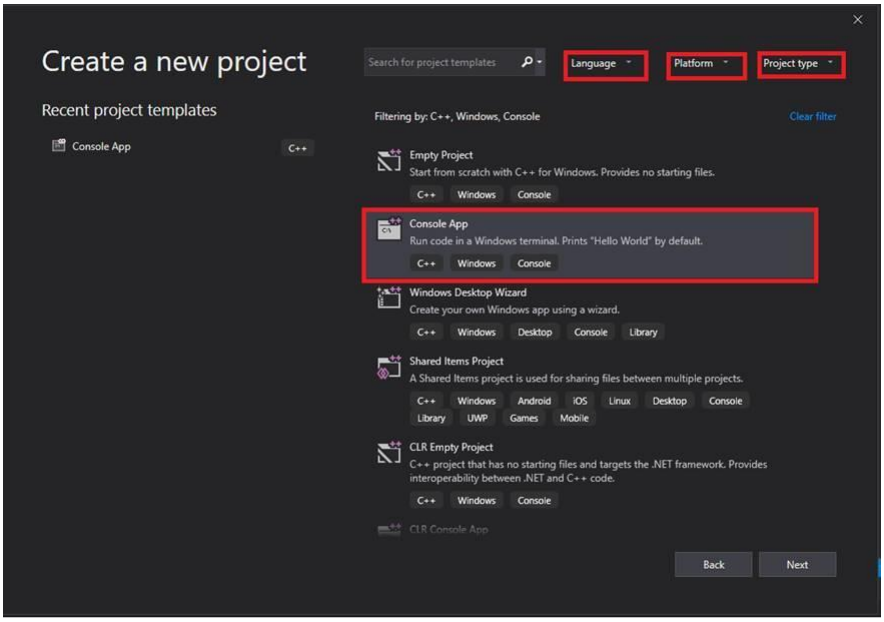
3. If C++ is not an already installed language in Visual Studio, you need to install **Desktop development with C++** from the Visual Studio Installer.



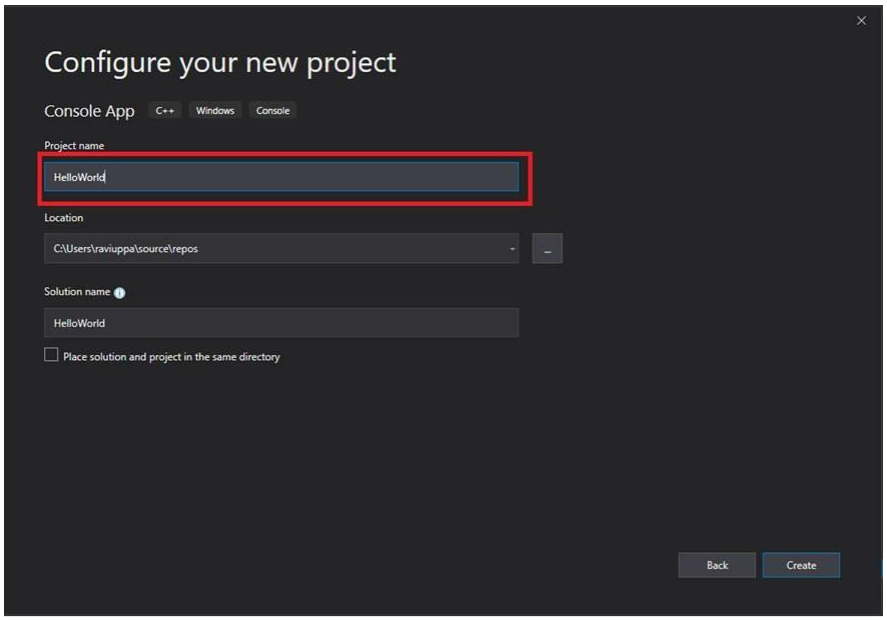
4. After installing the Desktop development with the C++ workload, open Visual Studio 2019, and click **Create a new project** to start creating your HelloWorld project.



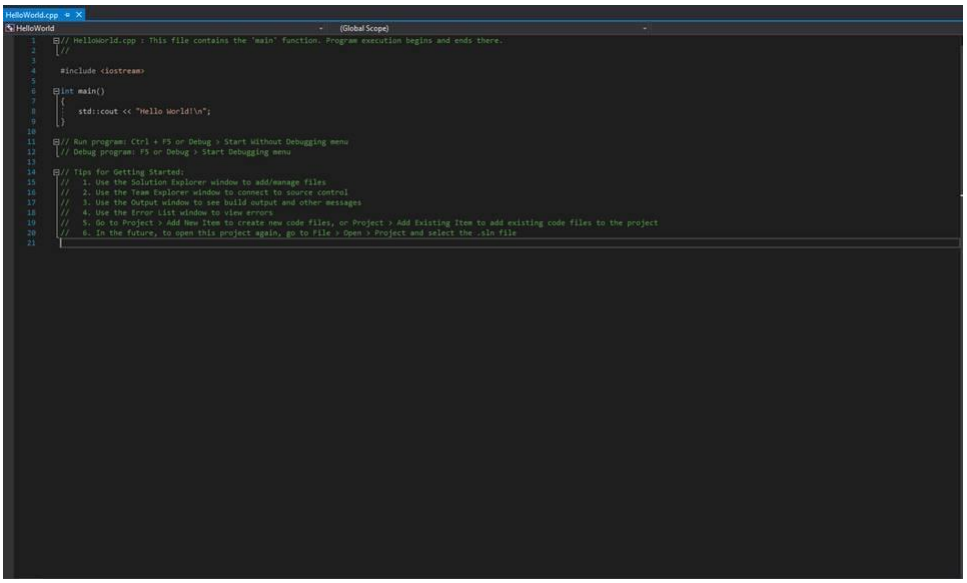
5. Select Language as **C++**, Platform as **Windows**, and Project Type as **Console**. Then select **Console App** and click **Next**.



6. Type “HelloWorld” as the project name and click **Create**.



7. You can see your first C++ project.



8. Replace the code with the following:

```
#include <iostream>

using namespace std;

   int main()
   {
    std::cout << "Hello World”! ;

    return 0;
}
```

```cpp
// Your First C++ Program

#include <iostream>

int main() {
    std::cout << "Hello World!";
    return 0;
}
```

Output

```
Hello World!
```

9. Compile and run your project by clicking the green triangle in the tool bar (**Local Windows Debugger**) or press F5 to **Run in Debug Mode**. Use Ctrl+F5 to run in **Release Mode**. Visual Studio allows single-click for build and debugging. Click **Yes** to build the project.
10. You can see “Hello World!” in the console window.

CS101L - Introduction to Computing and AI Lab

III. Lab 2 Variables. Arithmetic Operations and Control Structures -1

The aims of this lab are to cover Control Structures of C++.

Learning Objectives:

- Variable Types
- Declaring Variables
- Arithmetic Operators
- Comments
- input/output in C++
- Comparison/Relational Operators
- Logical Operators
- If Statement
- If - else statement
- else - if Statement
- Introduction to Switch and Case Statement
- General Syntax of Switch and Case Statements
- Use of Break Statement

Outcomes:

- Students should be able to know C++ variables, their types and arithmetic operators.
- Students should be able to use cin and cout methods in the programs.
- Students should be able to use all types of C++ operators
- Students should be able to use if, if-else and else-if statements
- Students should be able to use Switch statement, Case statement, Break statement

CS101L - Introduction to Computing and AI Lab

1.7 Variables, Types and Operators

A variable is a place to store a piece of information. Just as you might store a friend's phone number in your own memory, you can store this information in a computer's memory. Variables are your way of accessing your computer's memory.

C++ imposes fairly strict rules on how you can name your variables:

- variable names must begin with a letter
- variable names are "case-sensitive" (i.e., the variable "myNumber" is different from the variable "MYNUMBER" which is different from the variable "mYnUmBeR")
- variable names can't have spaces
- variable names can't have special characters (typographic symbols)

What can you name your variables? In general, variable names can be composed of **letters**, **numbers**, and **underscores** (_).

Keywords: C++ reserves certain names which have special meaning to the language, and you are not allowed to use any of these keywords as variables names. Some examples of C++ keywords are **int**, **for**, **else**, and **class**. You can, however, use keywords in the middle of a variable name, such as "**foreign**" or "**classical**".

1.4 Variable Types

The values of variables are stored somewhere in an unspecified location in the computer memory as zeros and ones. Our program does not need to know the exact location where a variable is stored; it can simply refer to it by its name. What the program needs to be aware of is the kind of data stored in the variable.
It's not the same to store a simple integer as it is to store a letter or a large floating-point number; even though they are all represented using zeros and ones, they are not interpreted in the same way, and in many cases, they don't occupy the same amount of memory.
Fundamental data types are basic types implemented directly by the language that represent the basic storage units supported natively by most systems. They can mainly be classified into:

- **Character types:** They can represent a single character, such as `'A'` or `'$'`. The most basic type is `char`, which is a one-byte character. Other types are also provided for wider characters.
- **Numerical integer types:** They can store a whole number value, such as `7` or `1024`.
  They exist in a variety of sizes, and can either be *signed* or *unsigned*, depending on whether they support negative values or not.
- **Floating-point types:** They can represent real values, such as `3.14` or `0.01`, with different levels of precision, depending on which of the three floating-point types is used.
- **Boolean type:** The boolean type, known in C++ as `bool`, can only represent one of two states, `true` or `false`.

Here is the complete description of all data types:

Group	Type names*	Notes on size / precision
Character types	**Char**	Exactly one byte in size. At least 8 bits.
	char16_t	Not smaller than `char`. At least 16 bits.
	char32_t	Not smaller than `char16_t`. At least 32 bits.
	wchar_t	Can represent the largest supported character set.
Integer types (signed)	**signed char**	Same size as `char`. At least 8 bits.
	signed **short** *int*	Not smaller than `char`. At least 16 bits.
	signed **int**	Not smaller than `short`. At least 16 bits.
	signed **long** *int*	Not smaller than `int`. At least 32 bits.
	signed **long long** *int*	Not smaller than `long`. At least 64 bits.
Integer types (unsigned)	**unsigned char**	(same size as their signed counterparts)
	unsigned short *int*	
	unsigned *int*	
	unsigned long *int*	
	Unsigned long long *int*	
Floating-point types	**Float**	
	Double	Precision not less than `float`
	long double	Precision not less than `double`

Boolean type	**Bool**	
Void type	**Void**	no storage
Null pointer	**decltype(nullptr)**	

2.2.3 Declaring Variables

Declaring a variable in C++ is simple. Let's say you want to declare a variable of type int called myAge.

That is to say, the variable myAge will store an integer. In C/C++, this is written: Int myAge;

All this does is tell the computer that you plan to use an integer, and that the integer's name is myAge. In some languages, variables are initialized to 0 - that is, a variable's initial value will be 0. This is not true of C++! Sometimes your variables will be initialized to 0, but sometimes they will be initialized with garbage. As you might anticipate, this can cause some nasty bugs. Hence, it is always a good idea to initialize your variables with some value. If you don't know what a variable's initial value should be, initialize it to 0.

CS101L - Introduction to Computing and AI Lab

Example 1:

Let's write a program that stores your age in a variable and outputs "My age is 21". The first line of the main function initializes myAge by assigning it a value immediately. compile and run the following code.

Code:

```
int main()
{
Int myAge   =     21;
cout<<"My age is "
<<myAge<<"years";
system("pause");
return 0;
}
```

1.5 Introduction to strings

Fundamental types represent the most basic types handled by the machines where the code may run. But one of the major strengths of the C++ language is its rich set of compound types, of which the fundamental types are mere building blocks.

An example of compound type is the `string` class. Variables of this type can store sequences of characters, such as words or sentences. A very useful feature!

The first difference with fundamental data types is that in order to declare and use objects (variables) of this type, the program needs to include the header where the type is defined within the standard library.

Code:

```
/   my     first
string  #include
<iostream>
#include
<string>   using
namespace   std;
int main ()
{
  string mystring;
  mystring = "This is a
  string"; cout << mystring;
  return 0;
}
```

As you can see in the previous example, strings can be initialized with any valid string literal, just like numerical type variables can be initialized to any valid numerical literal. As with fundamental types, all initialization formats are valid with strings:

```
string mystring = "This is a string";
string mystring ("This is a string");
string mystring {"This is a string"};
```

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and change its value during execution:

Example:

```
/ my first string
#include <iostream>
#include <string> using
namespace std;

int main ()
{
  string mystring;
  mystring = "This is the initial string content"; cout
  << mystring << endl;
  mystring = "This is a different string content"; cout
  < mystring << endl;
  return 0;
}
```

Note: inserting the endl manipulator ends the line (printing a newline character and flushing the stream).

The string class is a compound type. As you can see in the example above, compound types are used in the same way as fundamental types: the same syntax is used to declare variables and to initialize them.

1.8 C++ Comments

A comment is text that the compiler ignores but that is useful for programmers. Comments are normally used to annotate code for future reference. The compiler treats them as white space. You can use comments in testing to make certain lines of code inactive.

A C++ comment is written in one of the following ways:

- The /* (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the */ characters. This syntax is the same as ANSI C.
- The // (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. Therefore, it is commonly called a single-line comment.

The comment characters (/*, */, and //) have no special meaning within a character constant, string literal, or comment.

Example:

```
int main()

{

//declaring integer and character variables

int a; char ch;

/*Initializing   the

variables

*/

a=10; ch='b';

/*Printing       the

variables

*/

cout<<"The value of ch is " <<ch<<endl;

cout<<"The value of a is "<<a<<endl;

system("pause");

return 0;

}
```

Output:

```
The value of ch is b
The value of a is 10
```

Notice the use of "**endl**" at the end of the **cout** statements. It simply adds a carriage return which ends the current line.

1.7 Inputting Multiple Values

If you have multiple format specifiers within the string argument of cin, you can input multiple values. All you need to do is to separate each format specifier with a ">>"a string that separates variables. As a default, cin stops reading in a value when **space**, tab or **Enter** is pressed. Consider

```
cin>>roll_number>>MyAge;
```

(Assume that roll_number and MyAge have been declared beforehand!). If I entered: 1 2 and pressed Enter, 1 would get assigned to roll_number, and 2 would get assigned to MyAge. But if I entered 1, 2 and pressed Enter, roll_number would equal 1, but MyAge won't get assigned 2 because cin was not expecting a comma in the input string.

Now let us look at a way to get input from the user and store it in variables. Consider the following program:

Example:

```
int main()

{
inta,b;
cout<<"Enter  value  of  a:  \n";
cin>>a;
cout<<"Enter  value  of  b:  \n";
cin>>b;

cout<<  "The  value  of  a  is:  "<<  a  <<endl;

cout<<"The value of b is: "<< b <<endl;

cout<< "Enter new value for both separated by a space: \n";
cin>>a >> b;

cout<<  "New values are:  "<< a  << "  "<<  b  <<endl;

system("pause");

return 0;

}
```

Output:

```
Enter value of a:
4
Enter value of b:
7
The value of a is: 4
The value of b is: 7
Enter new value for both separated by a space:
2 5
New values are: 2 5
```

1.8 Arithmetic Operators

Arithmetic operators are commonly used in a variety of programming languages. In C, there are five of them, and they all take two **OPERANDS**. Recall that an operand is an expression that is required for an operator to work. For example, for 8 + 4, 8 and 4 are considered as the operands.

CS101L - Introduction to Computing and AI Lab

Operator Name	Symbol
Multiplication	*
Division	/
Modulus	%
Addition	+
Subtraction	-

- What's With the % ?

Multiplication, addition and subtraction are the simplest to use. Division is also easy, but watch out for the truncation of an int divided by an int! Now, the one that confuses novices is the modulus operator, sometimes known as the remainder operator.

To keep things simple, a%b returns the **REMAINDER** that occurs after performing a/b. For this operator, a and b MUST be integers!

For example, 6%3 returns 0 because 3goes into 6 EXACTLY. Similarly, 4%4, 8%2 and 16%8 all return
0. So why does 3%4 return 3? Picture it this way: you have 3 holes to fill, but you can only fill 4 holes at a time. You can't fill a group of 4 holes, therefore the 3 holes you had are still empty. Similar story for 7%4 because you can fill in one group of 4 but still have 3 holes remaining.

Example:

```
int main()

{

inta,b;

int sum;

cout<<"Enter  value  of  a:  \n";

cin>>a;

cout<<"Enter  value  of  b:  \n";

cin>>b;

sum=a+b;

cout<<"Sum:  "<<sum<<endl;

return 0;

}
```

Output:

```
Enter value of a:
3
Enter value of b:
6
Sum: 9
```

1.9 Practice Problem

Write a program that prompts the user to input a decimal number and outputs the number rounded to the nearest integer.

Solution:

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
  double userNumber;

  cout << "Enter a decimal number: ";
  cin >> userNumber;

  cout << setprecision(0) << userNumber << endl;

  return 0;
}
```

1.10 Practice Problem

Write a C++ program that prompts the user to input the elapsed time for an event in seconds. The program then outputs the elapsed time in hours, minutes, and seconds. (For example, if the elapsed time is 9630 seconds, then the output is 2:40:30.)

Solution:

```
#include <iostream>

using namespace std;

int main()
{
   int secondsElapsed, hours, minutes, seconds;

   const int secondsPerMinute = 60;
   const int secondsPerHour = 60 * secondsPerMinute;

   cout << "Please enter the number of seconds elapsed: ";
   cin >> secondsElapsed;

hours = secondsElapsed / secondsPerHour; secondsElapsed
= secondsElapsed % secondsPerHour; minutes =
secondsElapsed / secondsPerMinute; seconds =
secondsElapsed % secondsPerMinute;

    cout << hours << ":" << minutes << ":" << seconds << endl;
   return 0;
}
```

CS101L - Introduction to Computing and AI Lab

Comparison/Relational Operators

Operator name	Syntax
Less than or equal to	a <= b
Less than or equal to	a <= b
Greater than	a > b
Greater than or equal to	a >= b
Not equal to	a != b
Equal to	a == b

Logical Operators

Operator name	Syntax
Logical negation (NOT)	!a
Logical AND	a && b
Logical OR	a || b

2.1 Decision Control Structures

Till now we have used sequence control structure in the programs, in which the various steps are executed sequentially i.e. in the same order in which they appear in the program. In C programming the instructions are executed sequentially, by default. At times, we need a set of instructions to be executed Depending upon different conditions to be satisfied.

In such cases we must use decision control instructions. This can be achieved in C using;

1. The if statement.
2. The if-else statement
3. The else - if statement

2.1.1 The C/C++ "if" Statement

The **if** statement controls conditional branching. The body of an **if** statement is executed if the value of the expression/condition specified in the **if** statement is true. The syntax of the if statement is as follows:

Syntax:

if(expression)

{

Block of statement;

}

Example: 1

Write a program in which it takes a number from keyboard as an input and if the number is greater than 100 it prints "The number is greater than hundred".

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
int number ;
cout<< "Enter an integer\n"; cin>>
number; if ( number >100 )
cout<<"The number is greater than 100"; getch();
return 0;
}
if (number<100)
{
cout<< "The number is greater than 100\n";
cout<< "No doubt that the number is greater than 100";
}
```

Note we did not use curly brackets '{ }'for body of **if ()** because it did not include multiple statements (block of statements). If it includes multiple statements, then it would have been something like this.

2.1.2 The C/C++ if-else Statement

if-else statement is similar to **if** with the addition of else statement. If the condition is false in **if** then its body will be skipped and the else statement's body will be executed.

Syntax:

```
if (expression)
{
else
{
Block of statement;
}
Block of statement;
}
```

Example: **2**

Write a program in which it takes two numbers from keyboard as input and subtract larger number from smaller.

```
#include <iostream>
#include<conio.h>
using namespace std;


int main()
{
```

```
    int a,b ;


    cout << "Enter first
    number\n"; cin >> a; cout
    << "Enter second
    number\n"; cin >> b;


    if ( a >=b )
    //this condition can also be written as if(a>b||a==b)
        cout <<a <<"-" <<b <<"=" <<a-b;
    else
        cout <<b <<"-" <<a << "=" <<b-a;


    getch();
    return 0;
}
```

2.1.3 Nested if-else statement.

Nested if is basically if inside if.

Example: 3

Write a program which take a number from keyboard and checks the number whether that number is less than 100 ornot if that number is lessthan100 than check that is it less than 50 or not.

```
#include <iostream>
#include <conio.h>
using namespace std;
int main()
```

```
{
     int number ;
     cout << "Enter an integer\n"; cin
     >>number; if ( number <100 )
     {
         cout<< "Yes the number is less
         than 100"; if ( number <50)
         {
         cout<< " and number is also less than 50";
         }
         else
         {
              cout<< " but the number is not less than 50";
         }
     }
     else
     cout<< "No the number is not less than
     100"; getch();
     return 0;
}
```

The above program checks if the number is less than 100. This check is done in the first **if** statement. If the result is true the program enters the second **if** statement which is also called the **nested if** statement. Here another check is performed. The number is checked if it less than 50 or not. If the number is less than 50 it is conveyed to the user. The rest is self-explanatory.

2.1.4 The C/C++ else-if Statement

Sometimes we wish to make a multi-way decision based on several conditions. The most general way of doing this is by using the else if variant on the if statement. This works by cascading several comparisons. As soon as one of these gives a true result ,the following statement or block is executed, and no further comparisons are performed.

Syntax:

```
if (expression)
{
Block of statement;
}
else if (expression)
{
                Block of statement
}
else if (expression)
{
Block of statement;
}
. . // you can add as many else-if statements as many you need
. .
. .
else
{
        Block of statement
}
```

Example: 4

Write a program which takes marks as input and shows the output as follows:

Marks	Output
Greater than or equal to 75	Passed: Grade A
Greater than or equal to 60	Passed: Grade B
Greater than or equal to	45 Passed: Grade C
Lessthan45	Failed

```
#include <iostream>
#include <conio.h>
using namespace std;


int main()
{
     int marks;
     cout<< “Enter an marks\n”;
     cin >> marks ;
     if (marks >= 75)
        cout<< "Passed: Grade A\n";
     else if (marks >= 60)
       cout<< "Passed: Grade B\n";
     else if (marks >= 45)
        cout<< "Passed: Grade C\n";
     else
        cout<< "Failed\n";


     getch();
     return 0;
}
```

In this example, all comparisons test a single variable called marks. In other cases, each test may involve a different variable or some combination of tests. The same pattern can be used with more or fewer else if's, and the final lone else may be left out. It is up to the programmer to devise the correct structure for each programming problem.

Example: 5

Write a program which takes marks as input and then shows output as follows:

Marks	**Output**
87 – 100	Grade A
80 - 87	Grade B+
72 – 80	Grade B
67 – 72	Grade C+
60 - 67	Grade C
below 60	Failed

```
#include <iostream>
#include <conio.h>
using namespace std;

int main()
{
     int marks;

     cout << "Enter an marks\n";
     cin >> marks;



     if (marks >= 87 && marks <=100 )
         cout << "Grade A\n";
     else if (marks >= 80 && marks < 87)
         cout << "Grade B+\n";
     else if (marks >= 72 && marks< 80)
```

```
        cout<< "Grade B\n";
    else if (marks >= 67 && marks < 72)
        cout << "Grade C+\n";
    else if (marks >= 60 && marks< 67)
        cout << "Grade C\n";
    else
        cout << "Failed\n";


    getch();
    return 0;
}
```

In the above example logical operator && is used and because of this && operator condition will only be satisfied if both the conditions in a single **if** are true. If any of the condition is false because of && operator the whole condition will be treated as false.

2.2 Switch statement

Switch statement is C/C++ language is used for selection control. The difference between if/else and switch selection statements is that the second one is used for selecting from multiple statements.

There are times when you'll find yourself writing a huge if block that consists of many else if statements.

The switch statement can help simplify things a little. It allows you to test the value returned by a single expression and then execute the relevant bit of code.

You can have as many cases as you want, including a default case which is evaluated if all the cases fail.

Let's look at the general form...

switch (expression) {

case expression1:

/ one or more statements */*

case expression2:

/ one or more statements */*

/ ...more cases if necessary */*

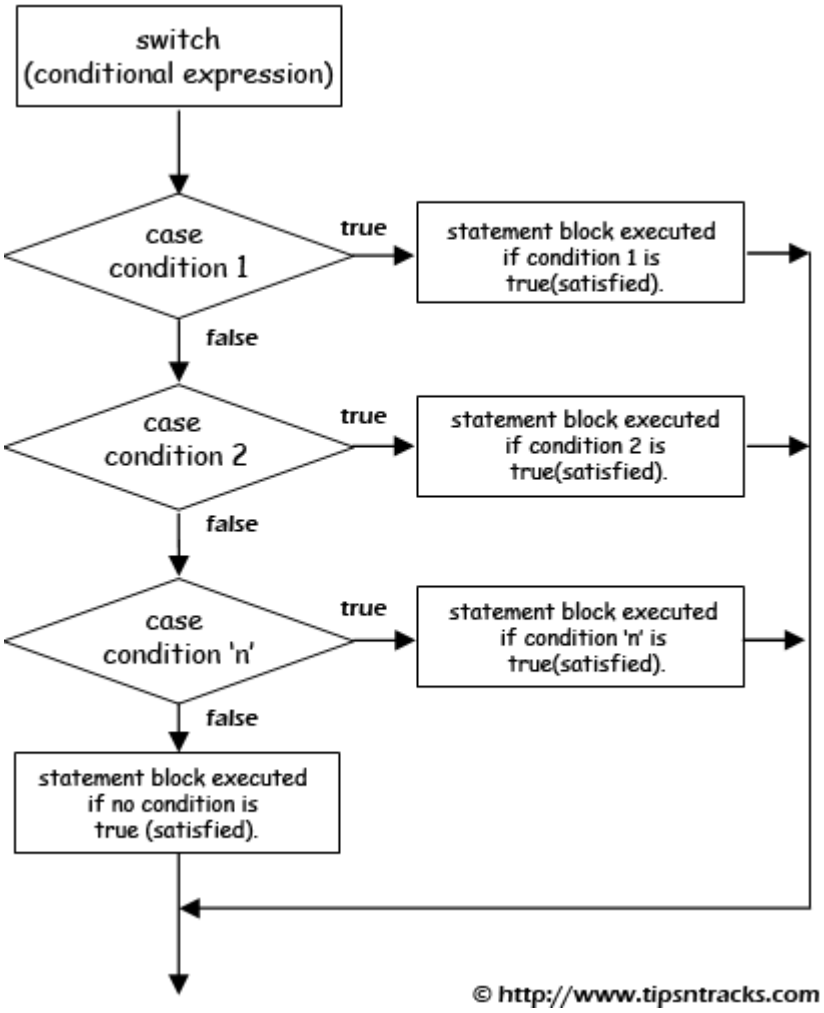
default:

/ do this if all other cases fail */*

}

expression is any legal C++ expression, and the statements are any legal C++ statements or block of statements. switch evaluates expression and compares the result to each of the case values. Note, however, that the evaluation is only for equality; relational operators may not be used here, nor can Boolean operations.

2.2.1 Flow Chart of Switch



Just look at the following example and examine the output:

Example 6

```
#include <iostream>
using namespace std;
int main()
{
     int a;
     cout<<"Pick a number from 1 to 4:\n";
     cin>>a;
     switch(a)
     {
          case 1:
          cout<<"You choose number 1\n";
          case 2:
          cout<<"You choose number 2\n";
          case 3:
          cout<<"You choose number 3\n";
          case 4:
          cout<<"You choose number 4\n";
          default:
          cout<<"That's not 1, 2, 3 or 4\n";
     }
     system("pause");
     return 0;
```

(Suppose I entered 2...)

Pick a number from 1 to 4:

2

You chose number 2

You chose number 3

You chose number 4

That's not 1,2,3 or 4!

You'll notice that the program will select the correct case but will also run through all the cases below it (including the default) until the switch block's closing bracket is reached.

To prevent this from happening, we'll need to insert another statement into our cases...

2.2.2 The break Statement

The *break* statement terminates the execution of the nearest enclosing do, for, switch, or while statement in which it appears. Control passes to the statement that follows the terminated statement.

Example: 7

```
#include <iostream>
using namespace std;
int main()
{
    int a;
    cout<<"Pick a number from 1 to
    4:\n"; cin>>a;
    switch(a)
    {
          case 1:
          cout<<"You choose number 1\n";
          break;
         case 2:
          cout<<"You choose number 2\n";
          break;

          case 3:
          cout<<"You choose number 3\n";
          break;
          case 4:
          cout<<"You choose number 4\n";
          break;
          default:
          cout<<"That's not 1, 2, 3 or 4\n";
    }
     system ("pause");
    return 0;
    }
```

On first inspection you'll find that it's virtually identical to the last example, except I've inserted a break statement at the end of each case to "break" out of the switch block.

Now it should work as expected:

```
Pick a number from 1 to 4:
2
You chose number 2
```

Example: 8

```
#include<iostream>
using namespace std;
int main()
{
     char grade;
     cout<<"Enter your grade: ";
     cin>>grade;
     switch(grade)
     {
         case'A':
         cout <<"Your average must be between 90 - 100"
         <<endl;
         break;
         case'B':
         cout <<"Your average must be between 80 - 89"
         <<endl;
         break;
         case'C':
         cout<<"Your average must be between 70 - 79"
```

```
        <  endl;
        break;
        case'D':
        cout<<"Your average must be between 60 -
        69" <<endl;
        break;
        default:
        cout<<"Your average must be below 60" << endl;
    }
    system("pause");
    return 0;
}
```

An example of break statement with loops:

Example 9

```
#include <iostream>
using namespace std;
int main()
{
    char c;
    int store;
    for(;;)
    {
          cout<<"\nPress any key, Q to
          quit:"; cin>>c;
          if(c == 'Q')
              break;
```

```
            else
            {
                store = (int) c; //convert character to corresponding ASCII
value
                cout<<store<<endl;
            }
        }
        system ("pause");
return 0;
}//loop exits only when 'Q' is pressed
```

NOTE: It is almost always a good idea to have a default case in switch statements. If you have no other need for the default, use it to test for the supposedly impossible case, and print out an error message; this can be a tremendous aid in debugging.

.

2.4 Practice Problem

McDonald offers chill out deals for students. They have special student discount of their different food items i-e: 25% discount on each chicken BBq platter, 10% on mayo garlic fries, 20% on each chicken burger and 15% on each beef burger. Original price of items are 400 ,700,300 and 600 rupees per item respectively. Moreover there 10% more discount on chicken Bbq if order contains more than 2 platters. Design a bill calculation system for McDonald.

Solution:

```
#include<iostream>

using namespace std;

 int main()
 {
        cout<<"***************Select Deals***************\n";
        cout<<"1. Chicken BBQ\n";
        cout<<"2. Mayo garlic fries\n";
        cout<<"3. Chicken burger\n";
        cout<<"4. Beef Burger\n";
        int op;
        cout<<"enter your favorite meal\n";
        cin>>op;
        int cbb=400;
        int mf=700;
        int cb=300;
        int bb=600;
        int amt;
        if(op==1)
        {
                cout<<"How many chicken BBq platter do you want??";
                cin>>amt;
                if (amt>2){
                cbb=cbb-(cbb*.25);
                amt=amt*cbb;
                amt=amt-(amt*.10);
                }
                else{
                        cbb=cbb-(cbb*.25);
                amt=amt*cbb;
                 }
                cout<<"YOur bill is: "<<amt<<endl;

         }
```

```
        else if(op==2)
        {
                                cout<<"How many fries platter do you want??";
                cin>>amt;
                mf=mf-(mf*.10);
                amt=amt*mf;
                cout<<"YOur bill is: "<<amt<<endl;
        }
        else if(op==3)
        {
           cout<<"How many chicken burger do you want??";
                cin>>amt;
                cb=cb-(cb*.20);
                amt=amt*cb;
                cout<<"YOur bill is: "<<amt<<endl;
        }
        else if(op==4)
        {
                                cout<<"How many Beef burger do you want??";
                cin>>amt;
                bb=bb-(bb*.15);
                amt=amt*bb;
                cout<<"YOur bill is: "<<amt<<endl;
        }
        else
        cout<<"Your entered option is not available please try again\n";
}
```

Lab#03

Control Structures II

The aims of this lab are to cover Control Structures of C++.

Learning Objectives:

- While Loop
- Nested While Loop
- Do While loop.

Outcomes:

- Students should be able to use while, nested while and do while loop loops.
- Students should be able to use For loop, Nested for Loop and differentiate them.

3.1 Loop

Loops are used to repeat a block of code. Being able to have your program repeatedly execute a block of code is one of the most basic but useful tasks in programming -- many programs or websites that produce extremely complex output are really only executing a single task many times. Now, think about what this means: a loop lets you write a very simple statement to produce a significantly greater result simply by repetition.

3.2 The while loop

The most basic loop in C is the while loop. A while statement is like a repeating if statement. Like an If statement, if the test condition is true: the statements get executed. The difference is that after the statements have been executed, the test condition is checked again. If it is still true the statements get executed again. This cycle repeats until the test condition evaluates to false.

A **while loop** is a control flow statement that allows code to be executed repeatedly based on a given condition. The *while* consists of a block of code and a condition. The condition is first evaluated and if the condition is true the code within the block is then executed. This repeats until the condition becomes false.

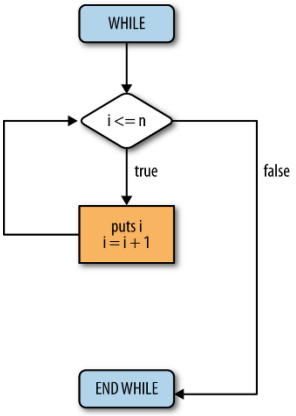
Syntax

Basic syntax of while loop is as follows:

```
while ( expression )
{
  Single
  statement or
  Block of statements;
```

The expression can be any combination of Boolean statements that are legal. Even, (while x ==5 ||v ==7) which says execute the code while x equals five or while v equals 7.

Flow Chart of While Loop



Example 3.1

Write a program using while loop who print the values from 10 to 1.

```
#include <iostream>
using namespace std;
int main()
{
     int i = 10;
     while ( i > 0 )
     {
          cout<<i; cout<<"\n";
          i = i -1;
     }
     system("pause");
```

```
    return 0;
}
```

a. This will produce following output:

10
9
8
7
6
5
4
3
2
1

Our next example is very simple. It usesawhilelooptocountfrom1 to 10. It does so by incrementing a variable named *counter* by 1 in each iteration or cycle. Incrementing a variable means to add something to it, so that its value is increased.

Example 3.2:
Print Numbers from 1 to 10 using a while loop

```
#include <iostream>
using namespace std;
int main()
{
    int counter = 1;
    while (counter <= 10)
    {
```

```
            cout<<counter;
            cout<<"\n";
            counter++;
        }
        system("pause");
        return 0;
    }
```

Our 3rd exampleisbasedonawhile-loopthatkeepsonrunninguntilacertainconditionisreached(a certain value is entered by the user). That certain value is called the 'sentinel value', 'signal value' or 'flag value'.

Our program asks the user for an integer value, if that value is not **-1**, it keep so running through the next cycle/iteration.

Example 3.3:

Printing the numbers you entered using a while loop

```
#include <iostream>
using namespace std;
int main()
{
    int flag;
    cout<<"Enter any number: ( -1 to quit)
    "; cin>>flag;
    cout<<"Entering the while loop now...\n";
    while (flag != -1)
    {
        cout<<"Enter any number: ( -1 to quit)
        "; cin>>flag;
        cout<<"You entered \n"<<flag;
    }
    cout<<"Out of loop now";
    system("pause");
```

```
    return 0;}
```

Nested While

Example 3.4:

Write a program who prints asterisk sign (*) in such a way using nested while loops

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

```
#include  <iostream>
using namespace std;
int main()
{
int i = 15, j;
while (i>0)
{
   j=j-1;
   while (j>0)
   {
      cout<<“*”;
```

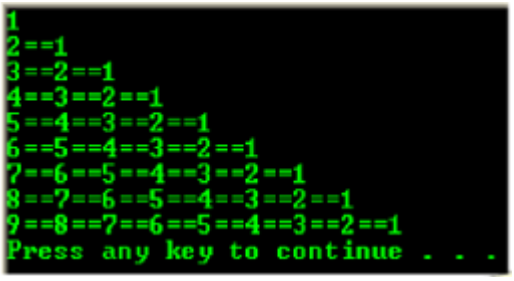
```
        j=j-1;
    }
cout<<"\n";
i=i-1;
}
system("pause");
return 0;
}
```

Example 3.5

A program to show the nested while loop

```
#include  <iostream>
using namespace std;
int main()
{
//variable for counter
int i = 1; j;
//outer loop, execute this first …
//for every i iteration, execute the inner loop
        while(i<=9)
        {
                cout << i;
        //then execute inner loop with loop index j,
        //the initial value of j is i-1
                j=j-1;
                while(j<0)
                {
                  cout<<"=="<<j;
//decrement j by 1 until j>0, i-e j=1
                        j=j-1;
                }
        }
        return 0;
}
```

Output:



3.3 The do-while loop

Its format is:

do statement while (condition);

Its functional it is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if condition is never fulfilled. For example, the following example program echoes any number you enter until you enter 0.

Example 3.6:

```
#include  <iostream>
using namespace std;
int main()
{
        long n;
        do
        {
        cout<<"Enter number (0 to end):";
        cin >> n;
        cout<< "You entered: "<< n <<"\n";
        } while
        (n!=0);
        return 0;
}
```

Output:

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
```

```
Enter number (0 to end): 0
You entered: 0
```

3.4 Practice Problem:

GIKI admin opens a joint account for their employees (Assistant professors, lecturers and Engineers) with starting amount of 50,000. Any employee in case of emergency in need of money can utilize the account and draw certain amount from the account i-e For Assistant professors its 25,000-maximum limit, for lecturers its 20,000 and for engineers its 10,000. Design a transaction system for the admin which applies all the checks (of maximum limit) on account and perform transactions. System should detect invalid inputs and display error messages.

Solution:

```
#include<iostream>
using namespace std;
int main()
{
	int money=50000;
	char op;
	int m;
	int t=1;
	cout<<"Welcome to giki joint account\n";
	cout<<"Enter options for following profession\n";
	cout<<"A. Assistant Professor\n";
	cout<<"B.Lecturer\n";
	cout<<"C. Engineer\n";
	while(t!=0)
	{
		cout<<"\nEnter your profession\n";
		cin>>op;
		if(op=='A')
		{
			cout<<"Enter money you want to transact\n";
			cin>>m;
			if(m<=25000)
			{
				money=money-m;
				cout<<"transaction successful\n";
				cout<<"Amount left in account\n";
				cout<<money;
```

```
                        }
                        else
                        if(op=='B')
                {
                        cout<<"Enter money you want to transact\n";
                        cin>>m;
                        if(m<=20000)
                        {
                                money=money-m;
                                cout<<"transaction successful\n";
                                cout<<"Amount left in account\n";
                                cout<<money;
                        }

                }
                else
                if(op=='C')
                {
                        cout<<"Enter money you want to transact\n";
                        cin>>m;
                        if(m<=10000)
                        {
                                money=money-m;
                                cout<<"transaction successful\n";
                                cout<<"Amount left in account\n";
                                cout<<money;
                        }


        }
        else{
                cout<<"Option entered is wrong\n";
        }
        cout<<"Want to continue\n";
        cin>>t;
}
}
}
```

3.5 Practice Problem:

Shanley hotel in Islamabad wants to replace their waiters with robots to take orders from their customers. Hotel owner wants to display menu on the screen of robots showing menu and their prices as:

3.5.1 Kabab 500 rupee with three pieces.

3.5.2 Rice with BBQ 200 per platter

3.5.3 Drinks 50 each

3.5.4 Mutton karahi 1200

Customer should be able to select items from menu as well as their amount (how much they want to order each item). Wrong inputs should be detected and error message should be displayed in case of wrong input. Bill should be displayed as well on the screen of the robot when customer is done with order. Write a program for above scenario.

Solution:

```
#include<iostream>
using namespace std;
int main()
{

        int t=1;
        int a1=500;
        int a2=200;
        int a3=50;
        int a4=1200;
        int amt=1;
        int op;
        int amount=0;


        do
        {
        cout<<"*******Shanley Hotel Menu******\n";
        cout<<"1. Kabab ------- 500 WITH 3 PIECES\n";
        cout<<"2.Rice with BBq ---- 200 PER PLATTER\n";
        cout<<"3.Drinks --- 50 each\n";
        cout<<"4. mutton Karahi ---- 1200\n";
        cout<<"Select your food items";
```

```
cin>>op;
if(op==1)
{
	cout<<"How many servings\n";
	cin>>amt;
	amount=amount+a1*amt;

}
else
	if(op==2)
{
	cout<<"How many servings\n";
	cin>>amt;
	amount=amount+a2*amt;

}
else
	if(op==3)
{
	cout<<"How many servings\n";
	cin>>amt;
	amount=amount+a3*amt;

}
else
	if(op==4)
{
	cout<<"How many servings\n";
	cin>>amt;
	amount=amount+a4*amt;

}
else{
	cout<<"Try again!! Wrong input\n";
}


	cout<<"Want to continue?? Press 1\n";
	cin>>t;
}while(t!=0);
```

```
        cout<<"Your total bill are:\n";
        cout<<amount;
}
```

Lab#04

Control Structures II

The aims of this lab are to cover Control Structures of C++.

Learning Objectives:

- Use of For Loop
- Use of Nested for Loop
- Comparison between For loop, while loop and Do While loop.

Outcomes:

- Students should be able to use while, nested while and do while loop loops.
- Students should be able to use For loop, Nested for Loop and differentiate them.

4.1 For Loop

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

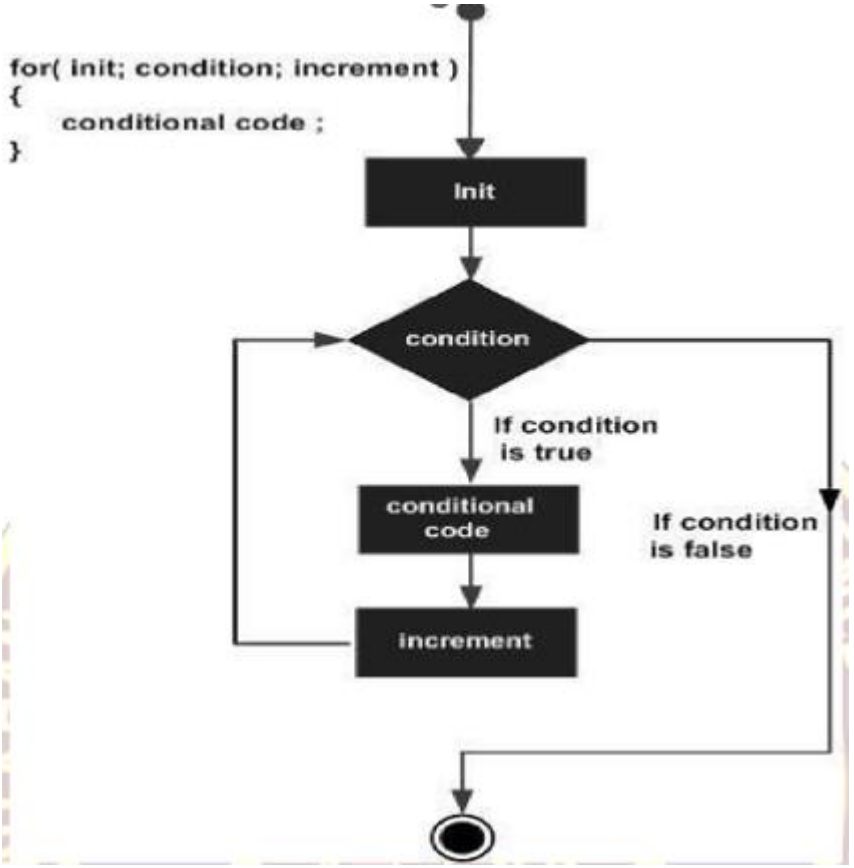
The syntax of a for loop in C++ is:

```
for ( init; condition; increment )
{
statement(s);
}
```

Here is the flow of control in a for loop:

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute, and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram:



Example: 4.1

```
  #include <iostream>
  using namespace std;
  int main ()
  {
      // for loop execution
      for( int a = 10; a < 20; a = a + 1 )
      {
cout<< "value of a: " << a <<endl;
```

```
    }
    return 0;
    system("pause");
}
```

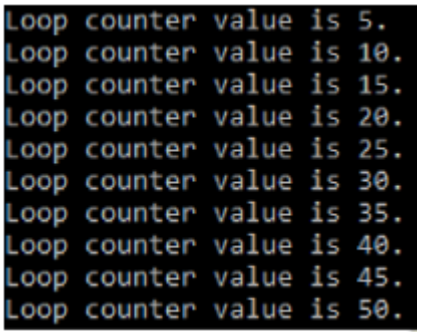
Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Example: 4.2

```
#include<iostream>
using namespace std;
int main()
{
    int x;
    for ( x=5; x <= 50; x = x+5 )
    {
        /   x = x + 5 could also be written x += 5
        cout<< "Loop counter value is " << x <<
        ".\n";
    }
    system ("pause");
    return 0; }
```

Output:



Can you modify the above program that counts from 50 to 5 with decrements of 5?

Nested For Loop

We have seen the advantages of using various methods of iteration, or looping. Now let's take a look at what happens when we combine looping procedures.

The placing of one loop inside the body of another loop is called **nesting**. When you "**nest**" two loops, the outer loop takes control of the number of complete repetitions of the inner loop. While all types of loops may be nested, the most nested loops are **for** loops.

Example: 4.3

Rectangle comprised of x's:

```
for (int rows = 0; rows < 4; rows++)
{
     for (int col = 0; col < 12; col++)
     {
           cout<<"x";
     }
}
```

Output of the program is displayed as follows:

xxxxxxxxxxxx

xxxxxxxxxxxx

xxxxxxxxxxxx

xxxxxxxxxxxx

Example: 4.4

Missing **init** statement in for loop

```
#include<iostream>
using namespace std;
int main()
{
     int x = 0;
     for ( ; x < 10; x++ )
     {
          cout<<"\n"<<x;
     }
     system ("pause");
     return 0;
}
```

Note: for (;;) works as an infinite loop.

Which Loop should I use?

- while: the loop must repeat until a certain "condition" is met. If the "condition" is FALSE at the beginning of the loop, the loop is never executed. The "condition" may be determined by the user at the keyboard. The "condition" may be a numeric or an alphanumeric entry. This is a good, solid looping process with applications to numerous situations.
- do-while: operates under the same concept as the while loop except that the do-while will always execute the body of the loop at least once. (Do-while is an exit-condition loop -- the condition is checked at the end of the loop.) This looping process is a good choice when you are asking a question, whose answer will determine if the loop is repeated.
- for: the loop is repeated a "specific" number of times, determined by the program or the user. The loop "counts" the number of times the body will be executed. This loop is a good choice when the number of repetitions is known or can be supplied by the user.

Example 4.5:

The following program fragments print the numbers 1 - 20. Compare the different looping procedures. Remember, there are always MANY possible ways to prepare code!

```
do-while:
int ctr = 1;
do
{
        cout << ctr++ << "\n";
} while
(ctr<=20); for:
int ctr;
for (ctr = 1; ctr <=20; ctr++)
{
        Cout<<ctr<<"\n";
}
while:
int ctr = 1;
while
(ctr<=20)
{
        Cout<<ctr++<<"\n";
}
```

Common Error

If you wish the body of a for loop to be executed, DO NOT put a semicolon after the for's parentheses. Doing so will cause the for statement to execute only the counting portion of the loop, creating the illusion of a timing loop. The body of the loop will never be executed.
Semicolons are required inside the parentheses of the for loop. The for loop is the only statement that requires such semicolon placement.

4.1 Practice Problem:

Write a program that prompts the user to input an integer and then outputs both the individual digits of the number and the sum of the digits. For example, it should output the individual digits of 3456 as 3 4 5 6, output the individual digits of 8030 as 8 0 3 0, output the individual digits of 2345526 as 2 3 4 5 5 2 6, output the individual digits of 4000 as 4 0 0 0, and output the individual digits of -2345 as 2 3 4 5.

Solution:

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
   int inputNumber, sum, individualNumber;
   cin >> inputNumber;

   inputNumber = abs(inputNumber); //Handle negative
   numbers sum = 0;

   do {
      individualNumber = inputNumber % 10; //Extract the last digit of the
      number sum += individualNumber;
      inputNumber = inputNumber / 10; //Remove the last
   digit } while (inputNumber > 0);

   cout << "The sum of the individual numbers is: " << sum << endl;


   return 0;
}
```

4.2 Problem 02

To make telephone numbers easier to remember, some companies use letters to show their telephone number. For example, using letters, the telephone number 438-5626 can be shown as GET LOAN. In some cases, to make a telephone number meaningful, companies might use more than seven letters. For example, 225-5466 can be displayed as CALL HOME, which uses eight letters. Write a program that prompts the user to enter a telephone number expressed in letters and outputs the corresponding telephone number in digits.

If the user enters more than seven letters, then process only the first seven letters. Also output the â€“ (hyphen) after the third digit. Allow the user to use both uppercase and lowercase letters as well as spaces between words.

Solution:

```
#include <iostream>

using namespace std;

int main()
{
   char letter;
   int counter = 0;

   cout << "Program to convert letters to their corresponding telephone digits" << endl;

   while (cin.get(letter) && counter < 7 ) {

      if (letter != ' ' && letter >= 'A' && letter <= 'z') {
         counter++; // Only increment the counter for valid characters
         if (letter > 'Z') {
            letter = (int)letter-32; // Convert lowercase to uppercase if required.
         }

         if (counter == 4) {
            cout << "-"; // Print the hyphen when required
         }

         switch (letter) {
            case 'A':
            case 'B':
            case 'C':
               cout << "2";
               break;
            case 'D':
            case 'E':
            case 'F':
               cout << "3";
               break;
            case 'G':
            case 'H':
            case 'I':
               cout << "4";
               break;
            case 'J':
```

```
            case 'K':
            case 'L':
               cout << "5";
               break;
            case 'M':
            case 'N':
            case 'O':
               cout << "6";
               break;
            case 'P':
            case 'Q':
            case 'R':
            case 'S':
               cout << "7";
               break;
            case 'T':
            case 'U':
            case 'V':
               cout << "8";
               break;
            case 'W':
            case 'X':
            case 'Y':
            case 'Z':
               cout << "9";
               break;
             default:
               break;
         }
      }

   }
   return 0;
   }
```

4.3 Problem 03

Write a C++ program that does the following.

4.3.1 It asks the user to enter an odd negative integer.

4.3.2 The program reads a value *n* entered by the user. If the value is not legal, the program repeatedly makes the user type in another value until a legal value of *n* has been entered.

4.3.3 The program computes the equivalent positive number and finds the entries of Fibonacci series up to that number.

Sample Output:

Solution:

```
#include  <iostream>
using namespace std;
int main()
{
        int c, r, n;
        int fibonacci, first = 0, second = 0;
        cout << "Enter a negative odd integer: ";
        cin >> n;
        while ((n >= 0) || (n % 2 == 0))
        {
                cout << "Illegal Entry. Try again: ";
                cin >> n;
        }

        n = -1*n;

  cout << "Fibonicci Series upto " << n << " Terms "<<
  endl; for ( int j = 0 ; j < n ; j++ )
```

```
        {


    if ( j <= 1 )
        fibonacci = j;
    else
    {
        fibonacci = first + second;


    }
    first = second;
    second = fibonacci;


    cout << fibonacci <<", ";
  }


        return 0;
}
```

Lab # 05

Exception Handling in C++

Learning Objectives:

- Functions concept
- Function prototype, definitions and functions calls
- Returning values from functions
- Function parameters/arguments

Outcomes:

- Students should be able to understand the concepts of functions.
- Students should practice the examples discussed in the tutorial to get the clear image of functions before performing the tasks.

5.1 Functions as modules:

A function is a block of code with a name. To motivate the idea of functions, imagine trying to program a robot to launch someone from Timbuktu to Boston via catapult. Dumping all the code into main would be absurdly long and difficult to keep track of, and nobody who read a single line would have a clue where that line fit in. We'd lose track of our programming goals.

It'd be much more intuitive to break it up into:

- Build the catapult
- Set up the catapult
- Fire the catapult

Before we go any further, this is a good way to code – write "pseudocode" first, using plain English to describe what's supposed to happen, then keep expanding each sentence until it's sufficiently detailed that you can express it as if-statements, loops, etc. Often some of the initial English descriptions will describe good ways to divide up the code into functions.

To turn the catapult pseudocode into code:

Example:

```
int main() {
buildCatapult();
setUpCatapult();
fireCatapult();
}
```

This style is often a good design for main – main a few calls to some functions that do all the real work. Each of the functions build Catapult, Setup Catapult, and fire Catapult is said to be “invoked” or “called” via a “function call” from “calling function” or “caller” (in this case, main). To call a function, type the name of the function, followed by parentheses. Now that we’ve packaged up each part of our big procedure and given it a name, we have a bunch of smaller units, each of which we can independently

- Write
- Debug
- Test
- Reuse in later programs

We can also now split development of the program among multiple people, with each developing a different set of functions. This sort of independence of program components is called modularity, and is critical to good software engineering. Splitting code up into functions increases the modularity of a program.

Functions mainly consist of three parts:

- Function prototype
- Function Definition
- Function call

5.2 Function Prototypes:

A function prototype is a declaration of the function that tells the program about the type of the value returned by the function and the number and type of arguments.

Function prototyping is one very useful feature of C++ function. A function prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values.

The prototype declaration looks just like a function definition except that it has no body i.e., its code is missing. This is the time you knew the difference between a declaration and a definition.

A declaration introduces a (function) name to the program whereas a definition is a declaration that also tells the program what the function is doing and how it is doing.

void printHellow();

bool isMultiple(int a,int b);

Therefore, you can say that a function prototype has the following parts :

- return type
- name of the function
- argument list

5.3 Function Definitions:

Example:

```
void   printHello()   {
cout << "Hello world!";
}
```

This definition specifies that we want to name the sequence of commands within the curly braces ({...}) printHello, so that we can then call it from another function, such as main, with the syntax printHello();.

Example:

```
bool isMultiple(int a, int b) {
 if( a % b == 0 )
 return true;
 else
return false;
}
```

Here, bool is the return type, is Multiple is the function name, a and b are argument names, and return true; and return false; are return statements. Each concept is discussed in detail below.

5.4 Function call:

To execute the codes of function body, the user-defined function needs to invoked(called).

In the above program, **isMultiple (2,3); printHellow()** inside main() function calls the user-defined function.

The function returns an integer which is stored in variable add.

Example:

```
int main()
{
printHellow();
isMultiple(2,3);
}
```

Example:

```
#include<iostream>
using namespace std;
void   interchange(int,int);
int main()
{
int     x=50,     y=70;
interchange(x,y);
cout<<"x="<<x<<"         and         "<<"y="<<y<<endl;
system("pause");
return 0;
}
void interchange(int x1,int y1)
{
int z1; z1=x1; x1=y1; y1=z1;
cout<<"x1="<<x1<<" and
"<<"y1="<<y1<<endl;}
```

5.5 Returning values:

The printHello function was just a shorthand way of telling the computer to run a sequence of commands – it issued instructions. A function like isMultiple can be thought of as asking the computer a question, which will require some commands to be run to figure out, but which ultimately gives back an answer.

In this example, the question we are asking is "Is a a multiple of b?" The name of the function must be preceded by a keyword specifying the data type of the answer we are expecting – the return type. In this case, since we are asking a yes or no question, we want our answer – our return value – to be a bool. At various points in the program, we specify return statements (here, return true; and return false;), which indicate what the answer should be if the conditions leading to that statement are fulfilled. The void return type specifies that there is no return value, which generally means that this function is for issuing instructions, not asking a question. Returning a value from a void function is a syntax error. Not returning a value from a non-void function is not a syntax error, but will usually cause runtime errors.

5.6 Parameters/Arguments:

In the isMultiple function definition, a and b are arguments – variables that are declared between the parentheses of the function definition and can then be used throughout the function body. The function call isMultiple(25, 5) specifies that when the body of the isMultiple function is executed for this call, the variables a and b in the function should store the numbers 25 and 5, respectively.

Functions can have arbitrarily large numbers of arguments – arbitrarily long argumentlists.

CS101L - Introduction to Computing and AI Lab

Lab# 06

User Defined Functions- II

Learning Objectives:

- Call by value and call by reference
- Constant and default parameters
- Scope of variables within function
- Parameters promotion or demotion
- Built in functions.

Outcomes:

- Students should be able to understand the concepts of call by value and call by reference .
- Students should be able to understand the concept constant and default parameter.
- Students should practice the examples discussed in the tutorial to get the clear image of functions before performing the tasks.

Any expression can be passed as an argument into a function – for instance, the following is a legal function call for a maximum function defined to take 3 arguments: maximum(a, b, a + b, maximum(c + d, e, f)).

C++ functions in can be compared to mathematical functions: just like f(x, y, z) means some mathematical expression named f evaluated for certain values of x, y, and z, isMultiple(a, b) means some set of instructions named isMultiple evaluated for certain values of a and b.

6.1 By-Value vs. By-Reference:

Arguments in a function can be passed by value or by reference. Variables that exist in the scope of the caller. For certain variable types (which we'll get to later), copying the value may be very computation-intensive, and may take up too much memory. To give the callee access to the caller's data, we pass the arguments by reference – the variable in the function actually refers to the same value in memory as in the caller.

The syntax for declaring arguments to be passed by value is simply an ampersand after the data type of the variable name.

Example:

```
void squareByReference(int &value)
{ value = value * value; }
```

CS101L - Introduction to Computing and AI Lab

If we have a variable number and pass it into squareByReference, the value of number will be updated when squareByReference exits. Since it updates the original value in memory, the function does not need to return a value.
Allowing the callee to modify the caller's data is very useful, but sometimes very dangerous.

Example:

```
#include<iostrea
m>          using
namespace std;
void      interchange(int&,
int&); int main()
{
int x=50, y=70;
cout<<"before calling the function the values
are"<<"x="<<x<<" and y="<<y<<endl; interchange(x,y);

cout<<"after calling the function the values are"<<"x="<<x<<"
and y="<<y<<endl;
system("pause");
return 0;
}
void interchange(int &x1,int &y1)
{
int z1; z1=x1; x1=y1; y1=z1;
}
 void squareByReference(int
 &value) { value = value * value;
 }
```

Practice problems:

Problem 1:

Classify Numbers

In this example, we use functions to write the program that determines the number of odds and evens from a given list of integers.

1. Initialize the variables, **zeros**, **odds**, and **evens** to **0**.
2. Read a number.
3. If the number is even, increment the even count, and if the number is also zero, increment the zero count; otherwise, increment the odd count.
4. Repeat Steps 2 and 3 for each number in the list.

The main parts of the program are: initialize the variables, read and classify the numbers, and then output the results. To simplify the function **main** and further illustrate

- A function initialize to initialize the variables, such as zeros, odds, and evens.
- A function getNumber to get the number.
- A function classifyNumber to determine whether the number is odd or even (and whether it is also zero). This function also incre- ments the appropriate count.
- A function printResults to print the results.

Main algorithm

We now give the main algorithm and show how the function main calls these functions.

1. Call the function initialize to initialize the variables.
2. Prompt the user to enter 20 numbers.
3. For each number in the list:
   a. Call the function getNumber to read a number.
   b. Output the number.
   c. Call the function classifyNumber to classify the number and increment the appropriate count.
4. Call the function printResults to print the final results.

Solution:

```
//
/  Program: Classify Numbers
/  This program reads 20 numbers and outputs the number of
/  zeros, odd, and even numbers.
//*****************************************************************
* #include <iostream>
#include
<iomanip> using
namespace std;
const int N = 20;
//Function
prototypes
void initialize(int& zeroCount, int& oddCount, int&
evenCount); void getNumber(int& num);
void classifyNumber(int num, int& zeroCount, int&
oddCount, int& evenCount);
void printResults(int zeroCount, int oddCount, int
evenCount); int main()
{
//Variable declaration
int counter; //loop control variable
int number;       //variable to store the new number
int zeros;        //variable to store the number of zeros
int odds;         //variable to store the number of odd integers int evens; //variable to store the
number of even integers
initialize(zeros, odds, evens);    //Step 1
cout << "Please enter " << N << " integers."
<< endl;          //Step 2
cout << "The numbers you entered are: "
```

```
<< endl;

for (counter = 1; counter <= N; counter++) //Step 3 {

getNumber(number); //Step 3a cout << number << " ";
//Step 3b classifyNumber(number, zeros, odds, evens);
//Step 3c
} // end for
loop cout <<
endl;



printResults(zeros, odds, evens); }



//Step 4 return 0;



void initialize(int& zeroCount, int& oddCount, int& evenCount)
{
zeroCount = 0;
oddCount = 0;
evenCount = 0;
}

void getNumber(int& num)
{
cin >> num;
}

void classifyNumber(int num, int& zeroCount, int& oddCount,
int& evenCount)
{
switch (num % 2)
{
case 0:
evenCount++;
if (num == 0)
zeroCount++;
break; case 1:
case -1:
oddCount++;
} //end switch
} //end classifyNumber
void printResults(int zeroCount, int oddCount, int evenCount)
{
```

```
cout << "There are " << evenCount << " evens, "
<   "which includes " << zeroCount << " zeros"
<   endl;

cout << "The number of odd numbers is: " << oddCount
<< endl;
} //end printResults
```

Sample Run:

Please enter 20 integers. The numbers you entered are:

0 0 12 23 45 7 -2 -8 -3 -9 4 0 1 0 -7 23 -24 0 0 12

0 0 12 23 45 7 -2 -8 -3 -9 4 0 1 0 -7 23 -24 0 0 12

There are 12 evens, which includes 6 zeros The number of odd numbers is: 8

CS101L - Introduction to Computing and AI Lab

Lab 07
User Defined Functions III

Learning Objectives:

- Functions concept
- Function prototype, definitions and functions calls
- Returning values from functions
- Function parameters/arguments
- Call by value and call by reference
- Constant and default parameters
- Scope of variables within function
- Parameters promotion or demotion
- Built in functions.

Outcomes:

- Students should be able to understand the concepts of functions.
- Students should practice the examples discussed in the tutorial to get the clear image of functions before performing the tasks.

Another way to pass the arguments in a function is to define it with const keyword.

7.1 const Arguments:

The "principle of least privilege" dictates that no module of a program should have the right to change more data than it needs to. When a function is passed an argument by reference, very often the caller doesn't want it changing the value of the argument. Even if an argument is passed by value, it still may be that the function doesn't need to change the argument's value anywhere, and trying to do so would likely be a logic error.

To prevent unwanted changes to arguments, the arguments can be declared const. For the duration of the function, an argument declared const cannot be changed.

Example:

```
int square(const int number) {
    return x * x;
}
```

Trying to change the value of number in the body of square would be a syntax error.

7.2 Default Arguments:

Default arguments allow you to specify default values for the last few arguments of a function, so that you need not pass any value for these arguments unless you want the arguments to be different from the default.

Example:

```
voidprint(const char *str = "\n") {
     std::cout << str;
}
```

This specifies to the compiler that the function call print() should be treated as equivalent to print("\n") – that is, the default thing to print is a newline, and we only need to explicitly pass a string if we want to print something else. If you call a function relying on default values, you cannot pass any values for arguments after the first argument for which you want to use the default. (Of course, this means that all the arguments after that must have default values.)

7.3 Scope:

Variables exist within scopes – blocks of code within which identifiers are valid. An identifier can be referenced anywhere within its scope, as long as the reference comes after its declaration.

Every set of braces is its own scope, and can contain local (i.e. non-global) variables. The moment the set of braces in which a variable was declared ends, the variable goes out of scope, i.e. it can no longer be referenced as an identifier. The program usually erases variables that have gone out of scope from memory. The scope of arguments to a function is the entire function body.

Example: A variable declared in the first line of a function can be referred to anywhere in the function, but nowhere outside of it. The moment the function exits, the variable ceases to exist in memory.

No two variables may share the same name within a scope. The exception is when you have opened another scope with a curly brace ({), at which point you may declare new variables of the same name. If another variable is declared within a smaller scope, the variable name refers to the variable in the narrowest enclosing scope.

Example: In the following code, there are two variables named i. When i is referenced from within the loop, it is interpreted as referring to the one declared in the loop. Every time the loop executes again, i is created anew.

Example:

```
void sampleFunc(int i) {
    for( int j = 0; j < 10; j++ ){ int
 i = 0;
 cout << i;
 }
}
```

7.4 Argument Promotion/Demotion:

Normally, the arguments must be of the types specified in the function definition. If the compiler has a predefined way of converting the arguments you pass into the types specified by the function definition, it will do this automatically. This can result in promotion or demotion of the argument value
– conversion into a higher or lower-precision data type, respectively.

Example:

A function defined as void myFunc(int a) {...} can be called as myFunc(24.3), since the compiler can convert 24.3 to an int by truncating the decimal.

Early exit:

No more code from a function is executed after one of that function's return statements; a return statement ends the function's execution.

Usually you want the logical path to return statements to be straightforward, but sometimes it's useful to return early from a function. You often want to do this in response to an error or an invalid argument.

Example:

```
bool   isPrime(int   number)   {
     if(number <= 1)
          return false;
     ...
}
```

No number less than 2 can be tested for primality, so if the argument the function received violated this constraint, it simply exits without even bothering to run the tests in the rest of the function.

7.5 Standard Library Functions:

Above discussed are the user defined functions. Other than user defined functions, there are lots of prepackaged functions to use in standard C++ libraries. To use them, you must include the appropriate header files.

These functions are very efficient and well-implemented. They should be used in preference to hand-crafted functions whenever possible.

Example: The sqrt function (in header file cmath) takes square roots. The rand function (in cstdlib) generates random integers.

Example:

```
#include<iostream>
using namespace std;
int main()
{
int number;
for(int i=1; i<=10; i++)
{
number=rand()%100;
cout<<number<<endl;
}
system("pause");
return 0;
}
```

7.6 Inline Functions

C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Following is an example, which makes use of inline function to return max of two numbers −

```
#include <iostream>

using namespace std;
```

```
inline int Max(int x, int y) {
   return (x > y)? x : y;
}

/  Main function for the
program int main() {
   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;

   return 0;
}
```

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

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

7.7 Function overloading in C++

Function overloading is a C++ programming feature that allows us to have more than one function having same name but different parameter list, when I say parameter list, it means the data type and sequence of the parameters, for example the parameters list of a function myfuncn(int a, float b) is (int, float) which is different from the function myfuncn(float a, int b) parameter list (float, int).

Function overloading is a compile-time polymorphism.
Now that we know what parameter list is let's see the rules of overloading: we can have following functions in the same scope.

sum(int num1, int num2)

sum(int num1, int num2, int num3)

sum(int num1, double num2)

The easiest way to remember this rule is that the parameters should qualify any one or more of the following conditions, they should have
different **type**, **number** or **sequence** of parameters.

For example:
These two functions have different parameter **type**:

sum(int num1, int num2)

sum(double num1, double num2)

These two have different **number** of parameters:

sum(int num1, int num2)

sum(int num1, int num2, int num3)

These two have different **sequence** of parameters:

sum(int num1, double num2)

sum(double num1, int num2)

All of the above three cases are valid case of overloading. We can have any number of functions, just remember that the parameter list should be different. For example:

int sum(int, int)

double sum(int, int)

This is not allowed as the parameter list is same. Even though they have different return types, its not valid.

Example 1: Overloading Using Different Types of Parameter

```
/  Program to compute absolute value
/  Works for both int and float

#include <iostream>
using namespace std;

/  function with float type
parameter float absolute(float
var){
      if (var <
       0.0) var
         = -var;
   return var;
}
```

```
/  function with int type
parameter int absolute(int var)
{
    if (var <
        0) var
        = -var;
   return var;
}

int main() {

   // call function with int type parameter
   cout << "Absolute value of -5 = " << absolute(-5) << endl;

   // call function with float type parameter
   cout << "Absolute value of 5.5 = " << absolute(5.5f) <<
   endl; return 0;
}
```

Output:

Absolute value of -5 = 5

Absolute value of 5.5 = 5.5

Example 2: Overloading Using Different Number of Parameters

```
#include <iostream>
using namespace std;

// function with 2 parameters
void display(int var1, double var2) {
   cout << "Integer number: " << var1;
   cout << " and double number: " << var2 << endl;
}

/  function with double type single
parameter void display(double var) {
   cout << "Double number: " << var << endl;
```

```
}

/  function with int type single
parameter void display(int var) {
    cout << "Integer number: " << var << endl;
}

int main() {

    int a = 5;
    double b = 5.5;

    /  call function with int type
    parameter display(a);

    /  call function with double type
    parameter display(b);

    /  call function with 2
    parameters display(a, b);

    return 0;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and double number: 5.5
```

Practice problems:

Problem 1:

During the tax season, every Friday, J&J accounting firm provides assistance to people who prepare their own tax returns. Their charges are as follows.

7.7.1 If a person has low income (<= 25,000) and the consulting time is less than or equal to 30 minutes, there are no charges; otherwise, the service charges are 40% of the regular hourly rate for the time over 30 minutes.

7.7.2 For others, if the consulting time is less than or equal to 20 minutes, there are no service charges; otherwise, service charges are 70% of the regular hourly rate for the time over 20 minutes.

For example, suppose that a person has low income and spent 1 hour and 15 minutes, and the hourly rate is $70.00. Then the billing amount is 70.00 * 0.40 * (45 / 60) = $21.00. Write a program that prompts the user to enter the hourly rate, the total consulting time, and whether the person has low income. The program should output the billing amount. Your program must contain a function that

takes as input the hourly rate, the total consulting time, and a value indicating whether the person has low income. The function should return the billing amount. Your program may prompt the user to enter the consulting time in minutes.

Solution:

```
#include <iostream>
#include <iomanip>

using namespace std;

double calculateBill(int income, int consultingMinutes, double hourlyRate);

int main()
{
   int income, consultingMinutes;
   double hourlyRate;
```

```
    cout << "Please enter the clients income: $" ;
    cin >> income;
    cout << "Please enter the consulting time in minutes:
    "; cin >> consultingMinutes;
    cout << "Please enter the hourly rate: $";
    cin >> hourlyRate;
    cout << fixed << showpoint << setprecision(2);
    cout << "Your total bill ammount comes to: $" << calculateBill(income,
consultingMinutes, hourlyRate) << endl;
    return 0;
}


double calculateBill(int income, int consultingMinutes, double
    hourlyRate){ if (income <= 25000) {
        if (consultingMinutes <= 30)
            return 0;
        else
            return hourlyRate * 0.40 * ((consultingMinutes - 30) / 60);
    }
    else {
        if (consultingMinutes <= 20)
            return 0;
        else
            return hourlyRate * 0.70 * ((consultingMinutes - 20) / 60);


        }
}
```

Problem 2

a) Write the definition of the function initialize that initializes x and y to 0 and z to the blank character.

b) Write the definition of the function getHoursRate that prompts the user to input the hours worked and rate per hour to initialize the variables hours and rate of the function main.

c) Write the definition of the value-returning function payCheck that calculates and returns the amount to be paid to an employee based on the hours worked and rate per hour. The hours worked and rate per hour are stored in the variables hours and rate, respectively, of the function main. The formula for calculating the amount to be paid is as follows: For the first 40 hours, the rate is the given rate; for hours over 40, the rate is 1.5 times the given rate.

d) Write the definition of the function printCheck that prints the hours worked, rate per hour, and the salary.

e) Write the definition of the function funcOne that prompts the user to input a number. The function then changes the value of x by assigning the value of the expression 2 times the (old) value of x plus the value of y minus the value entered by the user.

f) Write the definition of the function nextChar that sets the value of z to the next character stored in z.

g) Write the definition of a function main that tests each of these functions.

```
#include <iostream>
#include <iomanip>

using namespace std;
```

```
void initialise(int& x, int& y, char& z);
void getHoursRate(double& rate, double& hours);
double payCheck(double rate, double hours);
void printCheck(double rate, double hours, double amount);
void funcOne(int& x, int y);
void nextChar(char& z);

int main()
{
   int x, y;
   char z;
   double rate, hours;
   double amount;

   initialise(x, y, z);
   getHoursRate(rate, hours);
   amount = payCheck(rate, hours);
   printCheck(rate, hours, amount);
   cout << "The value of x is currently: " << x << endl;
   funcOne(x, y);
   cout << "The value of x is now: " << x << endl;
   nextChar(z);
   cout << "The value of z is now: " << z << endl;

   return 0;
}

void initialise(int& x, int& y, char& z) {
```

```
    x = 0;
    y = 0;
    z = ' ';
}

void getHoursRate(double& rate, double& hours) {
    cout << "Please enter the hourly rate: ";
    cin >> rate;
    cout << "Please enter the hours worked: ";
    cin >> hours;
}

double payCheck(double rate, double hours) {
    double amount = 0;

    if (hours > 40) {
        amount += ((hours - 40) * rate) * 1.5;
        amount += 40 * rate;
        return amount;
    } else {
        amount = hours * rate;
        return amount;
    }
}

void printCheck(double rate, double hours, double amount) {
    cout << "For working " << hours << " hours at $" << rate << "/hr you get a total of
    $"; cout << fixed << showpoint << setprecision(2);
```

```
    cout << amount << endl;
}


void funcOne(int& x, int y) {
    int tempNum;
    cout << "Please enter a number: ";
    cin >> tempNum;

    x = (x * 2) + y - tempNum;
}


void nextChar(char& z) {
    z++;
}
```

Lab# 08

Arrays

Learning Objectives:

- Array definition
- When to use Array
- Array declaration
- Array Initialization
- Accessing Array elements
- Copying Arrays

Outcomes:

- Students should be able to understand and use arrays.

8.1 What is Array?

Array is a data structure in C++, which stores a fixed size sequential collection of elements of the same type. An array is used to store a collection of data. It is often 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.

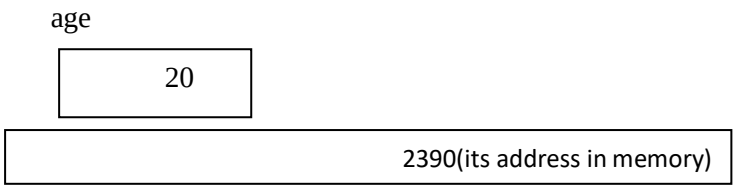
8.2 When to use array?

Let's start by looking at listing 1 where a single variable is used to store a person's age. **Example 1**

```
#include <iostream>
using namespace std;
int main()
```

```
{
    int age;
    age=23;
    cout<< age;
    return 0;
}
```

It is quite simple. The variable age is created at line (5) as int. A value is assigned to it. Finally, age is printed to the screen.



Now let's keep track of 4 ages instead of just one. We could create 4 separate variables but creating 4 separate variables is not a good approach. (If you are using 4 separate variables for it, then consider keeping track of 1000 ages instead of just 4). Rather than using 4 separate variables, we'll use an array because it is quite easy to handle one variable instead of 1000.

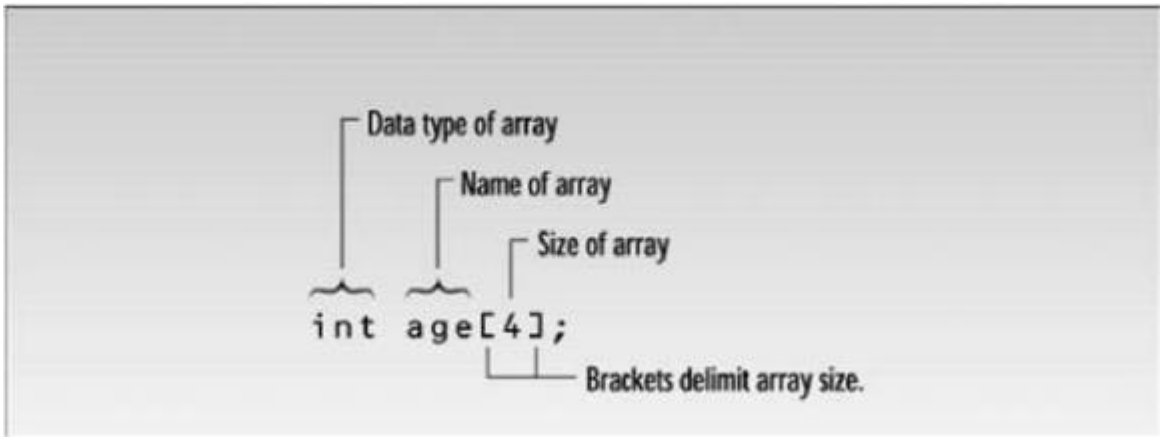
8.3 Declaration of array?

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 4-element array called age of type int, use this statement:

int age[4];



Example 2

```
#include <iostream>
using namespace std;
int main()
{
    int age[4]; //declaration of Array
    age[0]=23; //initialization of Array elements
    age[1]=34;
    age[2]=65;
    age[3]=74;
    return 0;
}
```

On line (5), an array of 4 int is created. Values are assigned to each variable in the array on line (6) through line (9). In memory these are contiguous set of locations shown in following figure.

age [0]	age [1]	age [2]	age [3]
23	34	65	74

Example 3

```
#include <iostream>
using namespace std;
int main()
{
    int age[4]; //array 'age' of 4
    ints for(int j=0; j<4; j++) //get
    4 ages
    {
        cout << "Enter an age: ";
        cin >> age[j]; //access array element
    }

    for(j=0; j<4; j++) //display 4 ages
        cout << "You entered " << age[j] <<
    endl; return 0;
}
```

Here's a sample interaction with the program in example 3

```
Enter an age: 20
Enter an age: 25
Enter an age: 30
Enter an age: 40
You entered 20
You entered 25
You entered 30
You entered 40
```

8.4 Initialization of array?

It is like a variable, an array can be initialized. To initialize an array, we provide initializing values which are enclosed within curly braces in the declaration and placed following an equal's sign after the array name. Here is an example of initializing an integer array.

int age [4] = {23,34,65,74};

age [0]	age [1]	age [2]	age[3]
23	34	65	74

Now how to initialize all the values in array to 0? It can be done by the following statement:

int age [4] = {0};

age [0]	age [1]	age [2]	age [3]
0	0	0	0

8.5 Accessing elements of array?

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. array_name [index];

Following the previous examples in which "age" had 4 elements and each of those elements was of type int, the name which we can use to refer to each element is the following:

age [0] age [1] age[2] age [3]

For example, to store the value 75 in the third element of age, we could write the following statement: age [2] = 75; //**note:** array index start with 0 in c.

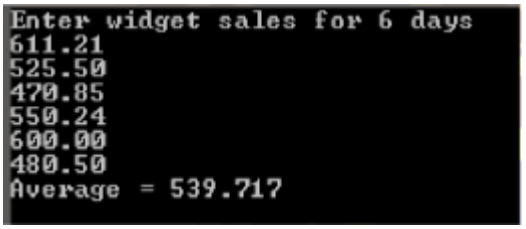
And, for example, to store the value of the third element of age to a variable called a, we could write: int a = age [2];

Here's another example of an array at work. This one, SALES, invites the user to enter a series of six values representing widget sales for each day of the week (excluding Sunday), and then calculates the average of these values. We use an array of type double so that monetary values can be entered.

Example 4

```
#include <iostream>
using namespace std;
int main()
{
    const int SIZE = 6; //size of array
    double sales[SIZE]; //array of 6
    variables cout << "Enter widget sales for
    6 days\n";
        for(int j=0; j<SIZE; j++) //put figures in
            array cin >> sales[j];
    double total = 0;
        for(j=0; j<SIZE; j++) //read figures from
            array total += sales[j]; //to find total
    double average = total / SIZE; // find
    average cout << "Average = " << average <<
    endl; return 0;
}
```

Here's some sample interaction with *sales*:



8.6 Copying arrays

Suppose that after filling our 4 element array with values, we need to copy that array to another array of 4 int.

Example 5

```
#include <iostream>
using namespace std;
int main()
{
    int age[4];
    int same_age[4];
```

```
    int i=0;
    age[0]=23;
    age[1]=34;
    age[2]=65;
    age[3]=74;
    for (;i<4;i++)
       same_age[i]=age[i];
    for (i=0;i<4;i++)
       cout<<same_age[i];
    return 0;
}
```

In the above program two arrays are created: age and same age. Each element of the age array is assigned a value. Then, in order to copy the four elements in age into the same age array, we must do it element by element. We have used for loop to access every element of array note that for loop takes value from 0 to 3.

Note: Like printing arrays, there is no single statement in the language that says, "copy an entire array into another array". The array elements must be copied individually. Thus, If we want to perform any action on an array, we must repeatedly perform that action on each element in the array.

8.7 Dealing with characters using arrays

You can also store characters and other type data (float etc.) in the arrays. Just declare it as we've done in the case with int. There is no difference in dealing with characters except you've to enclose the value in a single quote.

Char ar [3];

ar [0] = 'a'; ar [1] = 'b'…..

8.8 Practice problems

8.8.1 Practice problem 1:

Write a C++ Program for Three Dimensional Array using arrays

```
#include<iostream>
using namespace std;
int main()
{
    int arr[3][4][2] = {
                {
                  {2, 4},
                  {7, 8},
                  {3, 4},
                  {5, 6}
                },
```

```
                    {
                      {7, 6},
                      {3, 4},
                      {5, 3},
                      {2, 3}
                    },
                    {
                      {8, 9},
                      {7, 2},
                      {3, 4},
                      {5, 1}
                    }
                  };
    cout<<"\narr[0][0][0] = "<<arr[0][0][0]<<"\n";
    cout<<"\narr[0][2][1] = "<<arr[0][2][1]<<"\n";
    cout<<"\narr[2][3][1] = "<<arr[2][3][1]<<"\n";
    return 0;
}
```

8.8.2 Practice problem 2:

C++ program to remove characters in string except alphabets

```
#include <iostream>
using namespace std;

int main()
{
   string line;
   int i;
   cout << "Enter any string :: ";
   cin>>line;
   cout << "\nThe Original String is :: " << line<<endl;
   int len = line.size();
   for(i=0;i<len;++i)
   {
      if (!((line[i]>='a' && line[i]<='z') || (line[i]>='A' && line[i]<='Z')))
      {
         line[i]='\0';
      }
   }
   cout << "\nAfter Removing Characters, String is :: " << line<<endl;

   return 0;
}
```

Lab# 09

Recursion

Learning Objectives:

- Concept of recursion
- Concept of memory allocation in recursive functions
- Difference between recursion and iteration
- Disadvantages of recursion over iteration

Outcomes:

- Students should be able to understand the concepts of recursion.
- Students should practice the examples discussed in the tutorial to get a clear image of recursion and to know about the difference in recursion and iteration before performing the tasks.
- In addition, students should also be able to use recursion while passing the function arguments as pointers or by reference.

9.1 Recursion:

In C++, it is possible for the functions to call themselves. A function is called recursive. if a statement within the body of a function calls the same function. Sometimes called circular definition, recursion is thus the process of defining something in terms of itself.

Every function in C++ may be called from any other or itself. Each invocation of a function causes a new allocation of the variables declared inside it.

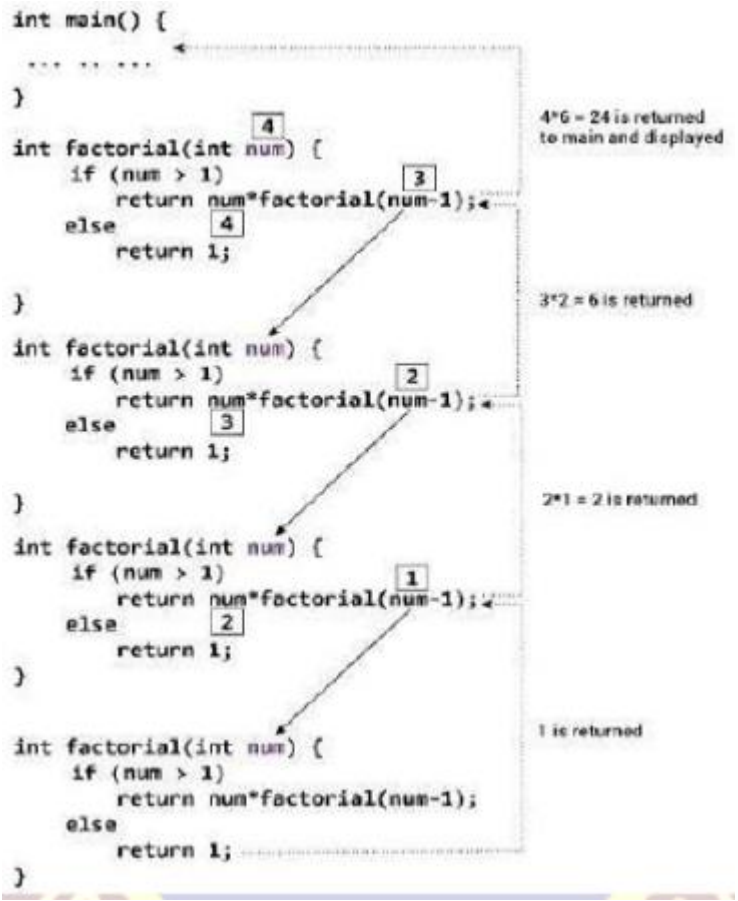
9.2 How recursion works in in C++:

The figure below shows how recursion works by calling itself over and over again.

The recursion continues until some condition is met. To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and other doesn't.

9.2.1 Example 1: Factorial of a number using recursion:

```
Enter a number to find factorial: 4 Factorial of 4:24
```



Explanation: How does this example work?

Suppose the user entered 4, which is passed to the factorial () function.

9.2.1.1 In the first factorial () function, test expression inside if statement is true. The return num*factorial (num-1); statement is executed, which calls the second factorial () function and argument passed is num-1 which is 3.

9.2.1.2 In the second factorial() function, test expression inside if statement is true. The return num*factorial(num-1); statement is executed, which calls the third factorial() function and argument passed is num-1 which is 2.

9.2.1.3 In the third factorial() function, test expression inside if statement is true. The return num*factorial(num-1); statement is executed, which calls the fourth factorial() function and argument passed is num-1 which is 1.In the fourth factorial() function, test expression inside if statement is false. The return 1; statement is executed, which returns 1 to third factorial() function.

9.2.1.4 The third factorial() function returns 2 to the second factorial() function.

9.2.1.5 The second factorial() function returns 6 to the first factorial() function.

9.2.1.6 Finally, the first factorial() function returns 24 to the main() function, which is displayed on the screen.

9.3 What is the difference between direct and indirect recursion?

A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun_new and fun_new calls fun directly or indirectly. The difference between direct and indirect recursion has been illustrated in the image below.

```
// An example of direct recursion
void directRecFun()
{
    // Some code....

    directRecFun();

    // Some code...
}

// An example of indirect recursion
void indirectRecFun1()
{
    // Some code...

    indirectRecFun2();

    // Some code...
}
void indirectRecFun2()
{
    // Some code...

    indirectRecFun1();

    // Some code...
}
```

9.4 How memory is allocated to different function calls in recursion?

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called, and memory is de-allocated and the process continues.

Let us take the example how recursion works by taking a simple function.

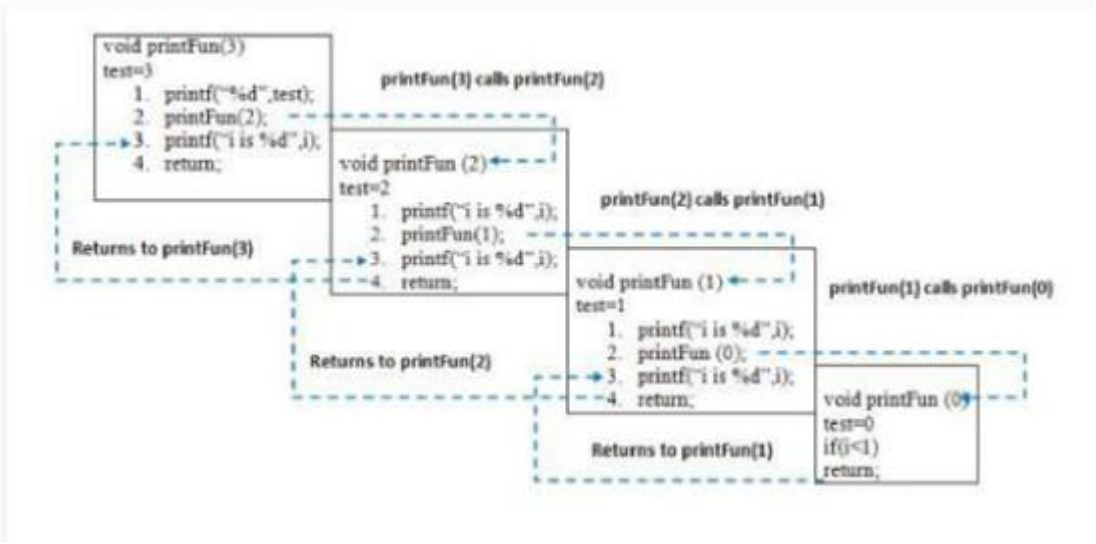
```
// A C++ program to demonstrate working of
// recursion
#include<bits/stdc++.h>
using namespace std;

void printFun(int test)
{
    if (test < 1)
        return;
    else
    {
        cout << test << " ";
        printFun(test-1);    // statement 2
        cout << test << " ";
        return;
    }
}

int main()
{
    int test = 3;
    printFun(test);
}
```

Output :

```
3 2 1 1 2 3
```

When printFun(3) is called from main(), memory is allocated to printFun(3) and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints _3'. In statement 2, printFun(2) is called and memory is allocated to printFun(2) and a local variable test is initialized to 2 and statement 1 to 4 are pushed in the stack.

Similarly, printFun(2) calls printFun(1)and printFun(1) calls printFun(0). printFun(0) goes to if statement and it return to printFun(1). Remaining statements of printFun(1) are executed and it returns to printFun(2) and so on. In the output, value from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in below diagram.



9.5 Recursion and Iteration:

Recursion and iteration both repeatedly execute the set of instructions. Recursion is when a statement in a function calls itself repeatedly. The iteration is when a loop repeatedly executes until the controlling condition becomes false. The primary difference between recursion and iteration is that is a recursion is a process, always applied to a function. The iteration is applied to the set of instructions which we want to get repeatedly executed.

9.5.1 What are the disadvantages of recursive programming over iterative programming?

Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.

9.6 Problems:

9.6.1 Practice Problem 1:

Write a program in C to count the digits of a given number using recursion.

Solution:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
  int n1,ctr;
    printf("\n\n count the digits of a given number :\n");
            printf(" \n"); printf(" Input a number : ");
    scanf("%d",&n1);


    ctr = noOfDigits(n1);

    printf(" The number of digits in the number is : %d
    \n\n",ctr); return 0;
}
int noOfDigits(int n1)
{
    static int ctr=0;

     if(n1!=0)
     {
          ctr++;
         noOfDigits(n1/10);
    }
    return ctr;
}
```

Sample Run:

```
Input a number : 50
The number of digits in the number is : 2
```

9.6.2 Problem 2:

Write a program to multiply two matrix using recursion

Solution:

```
#include <iostream>
#include <cstdlib>
using namespace std;
#define MAX 10

void multiplyMatrix(int [MAX][MAX],int [MAX][MAX]);
int rone,cone,rtwo,ctwo;
int crm[MAX][MAX];

int main()
{

   int arm[MAX][MAX],brm[MAX][MAX],i,j,k;

            printf("\n\n Multiplication of two Matrices :\n");
     printf("                                    \n");

   printf(" Input number of rows for the first matrix : ");
   scanf("%d",&rone);
   printf(" Input number of columns for the first matrix : ");
   scanf("%d",&cone);

   printf(" Input number of rows for the second matrix : ");
   scanf("%d",&rtwo);
   printf(" Input number of columns for the second matrix : ");
   scanf("%d",&ctwo);

   if(cone!=rtwo)
   {
       printf("\n Check col. of first and row of second matrix.");
       printf("\n They are different. Try again.\n");
     }
    else
```

```
{

   printf("\n Input elements in the first matrix :\n");
   for(i=0;i<rone;i++){
   for(j=0;j<cone;j++){
                    printf(" element - [%d],[%d] : ",i,j);
       scanf("%d",&arm[i][j]);}}

   printf(" Input elements in the second matrix :\n");
   for(i=0;i<rtwo;i++){
   for(j=0;j<ctwo;j++){
                    printf(" element - [%d],[%d] : ",i,j);
       scanf("%d",&brm[i][j]);}}

   printf("\n Here is the elements of First matrix : \n");
   for(i=0;i<rone;i++)
   {
   printf("\n");
   for(j=0;j<cone;j++)
   {
       printf(" %d\t",arm[i][j]);
   }
   }

   printf("\n Here is the elements of Second matrix : \n");
   for(i=0;i<rtwo;i++)
   {
   printf("\n");
   for(j=0;j<ctwo;j++)
   {
       printf(" %d\t",brm[i][j]);
   }
   }
   multiplyMatrix(arm,brm);
}

printf("\n The multiplication of two matrix is : \n");
for(i=0;i<rone;i++)
{
   printf("\n");
   for(j=0;j<ctwo;j++)
   {
       printf(" %d\t",crm[i][j]);
   }
}
printf("\n\n");
```

```
  return 0;

}

void multiplyMatrix(int arm[MAX][MAX],int brm[MAX][MAX])
{

    static int sum,i=0,j=0,k=0;

    if(i<rone)
    { //row of first
    matrix if(j<ctwo)
    { //column of second
        matrix if(k<cone)
        {
            sum=sum+arm[i][k]*brm[k][j];
            k++;
            multiplyMatrix(arm,brm);
        }
        crm[i][j]=sum;
            sum=0;
            k=0;
            j++;
            multiplyMatrix(arm,brm);
    }
    j=0;
    i++;
    multiplyMatrix(arm,brm);
    }
}
```

Sample Run:

Input number of rows for the first matrix : 2
Input number of columns for the first matrix : 1
Input number of rows for the second matrix : 1
Input number of columns for the second matrix : 2

Input elements in the first matrix :
element - [0],[0] : 1
element - [1],[0] : 2
Input elements in the second matrix :
element - [0],[0] : 3
element - [0],[1] : 4

Here is the elements of First matrix :

1
2
Here is the elements of Second matrix :

3 4
The multiplication of two matrix is :

3 4
6 8

Lab# 10

Pointers

Objectives:

- To understand the working of computer memory
- What are pointers?
- Referencing operator and dereferencing operator
- Working of pointers
- Pointer's arithmetic
- Passing pointers as function arguments
- Function pointers
- Pointers and Arrays
- Memory leaks and how to avoid it?

Outcomes:

- Students should be able to understand the concept of memory allocation to variables.
- Students should be able to understand the concept of pointers and be able to allocate different memory locations to pointers.
- They should be able to handle pointers arithmetic and also know how to use pointers as functions arguments and to access the array data using pointers.
- Students should be familiar with the concept of memory leaks, what are their disadvantages, how they occur and how we can avoid it while using the pointers.

10.1 Introduction?

In this lab we will be discussing pointers in detail. This is one of the most important concepts in C++ language. Pointers are used everywhere in C++, so if you want to use the C++ language fully you have to have a very good understanding of pointers. They have to become *comfortable* for you.

C++ uses pointers in three different ways:

- C++ uses pointers to create dynamic data structures -- data structures built up from blocks of memory allocated from the heap at run-time.

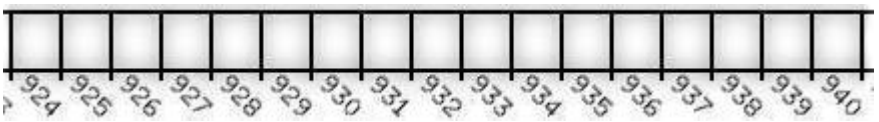
- C++ uses pointers to handle variable parameters passed to functions.

- Pointers in C++ provide an alternative way to access information stored in arrays. Pointer techniques are especially valuable when you work with strings. There is an intimate link between arrays and pointers in C++.

To fully grasp the concept of pointers all you need is the concept and practice of pointers. Before talking about pointers let's talk a bit about computer memory.

10.2 Computer Memory:

Essentially, the computer's memory is made up of bytes. Each byte has a number, an address, associated with it. The picture below represents several bytes of a computer's memory. In the picture, addresses 924 thru 940 are shown.



10.2.1 Variable and Computer Memory:

A variable in a program is something with a name, the value of which can vary. The way the compiler handles this is that it assigns a specific block of memory within the computer to hold the value of that variable. The size of that block depends on the range over which the variable is allowed to vary. For example, on 32 bit PC's the size of an integer variable is 4 bytes. On older 16-bit PCs integers were 2 bytes.

10.2.2 Example:

Code:

```
#include<iostream>

using  namespace  std;

int main()


{
float fl=3.14;

cout<<fl;

cin>>fl;

return 0;

}
```

Output:

How It Works: (Explanation)

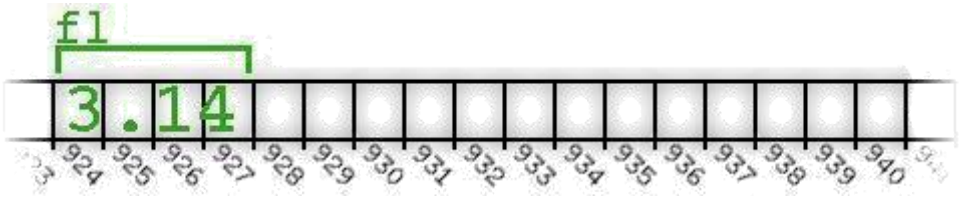
At line (4) in the program above, the computer reserves memory for fl. Depending on the computer's architecture, a float may require 2, 4, 8 or some other number of bytes. In our example, we'll assume that a float requires 4 bytes.

When fl is used in line (5), two distinct steps occur:

10.2.2.1 The program finds and grabs the address reserved for fl in this example 924.

10.2.2.2 The contents stored at that address are retrieved.

The illustration that shows 3.14 in the computer's memory can be misleading. Looking at the diagram, it appears that "3" is stored in memory location 924, "." is stored in memory location 925, "1" in 926, and "4" in 927. Keep in mind that the computer converts the floating-point number 3.14 into a set of ones and zeros. Each byte holds 8 ones or zero. So, our 4-byte float is stored as 32 ones and zeros (8 per byte times 4 bytes). Regardless of whether the number is 3.14, or -273.15, the number is always stored in 4 bytes as a series of 32 ones and zeros.



10.3 Pointers:

In C++ a pointer is a variable that points to or references a memory location in which data is stored. A pointer is a variable that points to another variable. This means that a pointer holds the memory address of another variable. Put another way, the pointer does not hold a value in the traditional sense; instead, it holds the address of another variable. A pointer "points to" that other variable by holding a copy of its address. Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address and that address points to a value.

10.3.1 Pointer declaration:

A pointer is a variable that contains the memory location of another variable. The syntax is as shown below. You start by specifying the type of data stored in the location identified by the pointer. The asterisk tells the compiler that you are creating a pointer variable. Finally you give the name of the variable.

Data_type *variable_name

Such a variable is called a pointer variable (for reasons which hopefully will become clearer a little later). In C++ when we define a pointer variable, we do so by preceding its name with an asterisk. In C++ we also give our pointer a type which, in this case, refers to the type of data stored at the address we will be storing in our pointer. For example, consider the variable declaration:

```
int    *ptr;
int k;
```

ptr is the name of our variable (just ask is the name of our integer variable). The '*' informs the compiler that we want a pointer variable, i.e. to set aside however many bytes is required to store an address in memory. The int says that we intend to use our pointer variable to store the address of an integer.

10.3.2 Referencing Operator

Suppose now that we want to store in ptr the address of our integer variable k. To do this we use the unary & operator and write:

```
ptr = &k;
```

What the & operator does is retrieve the address of k, and copy that to the contents of our pointer ptr. Now, ptr is said to "point to" k.

10.3.3 Dereferencing operator

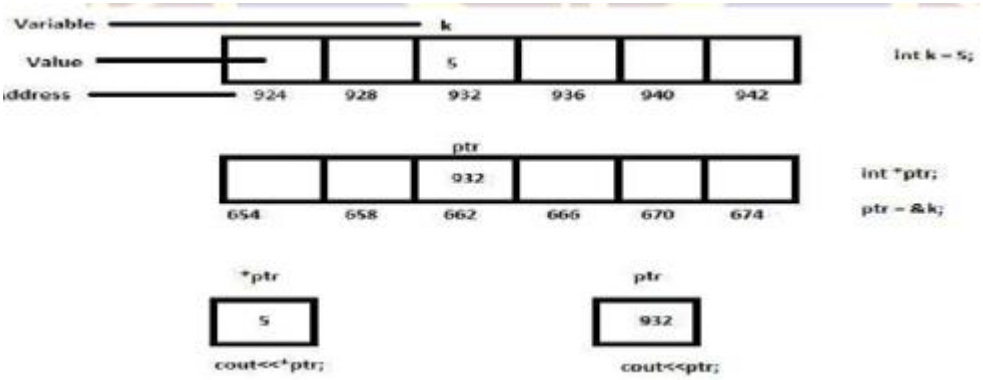
The "dereferencing operator" is the asterisk, and it is used as follows:

```
*ptr = 7;
```

will copy 7 to the address pointed to by ptr. Thus, if ptr "points to" (contains the address of) k, the above statement will set the value of k to 7. That is, when we use the '*' this way we are referring to the value of that which ptr is pointing to, not the value of the pointer itself.
Similarly, we could write:

```
cout<<*ptr<<endl;
```

to print to the screen the integer value stored at the address pointed to by ptr;.
Here is graphical representation of Pointers.



This Example will be very helpful in understanding the pointers. Understand thoroughly how it works and then proceed.

10.3.4 Example:

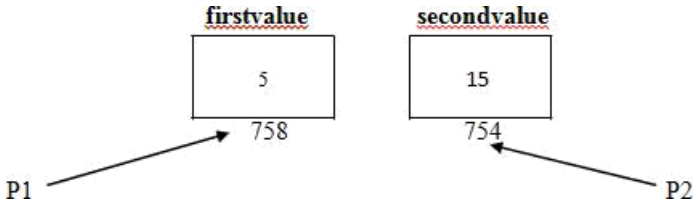
Code:

```
#include<iostream>
using namespace std;
int main ()
{
int firstvalue = 5, secondvalue = 15;
int * p1, * p2;

 p1 = &firstvalue;                  // p1 = address of firstvalue
p2 = &secondvalue;                 // p2 = address of secondvalue
*p1 = 10;                          // value pointed by p1 = 10
*p2 = *p1;                         // value pointed by p2 = value pointed by p1
p1 = p2;                           // p1 = p2 (value of pointer is copied)
*p1 = 20;                          // value pointed by p1 = 20
cout<<"First Value is " << firstvalue<<endl;
cout<<"Second Value is " <<secondvalue<<endl;
return 0;
}
```

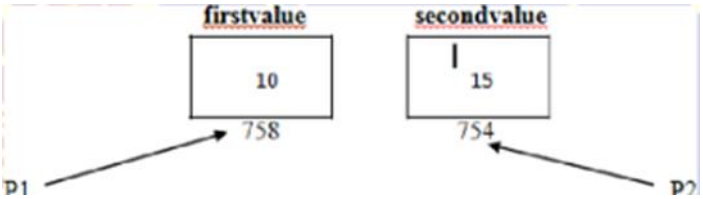
Output:

How it Works:

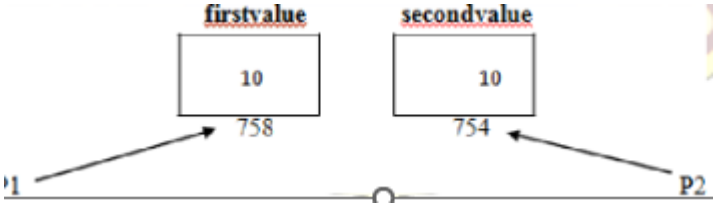
Here in this code we are trying to play with memory and address of our variables for the better understanding of Pointers. On line number 5 we have two integer variables (i.e first value and second value). Both are assigned values of 5 and 15 respectively. On line number 6 we have two integer pointer variables (i.e p1 and p2). Both are assigned addresses of variables in line 5 first value and second value respectively in line 7 and 8.



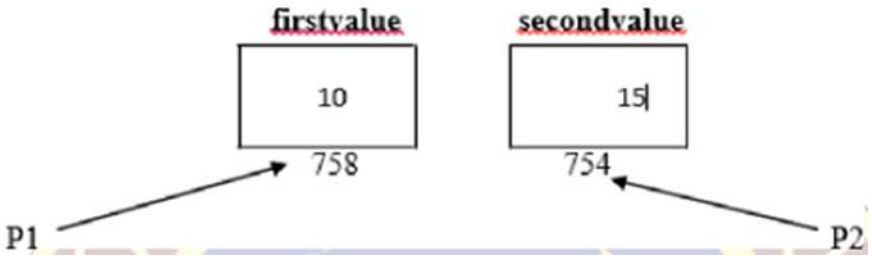
In line 9 we see that *p1 is assigned value 10. This means that 10 should be copied in the variable, which is lying on an address to which p1 is pointing. We know that p1 is pointing to address of firstvalue. So line 9 results in assigning firstvalue the value of 10.



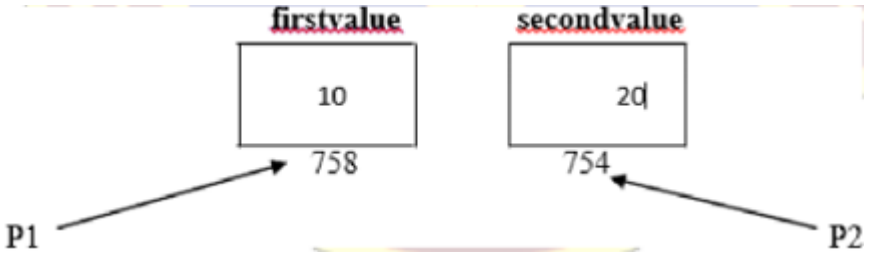
In line 10 we encounter another assignment which says that value of variable pointed by p2 should be replaced with the value of variable pointed by p1. So now secondvalue is assigned with value 10 as well.



Well the assignment in line 11 is a bit confusing but very simple, all this assignment is doing is that now p1 is pointing to the same address as p2. So now we can say p1 and p2 are pointing at same address.



In line 12 we see that *p1 is assigned value 20. This means that 10 should be copied in the variable, which is lying on an address to which p1 is pointing. We know that p1 is now pointing to address of secondvalue because in last line we pointed p1 to the address being pointed by p2. So line 12 results in assigning secondvalue the value of 20.



Now when we print the value of first value and second value it prints 10 for first value and 20 for second value, which is right due to the reasons explained above.

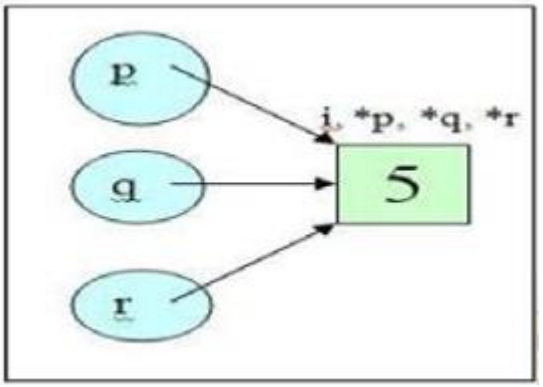
12.3.5 Pointers: Pointing to the Same Address

Here is a cool aspect of C++: Any number of pointers can point to the same address. For example, you could declare p, q, and r as integer pointers and set all of them to point to i, as shown here:

Example

```
int i;

int *p, *q, *r;
p = &i;

q = &i;

r = p;
```

Note that in this code, r points to the same thing that p points to, which is i. You can assign pointers to one another, and the address is copied from the right-hand side to the left-hand side during the assignment. After executing the above code, this is how things would look.



The variable i now has four names: i, *p, *q and *r. There is no limit on the number of pointers that can hold (and therefore point to) the same address.

10.4 Pointer Arithmetic's

Like other variables pointer variables can be used in expressions. For example if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

Example

```
y=*p1  *  *p2;
sum=sum+*p1;
z= 5 - *p2/*p1;
*p2= *p2 + 10;
```

C++ allows us to add integers to or subtract integers from pointers as well as to subtract one pointer from the other. We can also use shorthand operators with the pointers p1+=; sum+=*p2; etc, We can also compare pointers by using relational operators the expressions such as p1 >p2 , p1==p2 and p1!=p2 are allowed.

When an integer is added to, or subtracted from, a pointer, the pointer is not simply incremented or decremented by that integer, but by that integer times the size of the object to which the pointer refers. The number of bytes depends on the object's data type.

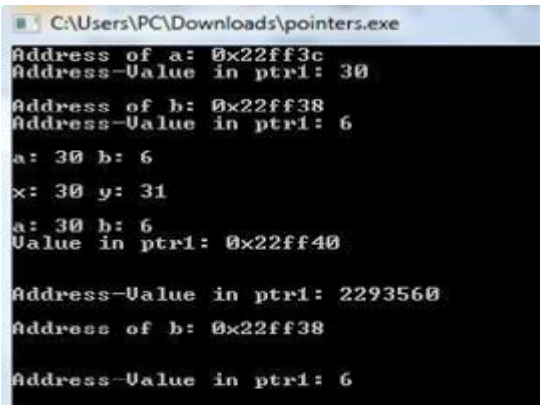
/*Program to illustrate the pointer expression and pointer arithmetic*/

10.4.1 Example:

Example

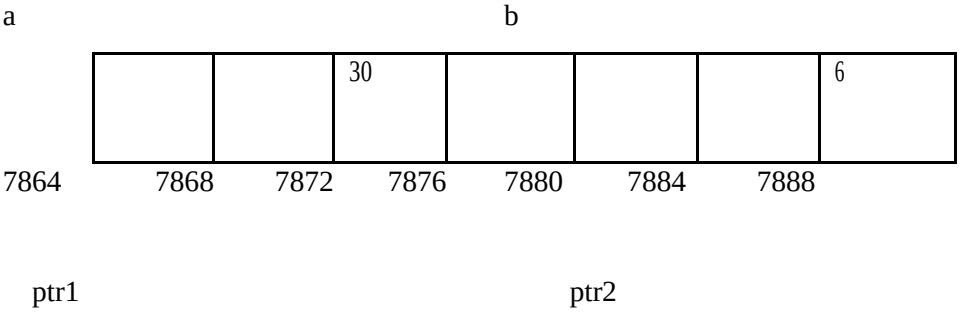
```
#include<iostream>
using namespace std;
 int main()
 {
 int  *ptr1,*ptr2;
 int    a,b,x,y,z;
 a=30;b=6;
 ptr1=&a;
 ptr2=&b
 x=*ptr1 + *ptr2 - 6; y=6
 - *ptr1 / *ptr2 +30;
 cout<<"Address of a: "<<ptr1<<endl;
 cout<<"Address-Value in ptr1: "<<*ptr1<<endl<<endl; //The comment
 value is the value of ptr1 (address of the a)
 cout<<"Address of b: "<<ptr2<<endl;
 cout<<"Address-Value in ptr1: "<<*ptr2<<endl<<endl; //The comment
 value is the value of ptr2 (address of the b)
 cout<<"a: " <<a<<" b: "<<b<<endl<<endl;
 //Simply prints the value of a and b
 cout<<"x: " <<x<<" y: "<<y<<endl<<endl; //Simply prints the value of
 x and y.
 ptr1=ptr1 + 1; // adds 4 in address of ptr1. (1*4 = 4)
 ptr2= ptr2;
 cout<<"a: " <<a<<" b: "<<b<<endl;
 //Simply prints the value of a and b
 cout<<"Value in ptr1: "<<ptr1<<endl<<endl; // 2293564 //The comment
 value is the new memory location value of ptr1
 cout<<"\nAddress-Value in ptr1: "<<*ptr1<<endl<<endl; // garbage value
 //The comment value is the new value of ptr1 (garbage value)
 cout<<"Address of b: "<<ptr2<<endl<<endl;
 cout<<"\nAddress-Value in ptr1: "<<*ptr2<<endl<<endl;
 cin>>a;

}
```

Here note that adding some thing in *ptr1 changes the value of the address stored in ptr. However adding some thing in ptr will change the address it is pointing to. Printing ptr1 after adding 1 in it gives different address as it has changed by 4 Bytes.

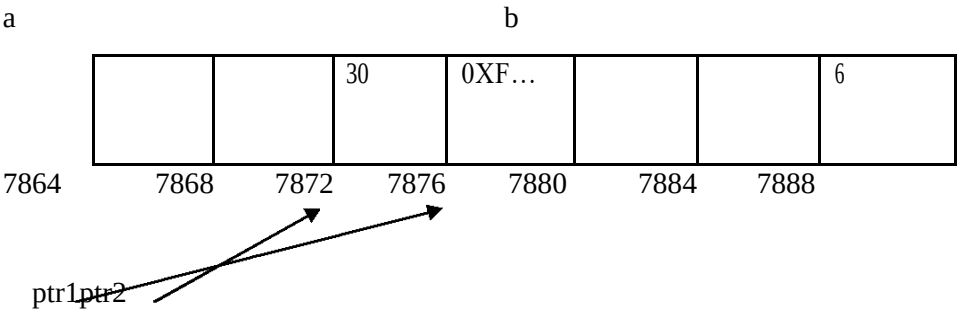


How it Works:

This code explains all the rules related to arithmetic of pointers. From line 1 to line 11, it is simply adding, subtracting and like manipulating with the pointers and variables. After all the manipulations and arithmetic it started printing values of pointers and other simple variables till line 12.



At line 18 it adds 1 to ptr1. Mostly people think that this will change the address of the pointer, but they are totally wrong. Remember pointer is pointing to an address. This addition does not change the address of the pointer, infect it changes the value of the pointer (not the value of the address pointer is pointing at.). ptr1 has the address of variable of variable a . So, it adds 1 *4 Bytes = 4bytes in the address of a which is stored in ptr1. Whereas ptr2 points at the same value as before due to the assignment of line 19.



Line 20 prints the same value as was printed by Line 16, because values of the variable were never changed, in fact ptr1's value which was address of a was changed. Now Line 21 will print.

the value stored in ptr1; which is address of memory 4 bytes ahead of variable a. Line 22 is trying to print the value at address, ptr1 is now pointing to, which was never assigned any value.

10.4.2 Sending Pointers as Arguments to Functions

When we pass pointers to some function, the addresses of actual arguments in the calling function are copied into the arguments of the called function. This means that using these addresses we would have access to the actual arguments and hence we would be able to manipulate them. The following program illustrates this fact. Try this out:

10.4.3 Example:

Example

```
#include<iostream>
void swap(int *,int *);
using  namespace  std;
int main( )

{
int a = 10, b = 20;
int *p, *q;
p = &a;
q = &b; swap(
p, q) ;
cout<<"a:    "<<a<<"b:    "<<b<<endl;
cin>>a;

}
void swap( int *x, int *y )

{
int  t = *x ;
*x = *y;
*y = t;
}


Output:

A=20 b=10
```

Note that this program manages to exchange the values of a and b using their addresses stored in x and y.

10.5 Function Pointers

A function pointer is a variable that stores the address of a function that can later be called through that function pointer. A useful technique is the ability to have pointers to functions. Their declaration is easy: write the declaration as it would be for the function, say

```
int func(int a, float b);
```

And simply put brackets around the name and a * in front of it: that declares the pointer. Because of precedence, if you don't parenthesize the name, you declare a function returning a pointer:

```
/* function returning pointer to int
*/ int *func(int a, float b); //Wrong

/* pointer to function returning int */

int (*func)(int a, float b);
```

Once you've got the pointer, you can assign the address of the right sort of function just by using its name: like an array, a function name is turned into an address when it's used in an expression. You can call the function as:

```
(*func)(1,2);
```

10.5.1 Example:

Code:

```
#include<iostream>
using namespace std;
void func(int);

int main(){
void (*fp)(int);
fp = func;
(*fp)(1);
cout<<endl;
fp(2);
system("PAUSE");
return 0;
}
Void func(int arg)
{
cout<<arg<<endl;
}
```

10.6 Pointers and arrays

The concept of arrays is related to that of pointers. In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type. For example, consider these two declarations:

```
int myarray [20];
int * mypointer;
```

The following assignment operation would be valid:

```
mypointer = myarray;
```

After that, mypointer and myarray would be equivalent and would have very similar properties. The main difference being that mypointer can be assigned a different address, whereas myarray can never be assigned anything, and will always represent the same block of 20 elements of type int. Therefore, the following assignment would not be valid:

```
myarray = mypointer;
```

Let's see an example that mixes arrays and pointers:

10.6.1 Example:

Code:

```
#include    <iostream>

using  namespace  std;

int main ()

{
int numbers[5];
int * p;
p = numbers;     *p = 10;
p++; *p = 20;

p = &numbers[2]; *p = 30;
p = numbers + 3; *p = 40;
p = numbers;     *(p+4) = 50;
for (int n=0; n<5; n++)
cout  <<  numbers[n]  <<  ",  ";
return 0;}
```

Output:

```
10, 20, 30, 40, 50,
```

Pointers and arrays support the same set of operations, with the same meaning for both. The main difference is that pointers can be assigned new addresses, while arrays cannot.

In the chapter about arrays, brackets ([]) were explained as specifying the index of an element of the array. Well, in fact these brackets are a dereferencing operator known as offset operator. They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced. For example:

```
a[5] = 0;  // a [offset of 5] = 0
```

```
*(a+5) = 0;       // pointed by (a+5) = 0
```

These two expressions are equivalent and valid, not only if a is a pointer, but also if a is an array. Remember that if an array, its name can be used just like a pointer to its first element.

10.7 Memory leakage in C++ and How to avoid it:

Memory leakage occurs in C++ when programmers allocates memory by using new keyword and forgets to de-allocate the memory by using delete() function or delete[] operator. One of the most memory leakages occur in C++ by using wrong delete operator.

The delete operator should be used to free a single allocated memory space, whereas the delete [] operator should be used to free an array of data values.

10.7.1 Disadvantage with memory leakage:

If a program has memory leaks, then its memory usage is satirically increasing since all systems have limited amount of memory and memory is costly. Hence it will create problems.

10.7.2 Example:

Code:

```
/   Program  with  memory  leak
#include <bits/stdc++.h>
using namespace std;

/   function with memory
leak void
func_to_show_mem_leak()
{
    int* ptr = new int(5);

    /  body

    /  return without deallocating ptr return;
}

/   driver  code
int main()
{

    /  Call the function
    /    to get the memory leak
    func_to_show_mem_leak();

    return 0;
}
```

10.8 How to avoid Memory Leak?

- Instead of managing memory manually, try to use smart pointers where applicable.
- use std::string instead of char *. The std::string class handles all memory management internally, and it's fast and well-optimized.
- Never use a raw pointer unless it's to interface with an older lib.
- The best way to avoid memory leaks in C++ is to have as few new/delete calls at the program level as possible – ideally NONE.
- Allocate memory by new keyword and deallocate memory by delete keyword and write all code between them.

10.8.1 Example to handle memory leaks:

Code:

```
/ CPP program to
/  illustrate how to avoid
/  memory leak
#include <bits/stdc++.h>
using namespace std;

/  function to see memory handling
void func_to_handle_mem_leak()
{
    int* ptr = new int(5);

    /  body

    /  Now delete pointer ptr using delete
    delete (ptr);
}

/   Driver
code int
main()
{

    /  Call function to handle
    /       thememoryleak
    func_to_handle_mem_leak()

        return 0;
}
```

10.9 Practice Problem:

10.9.1 Practice problem 1:

Given the string "A string." Print on one line the letter on the index 0, the pointer position and the letter t. undate the pointer to pointer +2. Then, in another line print the pointer and the letters r and g of the string (using the pointer).

Solution:

Code:

```
#include    <iostream>
using namespace std;

int main(void) {
   char str[] = "A string."; char
   *pc = str;

   cout << str[0] <<' '<< *pc <<'
   '<<pc[3]<<"\n"; pc += 2;
   cout <<*pc<<' '<< pc[2] <<' '<< pc[5];

   return 0;
}
```

10.9.2 Practice problem 2:

Write a program in C to print a string in reverse using a pointer.

Solution:

```
#include <iostream>
using namespace std;

int main()
{
    char str1[50];
    char revstr[50];
    char *stptr = str1;
    char *rvptr = revstr;
    int i=-1;
    cout<<"\n\n Pointer : Print a string in reverse order :\n";
    cout<<"                                                \n";
    cout<<" Input a string : ";
    cin>>str1;
    while(*stptr)
```

```
    {
     stptr++;
     i++;
    }
    while(i>=0)
    {
     stptr--;
     *rvptr = *stptr;
     rvptr++;
     --i;
    }
    *rvptr='\0';
    cout<<" Reverse of the string is :
    ",revstr; return 0;
}
```

Lab # 11

File Handling

Learning Objectives:

- Streams in C++
- Input/output with files
- Open a file
- Closing a file
- Checking for end of file

Outcomes:

- Students should be able to understand the concept of streams in C++.
- Students should be familiar with the concept of input/output with files, opening a file, closing a file and checking for end of file.

11.1 Streams in C++

A stream is basically flow of data from or to the program. There are two types of streams.

1. Input Stream- An input stream is flow of data into the program (e.g. cin)

2. Output Stream- An output stream is flow of data out of the program (e.g. cout)

11.1.1 What does iostream.h contain?

```
#include <iostream>
```

When we write this at the beginning of the program, it basically means that we are instructing the compiler to include the contents of the file iostream.h into our source file before compiling.

iostream.h (short for input/output stream) has certain classes and objects defined within the file, which we can use in our program directly. It contains the following classes for input/output of data:

- ios- Base class for all other classes for input and output
- istream- Derived from ios. Used to input data into the program (input stream). cin is an object of this class.

- ostream- Derived from ios. Used to output data out of the program (output stream). cout is an object of this class.

11.1.2 Meaning of cin>> and cout<<

When we write the statement:

```
cout << "Hello";
```

the << (insertion operator) passes the value following it (i.e a char * pointing to the string “Hello”) to the object cout, which in turn acts as an output stream, and hence passes it on to the screen from the program.

Similarly, when we write the statement:

```
string     s;
cin >> s;
```

The >> (extraction operator) passes each character of the value the user enters to the object cin, which in turn acts as an input stream, and hence passes it from the keyboard into the program.

The problem with cout<< is that it only allows flow of data from the program to the screen. Similarly, cin>> only allows flow of data from the keyboard into the program.

In order to output data from our program to a physical file, or to input from a physical file into our program, we use the concept of file-handling. We will work with text files for now.

11.2 Input/output with files

C++ provides the following classes to perform output and input of characters to/from files:

#include<fstream>

The file fstream.h contains all the contents of iostream.h (istream class, ostream class etc.), along with certain classes of its own. It contains:

- ifstream- Class derived from istream. Allows input from other sources apart from just the keyboard.
- ofstream- Class derived from ostream. Allows output other than just the screen.
- fstream- a class which contains all the contents of ifstream, and ofstream through. multiple inheritance.

11.2.1 Writing to a physical file

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
fstream myfile; //Declare an object of class fstream
myfile.open("abc.txt",ios::out); //open the file using a stream
myfile << "Hello" << endl; //write to file using the
object declared
myfile << "World" << endl;
myfile.close(); //close the file
return 0;
}
```

This code creates a file called abc.txt and inserts words into it in the same way we are used to do with cout, but using the file stream `myfile` instead.

11.2.2 Reading from a physical file

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
fstream myfile; //Declare an object of class fstream

myfile.open("story.txt",ios::in); //open the file using a
stream string s;
myfile >> s; //read from file using the object
declared myfile.close(); //close the file.
return 0;
}
```

This piece of code reads from a physical file. But let's go step by step:

11.3 Open a file

The open() function is used to opening an object in a particular stream. It takes two arguments, the name of the file (datatype char*), and the mode in which to open the file.

- ios::out- Output mode to take data from program out to a file.
- ios::in- Input mode to take data from file into the program.
- ios::app- Append mode to add data to an existing file. If this is not specified, opening the file in output mode will overwrite the existing contents of the file, if the file already exists.
- ios::trunc- Truncate mode to indicate that the file, if it exists, should be overwritten on.

The mode can be have more than one value, separated by the | operator. For example:

```
myfile.open("abc.txt", ios::out | ios::app);
```

Each one of the open() member functions of the classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even ifa mode that does not include them is passed as second argument to the open() member function.

The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not
combined.

11.4 Additional ways to read from file

If you have an object which opens a file in input mode, writing

F >> s;

Is similar to doing cin >> s. Just like cin >>, F >> would read just a word from the file, as it takes any white-space (i.e space, tab or new line) as a string terminator. There are a few alternate ways of inputting data:

11.4.1 getline()

It is a member function of class istream, and is used to input a line from the user. For example:

```
cin.getline(str,80);
```

It takes 3 arguments:

1. The name of the string (usually a character array and not a cstyle string)
2. Maximum number of characters in the string (it is important to remember that the last character must also be string terminator)
3. A delimeter which terminates string input as soon as it is encountered (default is the enter key) i.e as soon as the user enters the enter key, the string is terminated.

We can also use getline() with fstream objects:

```
fstream     F;
char str[80];
F.getline(str,80);
```

11.5 get()

It is a member function of class istream, and is used to input a character from the user. For example:

```
char              ch;
ch=cin.get();
```

Similarly it can also be used with fstream objects:

```
ch=F.get();
```

11.6 Closing a file

When we are finished with our input and output operations on a file we shall close it so that its resources

become available again. In order to do that we have to call the stream's member function close(). This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

```
myfile.close();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function close().

11.7 Checking for end of file

When reading the entire contents of a file, it is often required to check whether we have reached the end of a file or not.

11.7.1 Function eof()- end of file

This is a function which returns true if we have reached the end of a file.

```
fstream f;
f.open("story.txt",ios::in);
string s;
f>>s;
while(!f.eof())
{
     cout<<s<<endl;
     f>>s;
}
```

11.8 Practice Problem

11.8.1 Practice problem 1:

Write a C++ program to maintain house record using file handling

Solution:

```
#include<fstream>
#include<iostream>
#include<string.h>
#include<stdio.h>
```

```
using namespace std;

int opt;

class housing
{
  int hno,income;
  char name[20],type[20];
  public:
      void assign()
      {
            if(income<15000)
            strcpy(type,"LIG");
            else if(income>=15000)
            strcpy(type,"MIG");
            else if(income>=25000)
            strcpy(type,"HIG");
      }

      void input()
      {
            cout<<"\n Enter House Number: ";
            cin>>hno;
            cout<<"\n House Name: ";
            cin>>name;
            cout<<"\n Annual Income: ";
            cin>>income;
            assign();
      }

       void output()
      {
             cout<<"House Number: "<<hno<<"\n"<<"House Name:
"<<name<<"\n"<<"Annual
Income: "<<income<<"\n"<<"Type: "<<type;
      }

      int retno()
      {
            return hno;
      }
};
```

```
int main()
{
     housing h,h1;
     fstream f;
     int hono;

     while(true)
     {
         cout<<"\n 1: Add Record"<<"\n 2: Modify Record"<<"\n 3:
Display Records"<<"\n 4: Exit\n"<<endl;
         cin>>opt;

         if(opt==1)
         {
              char ch='y';

f.open("C:\\Users\\acer\\Documents\\file4.txt",ios::out|ios::binary|
ios::app);
              while(ch=='y')
              {
                   cout<<"\n Enter Details: ";
                   h.input();
                   f.write((char*)&h,sizeof(h));
                   cout<<"\n Want to Enter More? y/n:
                   "<<endl; cin>>ch;
              }
             f.close();
         }

         if(opt==2)
         {
              cout<<"\n Enter House No of Record to be modified:
              "; cin>>hono;

f.open("C:\\Users\\acer\\Documents\\file4.txt",ios::in|ios::out|ios:
:binary|ios::ate);
              f.seekg(0);

              while(f.read((char*)&h,sizeof(h)))
              {
                   if(h.retno()==hono)
```

```
                    {
                         cout<<"\n New Value: ";
                         h1.input();
                         f.seekp(-sizeof(h),ios::cur);
                         f.write((char*)&h1,sizeof(h1));
                    }
               }
              f.close();
          }

          if(opt==3)
          {

 f.open("C:\\Users\\acer\\Documents\\file4.txt",ios::in|ios::binary);
               f.seekg(0);

               while(f.read((char*)&h,sizeof(h)))
               h.output();

               f.close();
          }

          if(opt==4)
              exit(0);

         cout<<"\nPress any key to continue. ......";

     }

     return 0;
 }
```

11.8.2 Practice problem 2:

Write a C++ program to encrypt file using file handling

```
#include<iostream>
#include<fstream>
#include<stdio.h>
#include<stdlib.h>
using namespace std;

int main()
```

```
{

        char fname[20], ch, choice;
        fstream fps, fpt;
        cout<<"Enter file name to encrypt ::
        "; cin>>fname;

        fps.open(fname);

        if(!fps)
        {
                cout<<"\nError in opening file..!!";
                cout<<"\nPress any key to exit...";

                exit(1);
        }

        fpt.open("C:\\Users\\acer\\Documents\\file5.txt");
        if(!fpt)
        {
                cout<<"\nError in creating file5.txt
                file..!!"; fps.close();
                cout<<"\nPress any key to exit...";

                exit(2);
        }
        while(fps.eof()==0)
        {
                fps>>ch;
                ch=ch+100;
                fpt<<ch;
        }

        fps.close();
        fpt.close();
        fps.open(fname);
        if(!fps)
        {
                cout<<"\nError in opening source file..!!";
                cout<<"\nPress any key to exit...";

                exit(3);
```

```
        }
        fpt.open("C:\\Users\\acer\\Documents\\file5.txt");
        if(!fpt)
        {
                cout<<"\nError in opening temp.txt
                file...!!"; fps.close();
                cout<<"\nPress any key to exit...";

                exit(4);
        }
        while(fpt.eof()==0)
        {
                fpt>>ch;
                fps<<ch;
        }
        cout<<"\nFile "<<fname<<" encrypted successfully..!!";
        cout<<"\n\nPress any key to exit...";
        fps.close();
        fpt.close();
        return 0;
}
```